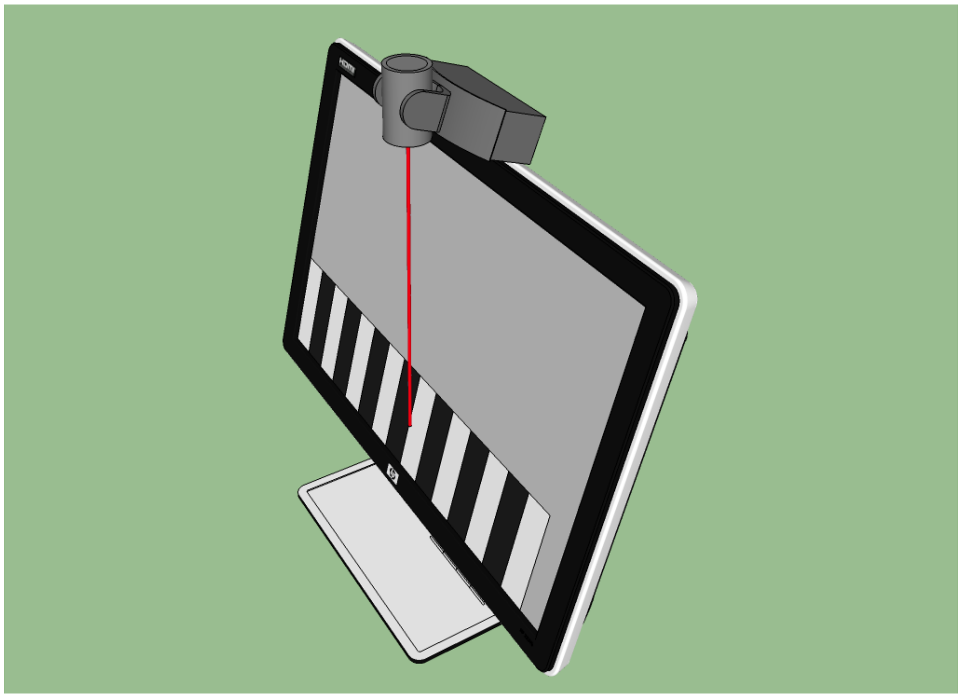
PIANO PLAYER WITH VIRTUAL TOUCH KEYBOARD



5/14/2014

Project Report

CSEE W4840 EMBEDDED SYSTEMS

Spring 2014 Prof. Stephen A. Edwards

By Daran Cai (dc2946@columbia.edu)

Linjun Kuang (lk2578@columbia.edu)

Wei Xia (wx2147@columbia.edu)

Wenyuan Zhao (wz2262@columbia.edu)

Contents

Piano player with Virtual touch keyboard

PROJECT REPORT

ABSTRACT

For our project, we implement a piano player. Unlike the traditional piano player software and applications, we use a normal screen and a camera to realize a virtual touch screen and a virtual keyboard. We use a Sockit board, with an ARM Cortex- A9 processor on it, a camera (Logitech HD Portable 1080p Webcam C615 with Autofocus), a VGA display monitor and a pair of speakers.

INTRODUCTION

The key point of our project is how to use a camera and a normal screen to realize the function of a touch screen. The basic idea is that camera is used to capture the image of fingers and the screen in real time, and then using computer vision algorithm to recognize the relative position of the finger on screen. Using this relative position, we can achieve the function of the touch screen.

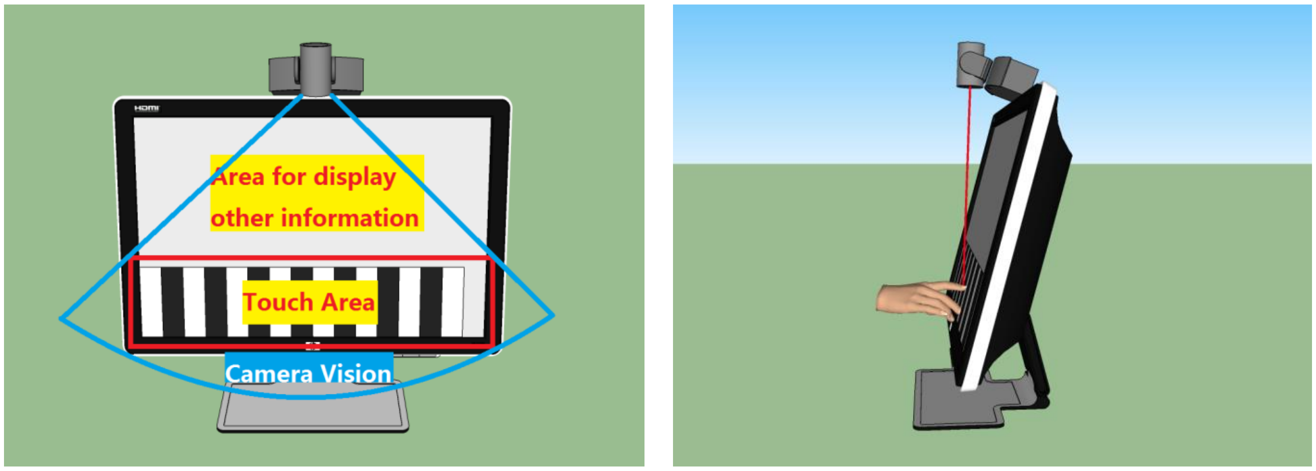


FIGURE 1: THE INTRODUCTION OF THE KEY CONCEPT

As for the user interface, we used sprite graphics and also directly assigned RGB value for some area, especially for the keyboard area. In order to make our project more user friendly and complete, we add some instructions to guide users in our user interface, and for this part, we use bit maps to display words and sentences on screen. When users press those virtual keys, the corresponding sounds will come out. Also, our system support two keys pressing together.

Except for the basic free mode, in which users can play whatever they like, we also provide auto play mode. In auto play mode, users can choose a song they want to play by typing the name of song in terminal (this guidance will appear on screen when users choose auto play mode), and the song will play automatically just as an mp3. Except for songs we put in there already, users can also create their own songs in this mode by typing the numbered musical notation into a specific file, our project support this function.

As for sound, we use Karplus-Strong algorithm to generate the sound of piano keys, and this algorithm is implemented in FPGA.

We got the basic idea of our project from a project made by Cornell student, but we did many innovations and improvements in our project. The description of this Cornell project is listed in reference part.

The implementation of image processing, denoise, sound and user interface are all done using FPGA. The implementation of image capture and preprocess are done in software.

HIGH LEVEL BLOCK DIAGRAM

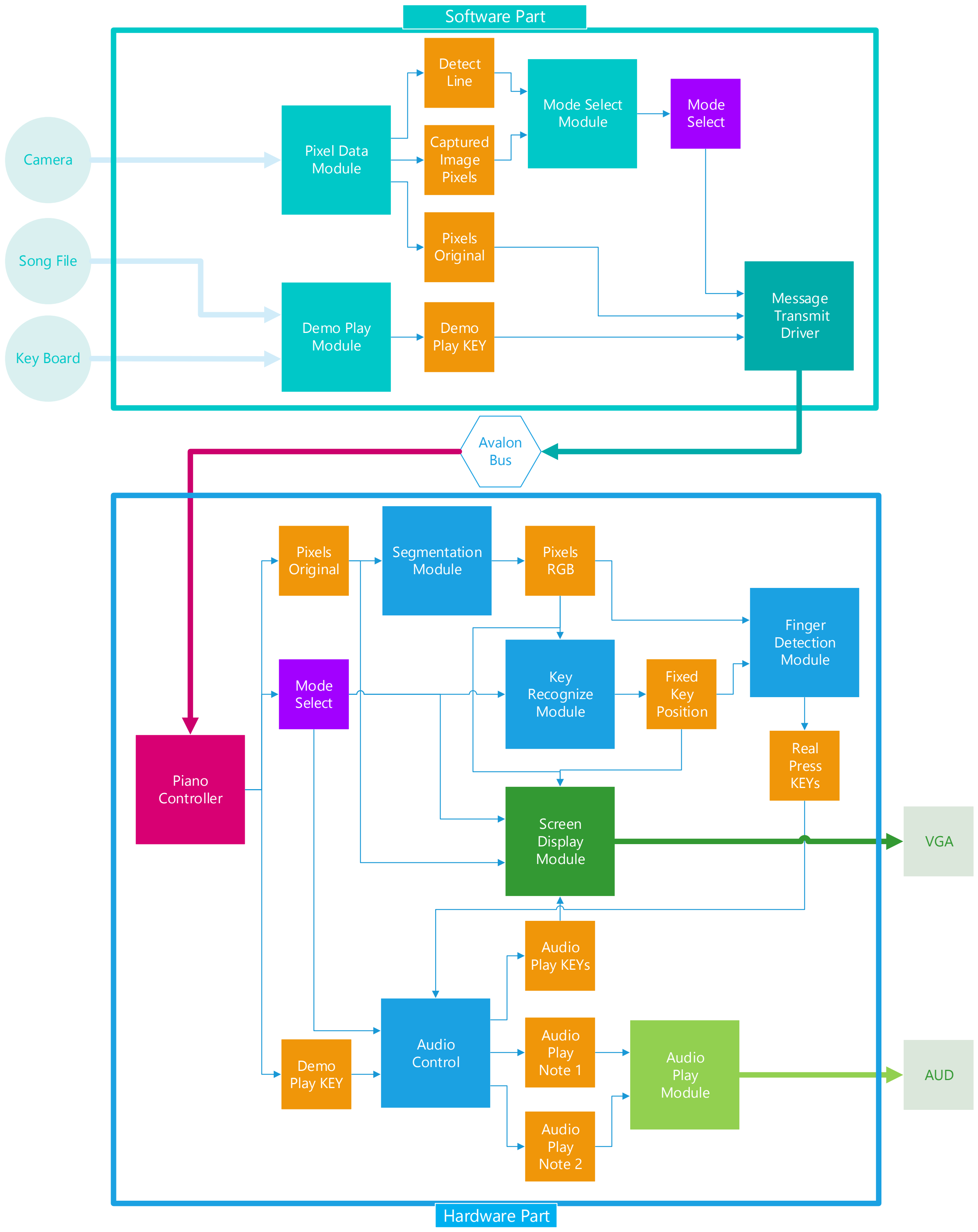


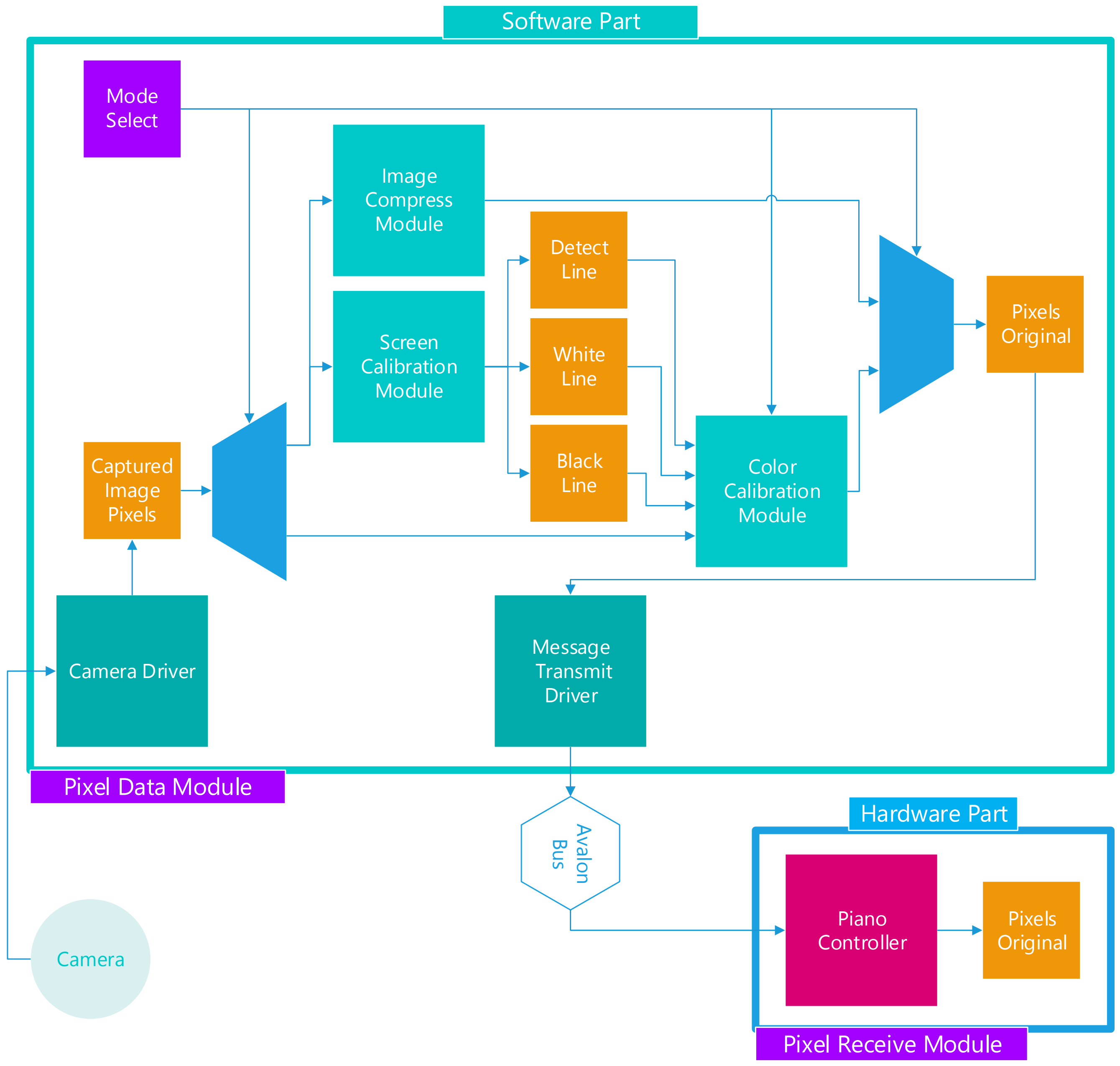
FIGURE 2: HIGH LEVEL BLOCK DIAGRAM

The above picture shows a high-level overview of our project. We have software part and hardware part in our system. Avalon bus is used to connect the software part and the hardware part. In software, we have pixel data module, demo play module, and module select module. In hardware, we have segmentation module, audio control module, key recognize module, and finger detection module. We will go into details about those main modules in the next section.

DETAILED IMPLEMENTATION

Detailed Software implementation

Pixel data module



In the image capture and preprocessing module, Avalon bus is used to connect the software part and hardware part.

In the software part, we write a camera driver to get the pixels RGB from the camera. Through the Avalon bus we send it to the piano controller which is implemented in hardware. Another main function of this module is the implementation of color calibration and help users to do the screen calibration.

Firstly, the software implementation of camera driver is shown below, we will discuss the color calibration and screen calibration later.

CAMERA DRIVER

The key part of our design is to use a Web-Camera to capture the image of the finger and the display monitor and recognize the position of the finger on the display monitor. We decide to split this part in to two main subparts. One is to get the real time captured image from the Web-Camera and send it to the memory in FPGA and the other one is to do some image processing program on this captured image to find the position of the finger. We decide to implement the first part in LINUX environment named "image capture and preprocessing" and implement the second part in FPGA environment named "finger detection". In this section, we will talk about the software implementation of the part "image capture and preprocessing". The high level block diagram of this part is showed below:

Introduction to V4L

Video4Linux (V4L) is the driver for video devices from LINUX kernel, it can provide a series of APIs for the program which intend to use video devices as input. If we want to use this driver, we have to make sure our Web-Camera can support UVC (USB Video Class). Fortunately, he Web-Camera we used can support UVC.

Under the LINUX environment and the function of V4L, all the peripherals are considered as special files named device file. The APIs the V4L diver provided can allow us to read and set this files to get the image data we want. This device file will be store at the folder "/dev/video".

Recompile the kernel and revise the dts tree to support V4L driver

Since the original Linux kernel provided by the board doesn't support the v4l driver, we have to recompile the kernel and use menuconfig command to enable the v4l driver support of the kernel. However, after this modified of the kernel, it cannot support the connection between the Linux system and FPGA part anymore, we have to revise the dts tree to enable this connection, the steps are showed here:

1. Have linux folder under /usr/src
2. Open the file under /linux/include/generated/utsreleases.h and revise to: 3.8.0-00111-g85cc90f
3. In source code folder of the kernel, find the file socfpga_cyclon5.dts under /linux/arch/arm/boot/dts and add this to the end of the file:

```
lightweight_bridge: bridge@0xff200000
{           #address-cells = <1>;
            #size-cells = <1>;
            ranges = < 0x0 0xff200000 0x200000 >;
            compatible = "simple-bus";
            vga_led: vga_led@0 {        compatible = "altr,vga_led";
                                        reg = <0x0 0x8>;
                                };
   };
```

4. Use this command to generat the dtb file: /linux/scripts/dtc/dtc -O dtb -o socfpga_cyclone5.dtb socfpga_cyclone5.dts
5. Replace socfpga.dtb to socfpga_cyclone5.dtb in sockit folder.

After all of above steps, the Linux OS in board can both support our camera driver and insmod vga_led.ko.

Preprocessing the captured image

The input of this part is a 3D array of the captured frame image which includes the RGB parameter for each pixel. In order to make it convenience for "finger detection" in hardware and decrease the memory cost, we want to use software part to do some preprocessing of the frame image. The block diagram for this algorithm is showed below:

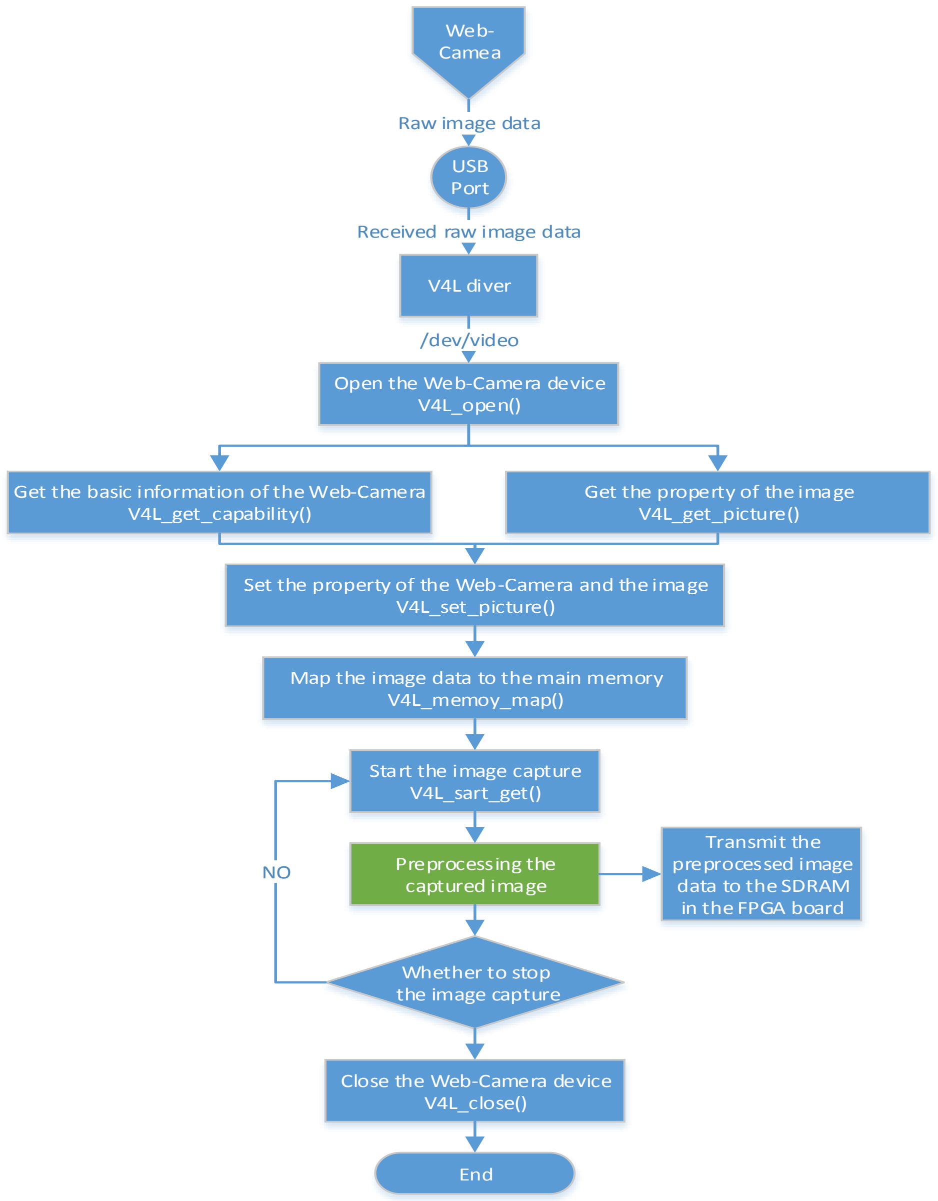

Web-Camea
Raw image data
USB Port
Received raw image data
V4L diver
/dev/video
Open the Web-Camera device V4L_open()
Get the basic information of the Web-Camera V4L_get_capability()
Get the property of the image V4L_get_picture()
Set the property of the Web-Camera and the image V4L_set_picture()
Map the image data to the main memory V4L_memoy_map()
Start the image capture V4L_sart_get()
Preprocessing the captured image
Transmit the preprocessed image data to the SDRAM in the FPGA board
NO
Whether to stop the image capture
Close the Web-Camera device V4L_close()
End

FIGURE 3: HIGH LEVEL BLOCK DIAGRAM FOR SOFTWARE IMPLEMENTATION OF "IMAGE CAPTURE AND PREPROCESSING" PART

Set the frame image property and map it to the main memory

- Use V4L_open() to open the device file, set the path of the device file to "/dev/video".

- Use V4L_get_capability() and ioctl() to get the information of the Web-Camera, and store those information to the struct video_capability.
- Use V4L_get_picture() and ioctl() to get the property of the image, and store those property to the struct video_picture.
- Use the V4L_set_picture() and ioctl() to set the property of the image in order to get the appropriate resolution and so on.
- Use V4L_get_mbuf() and ioctl() to get the information of frame image from the buffer of Web-Camera, and store information to the struct video_mbuf.
- Use V4L_set_mmap() to set the property of the frame image we want to get from the device file.
- Use V4L_memoy_map() and ioctl() to map the device file to the main memory.

Capture the frame image data

Use V4L_sart_get() and ioctl() to begin mapping the frame image to the main memory. After get the frame image, convert this image to the 3-D array to present the RGB parameter of each pixel. The block diagram is showed below.

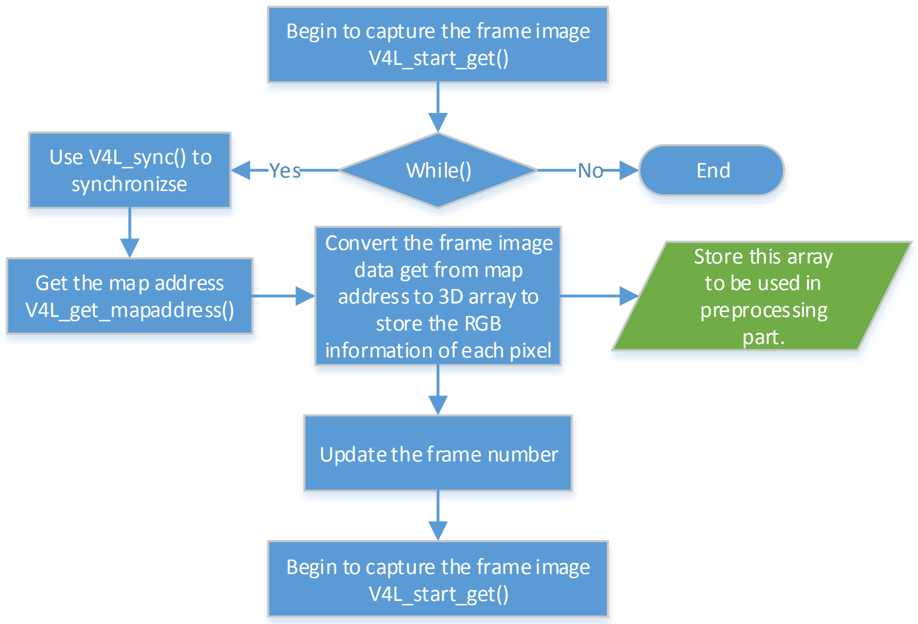
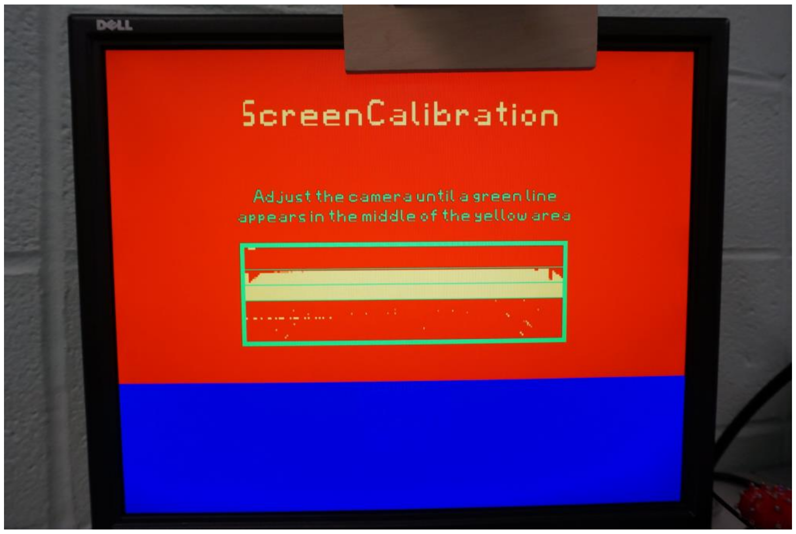

Begin to capture the frame image V4L_start_get()
Use V4L_sync() to synchronizse
Yes
While()
No
End
Get the map address V4L_get_mapaddress()
Convert the frame image data get from map address to 3D array to store the RGB information of each pixel
Store this array to be used in preprocessing part.
Update the frame number
Begin to capture the frame image V4L_start_get()

FIGURE 4: BLOCK DIAGRAM FOR CAPTURING THE FRAME IMAGE

SCREEN CALIBRATION

In order to make the function works, we have to make sure the camera can capture the area for touch detection. Thus, we provide user a convenient way to calibrate the screen position.

- Segment the captured image and display the touch area can be captured in screen.
- Give the visual feedback for the successful calibration.
- When the user calibrate the screen position successfully, the "Next" button will appeared.

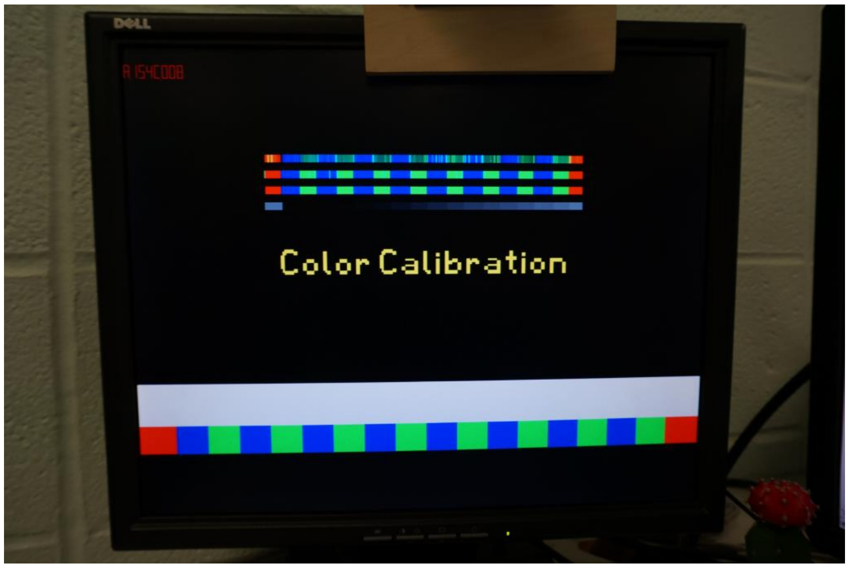

ScreenCalibration
Adjust the camera until a green line appears in the middle of the yellow area

COLOR CALIBRATION

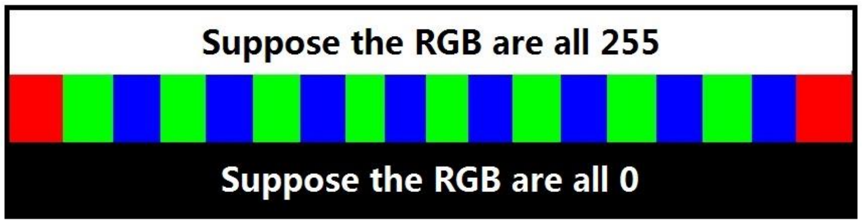
Due to the particular angle between camera and screen, the plane of the screen will reflect some light from the environment to the camera which will influence the color segmentation of the captured image as below picture showed. Therefore, an Algorithm for color Calibration is needed.

Our solution is that:

1. In color calibration mode, the screen will display a white area and a black area just above and below the key recognize area as figure showed:

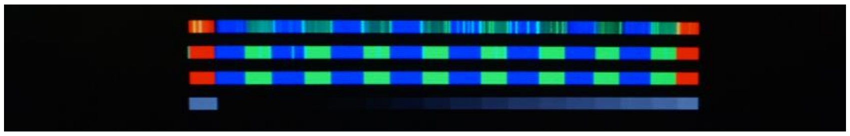


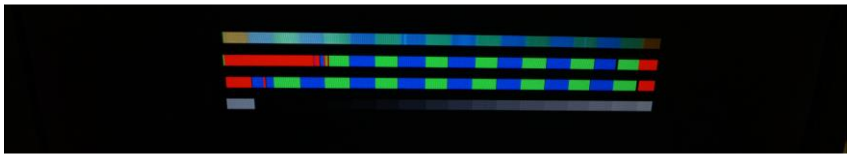
2. In this mode, the camera will recode a pixel line in white area and a pixel line in black area. Since the color for this two area are white which should has RGB value all the 255 and black which should has RGB value all the 0, we just suppose the value for two pixel line should be 255 and 0.



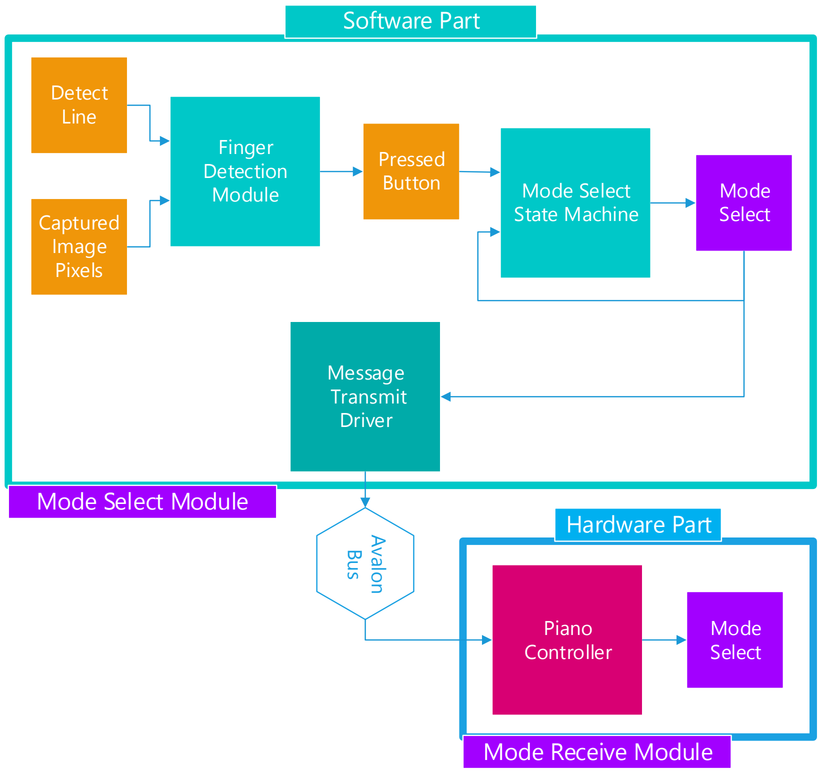
3. Due to the light reflect, the real pixel RGB in white and black area cannot be 255 and 0, therefore, the difference between the real RGB and 255 or 0 can tell us the information about environment light.
4. By using this light information, we can calibrate the color of capture pixel in key recognize line or finger detect line.

There is a comparison between the pixel get by implementing this color calibration and the original pixel, you can find the first picture which shows the pixel get by implementing the color calibration can segment the color successful in strong light environment. However, the below picture which shows original pixel, some of the blue and green pixel are segmented as red color.



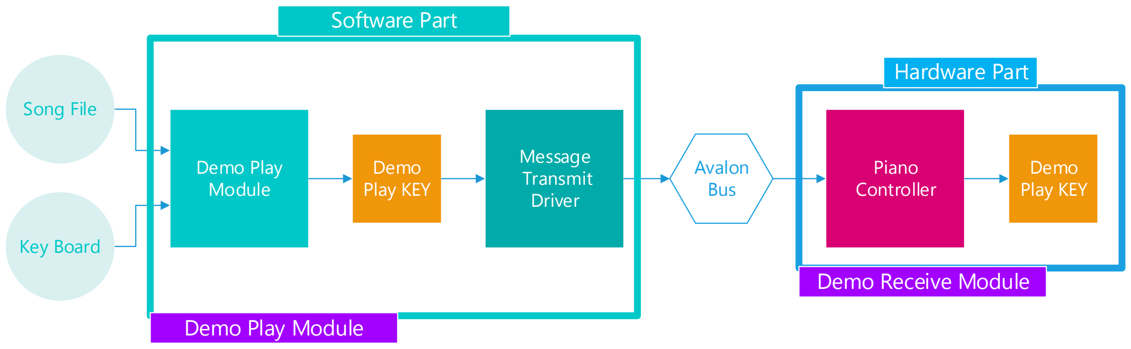


Mode select module



In this module, we implement the control logic of our program which means the implementation of the control logic of user interface. In this module, we run a simple finger detection algorithm in software part which has lower speed and accuracy than hardware implementation but is enough for detection 5 button press action in touch area. Our control logic can be saw as a state machine, the transfer of the state is achieved by user's button press action. The tag for each state will be recode as Mode Select signal and will be sent to FPGA as control signal for FPGA part.

Demo play module

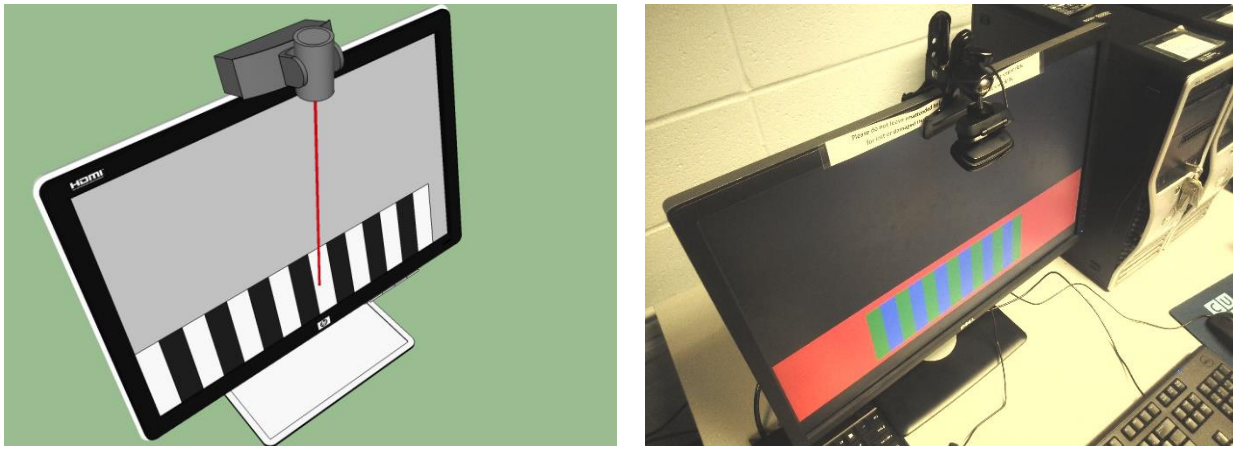


In this module, we implement an application of our system which is the piano player simulation. In software part, user can set a song file as input file to the program which can be define by user or provide by us. Then if the mode select signal transfer to demo play mode, the program will use the numbered musical notation recorded in song file to play a song. Therefore, the function of this module is to provide the interface for user to select the song file and transmit those numbered musical notation to the FPGA part.

Device we used

As the design sketched in Figure 5(a) showed, in order to implement a virtual touch display screen, we have to use a camera to capture the image of finger and display monitor in a special angle. To approach this design, in real hardware implementation for capturing image, there are three main parts which: a web-camera, a camera clip and a normal display monitor.

In our implementation, we fix the web-camera with the camera clip and fix this camera clip at the top of the display monitor. Since the camera clip we used can be set to almost any angle, we will set it to an appropriate angle to make sure the web-camera can capture the entire keyboard area in the display monitor. The whole hardware implementation is showed in Figure 5(b).



(a) Desin sketch. (b) Real implemenation.

FIGURE 5: THE WHOLE HARDWARE IMPLEMENT FOR IMAGE CAPTURE.

Details about each component are discussed below.

WEB-CAMERA

FIGURE 6: LOGITECH C615 WEB-CAMERA

Since there are several USB ports in the SocKit board, we decide to use a USB Web-camera as the camera to capture the image of finger and display monitor. In order to make this real-time reorganization system sensitive enough to detect the position of the finger, we hope this Web-Camera have higher resolution and fps. Therefore, finally, we decide to use Logitech C615 Web-Camera and the specifications of the Web-Camera are list below:

- Full HD 1080p video capture (up to 1920 x 1080 pixels) with recommended system
- Autofocus
- Hi-Speed USB 2.0 certified (recommended)
- Universal clip fits laptops, LCD or CRT monitors

CAMERA CLIP

For the camera clip, it should satisfy the follow requirements:

1. Can be fixed at any kinds of display monitor.
2. Can hold the Web-Camera stable.
3. Can be set to any angle.

After seeing many different kinds of camera clips, we found the camera clip in Figure 7 can satisfy those requirements.

1. The clip part of this camera clip can fix at almost any kind of display monitor.
2. By using screw to combine with Web-Camera, it can make sure the stability of the Web-Camera.
3. The ball head part of this clip make it possible to be set to the any angle.

Therefore, in our design, we decide to use this camera clip.

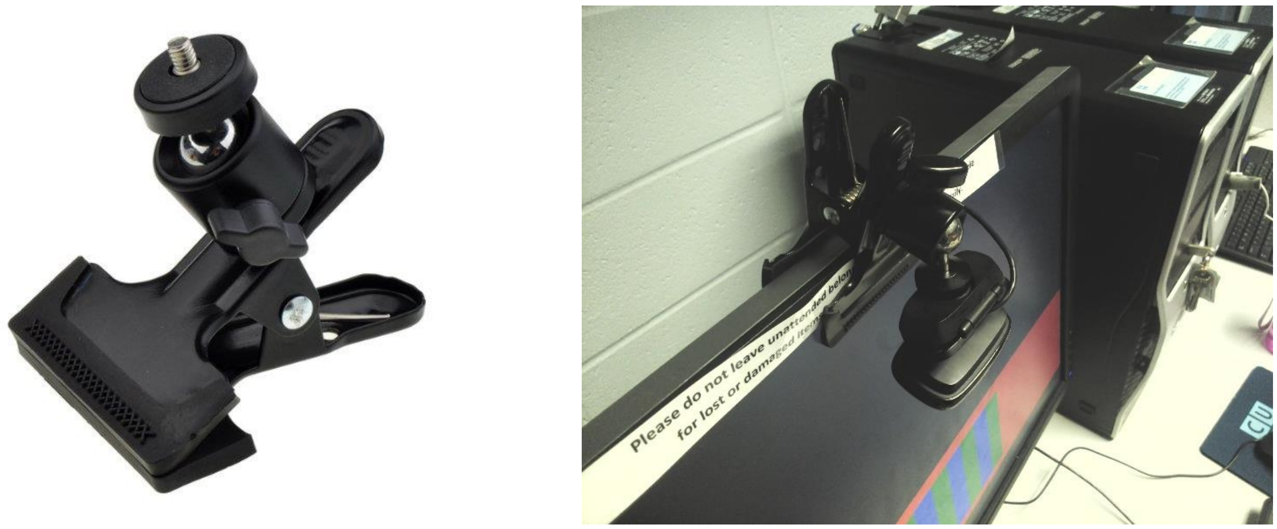


FIGURE 7: NEEWER® MULTI-FUNCTION SPRING CLAMP CLIP HOLDER MOUNT WITH BALL HEAD W/ STANDARD 1/4" SCREW FOR SLR, DIGITAL SLR, VIDEO CAMERAS

Detailed Hardware implementation

Audio play module

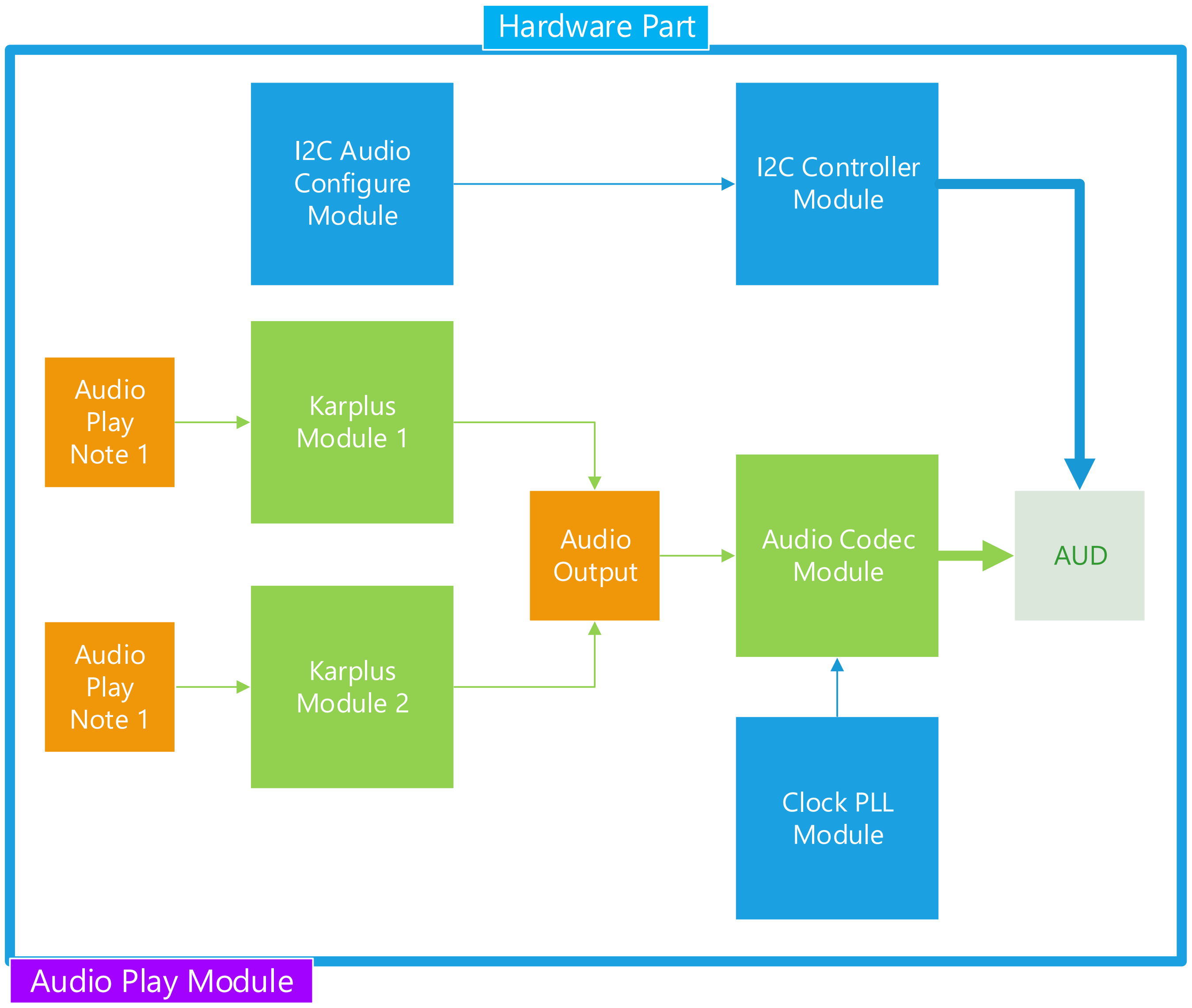


Figure shows the block diagram of the audio part.

For the audio part of our project, we use Karplus-Strong string synthesis algorithm to simulate the sound of the key of a piano. We use the Verilog program to implement the algorithm to create the waveform. And the CODEC part on the FPGA is used to play the sound out.

When a key is pressed, the finger is detected by the camera and after the image processing, it will call the corresponding sound samples and the piano sound will be played via speaker. Moreover, we implement the function that when two keys are pressed together, we can play the mixing sound of the two keys.

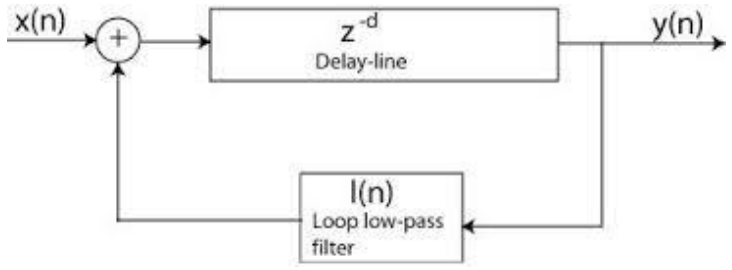
KARPLUS-STRONG ALGORITHM

The Karplus-Strong string synthesis Algorithm is widely used to simulate the real instrument sound produced by strings. It is a method of physical modeling synthesis that makes use of looping a short wave signal through a filtered delay processing to simulate the string sound.

There are four basic steps for this algorithm (wikipedia):

- A short excitation waveform (of length L samples) is generated. In the original algorithm, this was a burst of white noise.
- This excitation is output and simultaneously fed back into a delay line L samples long.
- The output of the delay line is fed through a filter. The gain of the filter must be less than 1 at all frequencies, to maintain a stable positive feedback loop. The filter can be a first-order lowpass filter (as pictured). In the original algorithm, the filter consisted of averaging two adjacent samples, a particularly simple filter that can be implemented without a multiplier, requiring only shift and add operations. The filter characteristics are crucial in determining the harmonic structure of the decaying tone.
- The filtered output is simultaneously mixed back into the output and fed back into the delay line.

The following figure shows the block diagram of how Karplus- Strong algorithm works.



SOUND

Firstly, we make use of the Karplus-Strong Algorithm to synthesize the sound of the piano hammer strike. This is accomplished by low pass filtering of the input signal. We make use of a first-order low pass filter, and set the gain as 0.125. And we also improve the sound by simulating three strings per note, and we call them lower string, middle string and higher string, which is considered from a real piano.

In the three strings for each note, the length of middle string is obtained using previous expression. However, for lower string and higher string, we set the length of the shift register to be a little different, typically the length of middle string plus or minus two. Finally, the output is formed by adding these strings together and right shift by 2 bits, which equals to divide by 4. Hence, we have three rams for each note and the output is pretty good. We use some tune test applications on mobile device to test and adjust the frequency of each note from the speaker, and finally get 16 notes correspond to the note frequency tale.

Also, we add another Karplus- Strong module to support the mixed sound of pressing two keys at the same time. If two keys are pressed together, the output of those two modules will be added and then divided by 2 to generate the mixing sound. This basic idea is the same for mixing three or more sound together.

DETAILS ABOUT AUDIO CODEC ON FPGA

(Reference: http://zhehaomao.com/blog/fpga/2014/01/15/sockit-8.html)

The SoCKit board uses the Analog Devices SSM2603 audio codec

I2C controller

Before the Audio Codec can start capturing or playing audio, it has to be configured with options such as the sampling rate and sample width (how many bits each sample is). The protocol the audio codec uses for configuration is the Inter-Integrated Circuit (I2C) protocol.

I2C is a master-slave protocol. In our case, the FPGA is the master and the audio codec is the slave. To initiate a transmission, the master keeps the SCLK line high and then does a high-to-low transition on the SDAT line. The start symbol is followed by transmission of the 7-bit slave address.

After the seven bits of the address are transmitted, an eight read/write bit is transmitted. This should be 0 if the transmission is a write and 1 if it's a read.

After the address and read/write bit are sent, the master allows the SDAT line to float for one SCLK cycle. The slave whose address has just been transmitted should send an acknowledgement (ACK) by pulling SDAT low for the entire cycle.

After receiving the ACK, the master transmits the data a bit at a time in the same way it sent the address and read/write bit. After every eight bits, the master should wait one cycle for an ACK from the slave.

After the last byte is acknowledged by the slave, the master sends a STOP symbol to end the transmission by sending a low-to-high transition on SDAT while SCLK is held high.

I2C configuration

The audio codec organizes its configuration variables into 19 9-bit registers. The first seven bits of the data transmission are the register address, and the last nine bits are the register contents.

Register 0 controls the input volume of the left channel ADC. Registers 2 and 3 control left-channel and right-channel DAC volume, respectively. Register 4 contains options for the analog audio path. Register 5 controls the digital audio path. Register 7 sets the options for the digital audio interface, which we will use to send and receive samples from the codec. Register 8 controls the sampling rate. We want a 44.1 kHz rate for both the ADC and the DAC. Finally, we set register 6 to all zeros to power on the output and bit 0 of register 9 to 1 to activate the digital core.

Audio codec

Audio codec is the main driver for putting all things together and processing the audio part. The clock passed in here is the audio clock from the PLL, which is then divided down for LRC and BCLK. The data is pushed out or read in through shift registers. There are also sample_end and sample_req signals, which are asserted high for one cycle after the last bit of an ADC sample is received or before the first bit of a DAC sample must be sent.

Karplus-strong note

This part is basically the main module for implementing Karplus-Strong algorithm.

The algorithm first generates a series of random white noise signal as input x (n), which is then feed into a shift register, which can generate a delay in input signals and act as delay line in the above block

diagram, and then the output of the shift register is processed by a low pass filter. And after the signal passing through the low-pass filter, we feed back the signal to the input of the shift register. In this process, the signal will decay gradually.
The note will sound different with various gains and shift registers. And the gain value of the filter can also change the sound mode. We set two kinds of different filters. Using the function: 'assign alpha = (control[3] == 0) ? 3'b100 : 3'b001'.

The length of the shift registers determines the pitch of the note. The sample rate is 44.1KHZ, the relationship between the frequency, the length of shift register and the sample rate can be described in the following expression:

$$freq = 44.1KHz(Sample\ Rate)/length$$

Since the frequency and pitch for a specific note is fixed, we set the length as below,

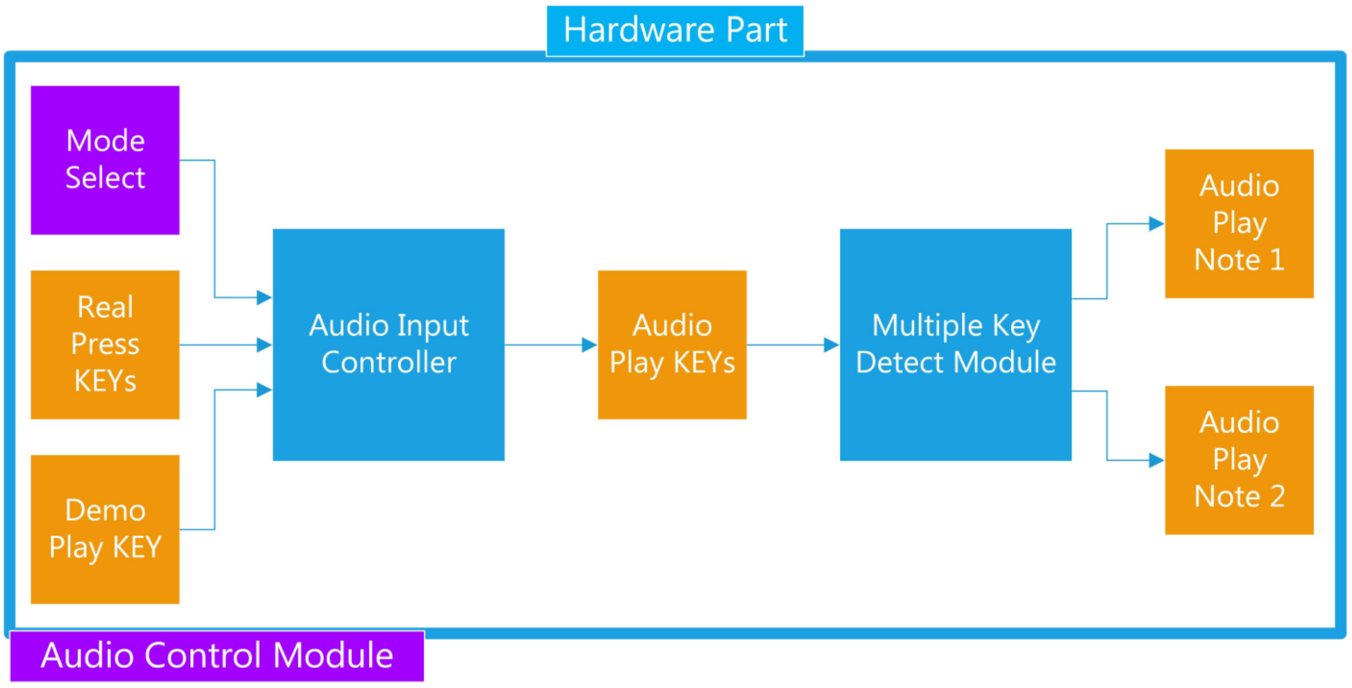
```
if (combination==17'b10000000000000000)//587-d2,16
  note <= 11'd79;
else if (combination==17'b01000000000000000)//523-c2,15
  note <= 11'd89;
else if (combination==17'b00100000000000000)//493-b1,14
  note <= 11'd94;
else if (combination==17'b00010000000000000)//440-a1,13
  note <= 11'd106;
else if (combination==17'b00001000000000000)//391-g1,12
  note <= 11'd119;
else if (combination==17'b00000100000000000)//349-f1,11
  note <= 11'd133;
else if (combination==17'b00000010000000000)//329-e1,10
  note <= 11'd142;
else if (combination==17'b00000001000000000)//293-d1,9
  note <= 11'd160;
else if (combination==17'b00000000100000000)//261-c1,8
  note <= 11'd179;
else if (combination==17'b00000000010000000)//246-b0,7
  note <= 11'd188;
else if (combination==17'b00000000001000000)//220-a0,6
  note <= 11'd210;
else if (combination==17'b00000000000100000)//196-g0,5
  note <= 11'd238;
else if (combination==17'b00000000000010000)//174-f0,4
  note <= 11'd261;
else if (combination==17'b00000000000001000)//164-e0,3
        note<= 11'd280;
else if (combination==17'b00000000000000100)//146-d0,2
  note <= 11'd315;
else if (combination==17'b00000000000000010)//130.8-c0,1
  note<= 11'd350;
```

Clock pll

The first clock we have to worry about generating is the master clock MCLK (denoted as AUD_XCK), and the frequency we choose for this clock equals to 11.2896 MHz. Since the reference clock is 50MHz, this clock cannot simply obtained by dividing reference clock by an integer factor. Hence, we need to use a pll to generate this clock.

The next clock we have to consider is the bit clock, BCLK (denoted as AUD_BCLK). This clock should be a quarter the frequency of the master clock. We can easily generate this using a frequency divider on the audio clock from the PLL.

Audio control module



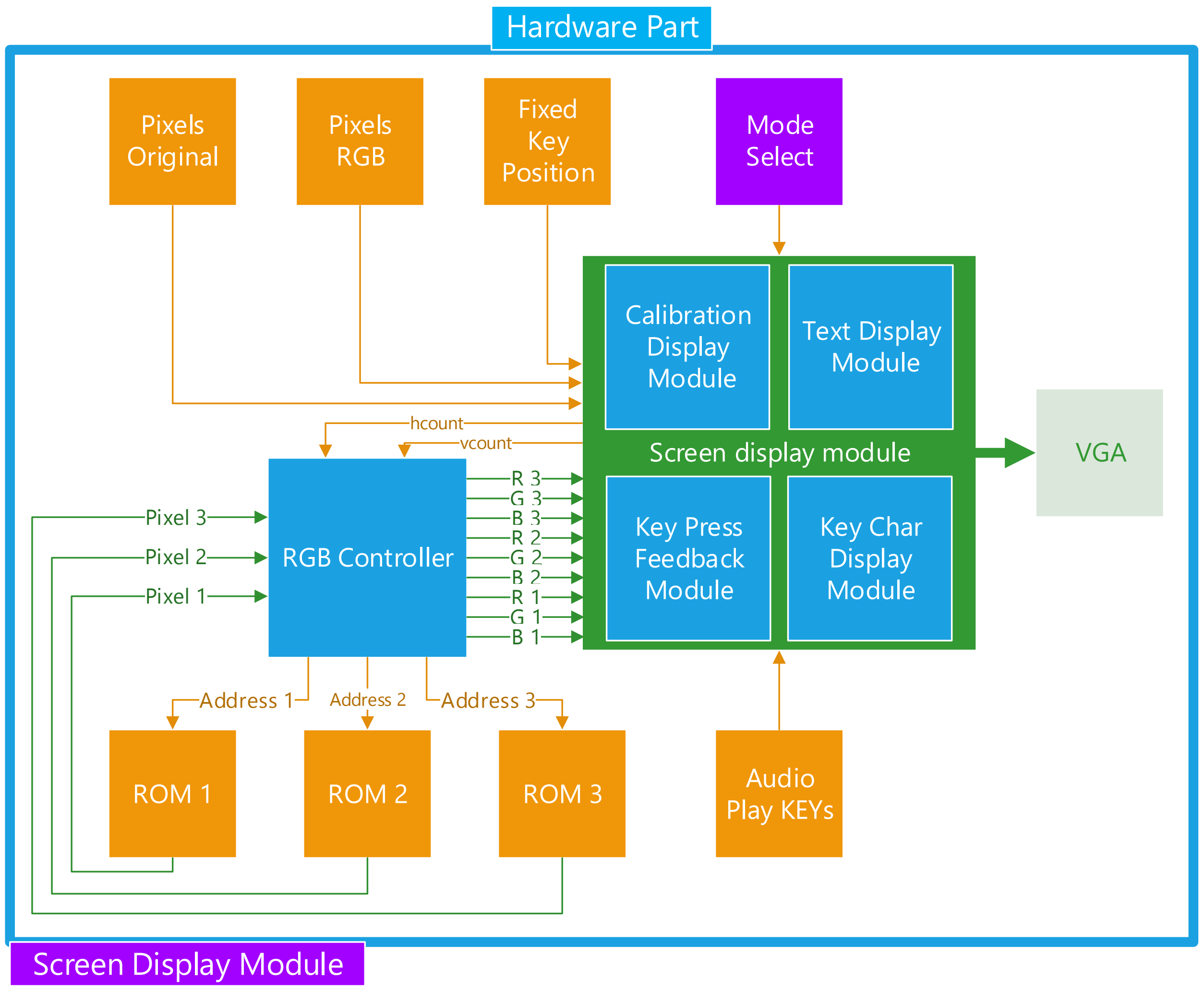
In the audio control module, we use a controller to decide what key source should we use. In free mode, we play the piano arbitrarily. In demo mode, we play the sample song. The key press information is fed into the sound mixing module, getting two key press information recording the pressed key position so that we can use karplus-strong algorithm to generate two sounds at the same time.

Sound Mixing module

In the finer detection module, we could detect every pressed key and then display the feedback on screen. But we could only play the sound of one note. For a real piano, we could generate sound as long as we press a key. So we would like to play the corresponding sound for each pressed key at the same time. Thus, we made a sound mixing module.

What finger detection ouputs is a one-dimensional array with 1s and 0s in it, indicating whether the corresponding key is pressed. We take this array as input and output two arrays, in which the first array have the position of first pressed key and the second array have the position of second pressed key. In the karplus-strong algorithm, we could only generate one sound. So we used two, each of which generated a sound. Then, we add the two sounds and divide the sum by two to get a mixed sound.

Screen display module



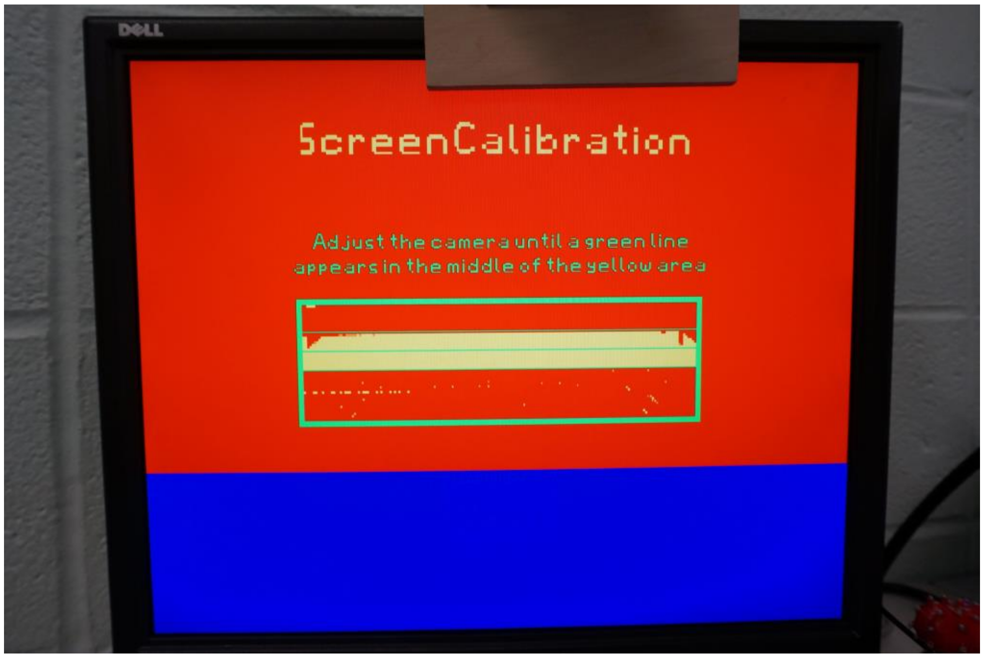
Our user interface module has several inputs, including pixels original, pixels RGB, fixed key position, three ROMs, mode select, audio play note. And two parts to get these inputs and process, including RGB controller and screen display module. And there are four parts in our screen display module. Calibration display module, text display module, key press feedback module, and the key char display module.

We have four pages for our user interface. Our images for user interface are shown as follows.

FIRST PAGE: SCREEN CALIBRATION

The method to draw the yellow block in the green frame is described as follows.

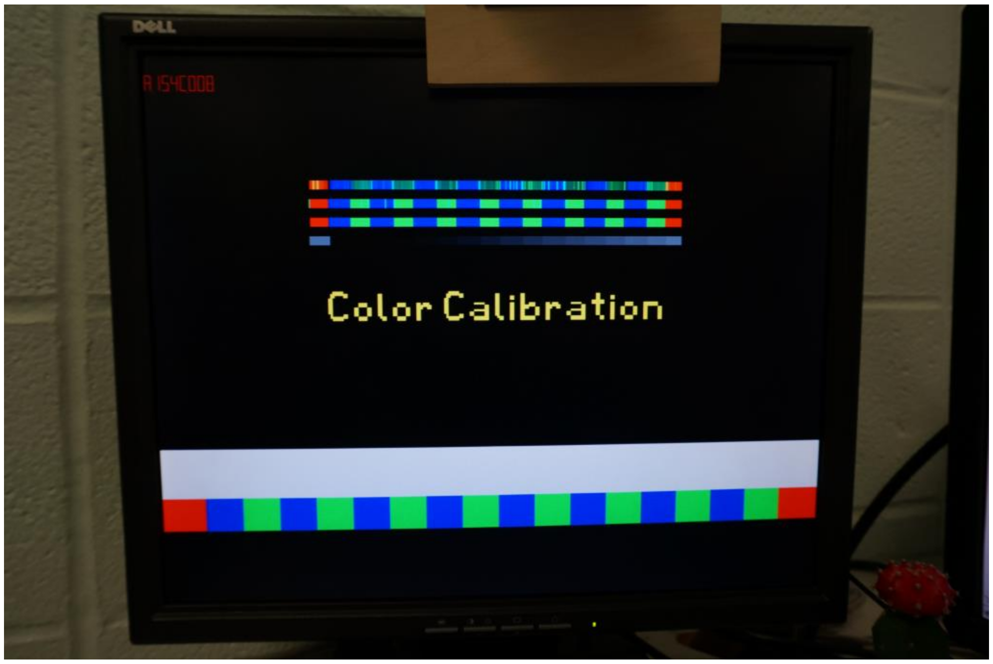
First, we get a long array which is the original pixel for the already processed image the camera get from the blue range. In fact, it contains several lines. The long array is restored to several lines using some methods. And they are displayed on the screen which 4*4 pixels represent one value.

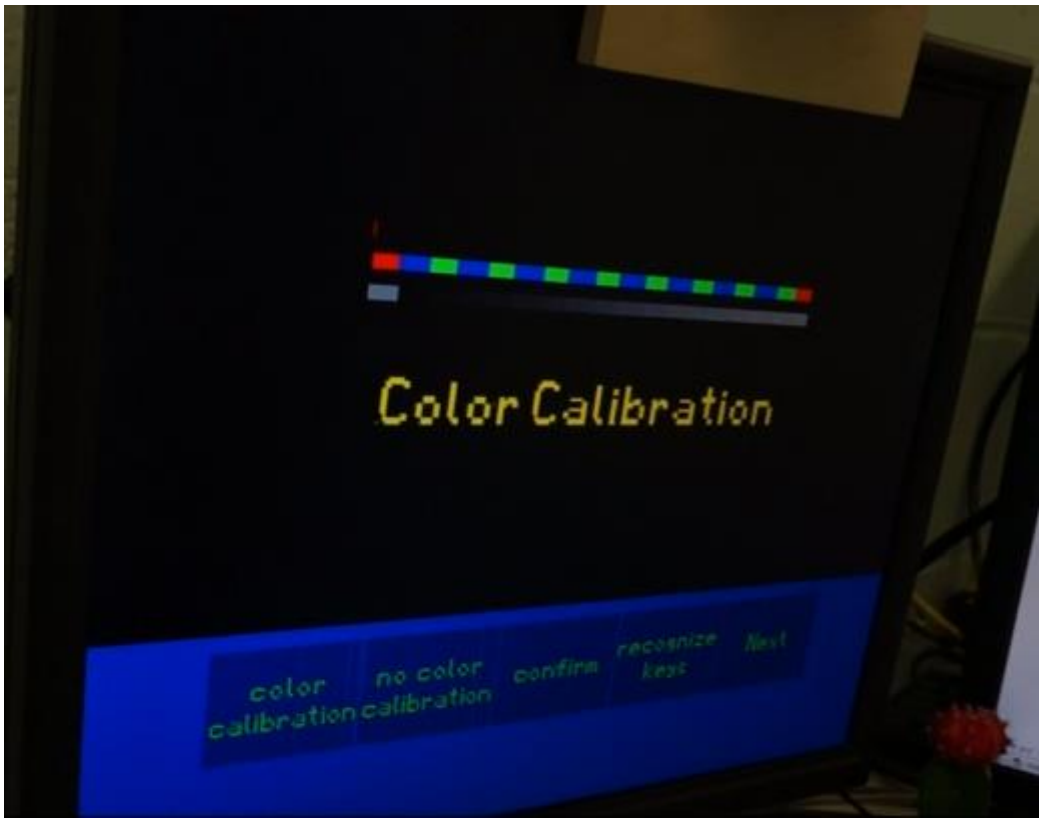

ScreenCalibration
Adjust the camera until a green line appears in the middle of the yellow area

SECOND PAGE: COLOR CALIBRATION

In this page we use 'HEX_CONVERT.sv' to convert the number or alphabet to the 7-segment display which is shown on the left corner of the screen. It is used to show some test results.

There are three narrow lines on the upper space. The first line is original pixels of the image get from the camera. The second line is pixel RGB with segmentation. The third line is gray value display to show the fixed key positions.





There are several words in this page. To implement this, we make the bitmap for the alphabets including capital and small letter. Then we make the words using these bitmaps for the alphabets putting together.

For example, the word ‘next’, is created like this,

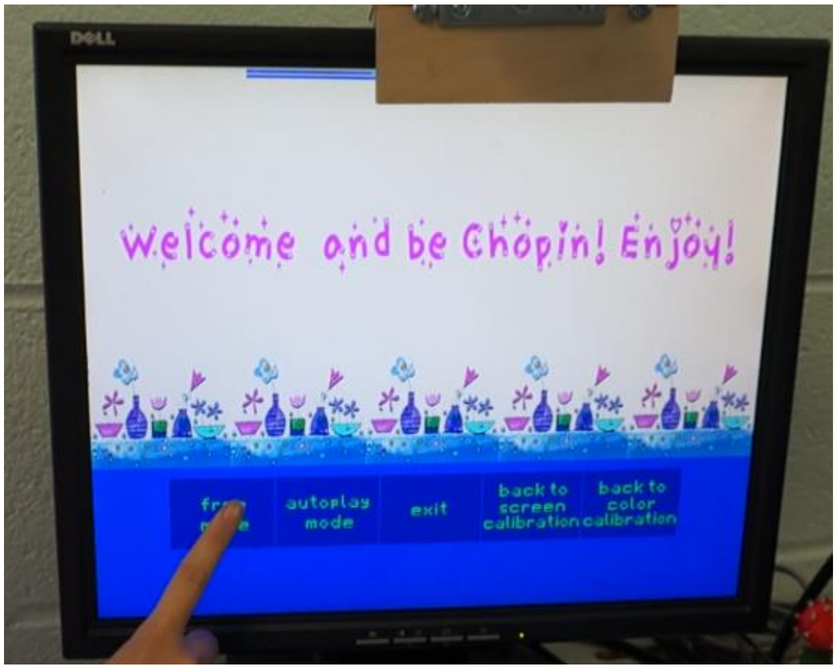
```
reg [43:0]  bit_next [0:13];
assign bit_next[0]   = 44'b00000000000000000000000000000000011000011110;
assign bit_next[1]   = 44'b00001100000000000000000000000000011000011110;
assign bit_next[2]   = 44'b00001100000000000000000000000000011000110110;
assign bit_next[3]   = 44'b00001100000000000000000000000000011000110110;
assign bit_next[4]   = 44'b00111111000000000000000000000000011000110110;
assign bit_next[5]   = 44'b00111111000110000110000011111000011001100110;
assign bit_next[6]   = 44'b00001100000110000110001111111110011001100110;
assign bit_next[7]   = 44'b00001100000011001100001100000110011001100110;
assign bit_next[8]   = 44'b00001100000011111100001111111110011011000110;
assign bit_next[9]   = 44'b00001100000000110000001111111110011011000110;
assign bit_next[10]  = 44'b00001100000011001100000000000110011011000110;
assign bit_next[11]  = 44'b00110000000111001110000000000110011110000110;
assign bit_next[12]  = 44'b00110000000110000110001111111100011110000110;
assign bit_next[13]  = 44'b00000000000000000000000000000000000000000000;
```

THIRD PAGE: WELCOME PAGE

The background of this page is stored in a ROM. We use a sprite of 340*128 pixels. And we duplicate it five times to make a whole background. We do this because of the limit of the storage of ROM.

Two ROMs are used here. ROM1 is used to save the image of the welcome page background, ROM2 is used to save the sentence ’Welcome and be Chopin! Enjoy!’ We use Matlab to change the image of ‘.png’ to ‘.mif’ to save it in the ROM.

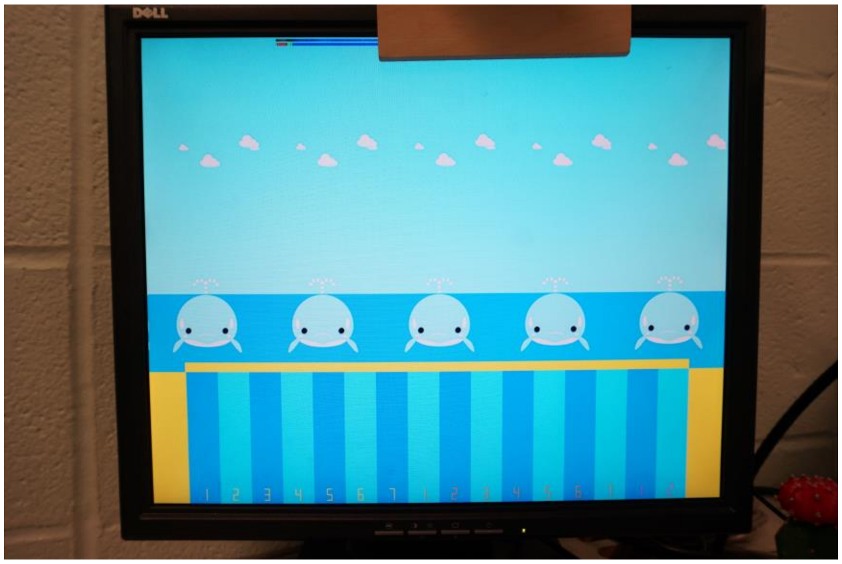
Screen display module gives out the values of hcount and vcount to the RGB controller. And the RGB controller calculates the address use the values to give out to the ROM. Then the ROM give back the according value of pixels to the RGB controller. Finally, the RGB controller processes the 24 bits value of pixels to three 8bits numbers which represent the value of R, G, B separately and gives it to the screen display module. Then the VGA can display the image which is stored in the ROM.


Welcome and be Chopin! Enjoy!
free mode
autoplay mode
exit
back to screen calibration
back to color calibration

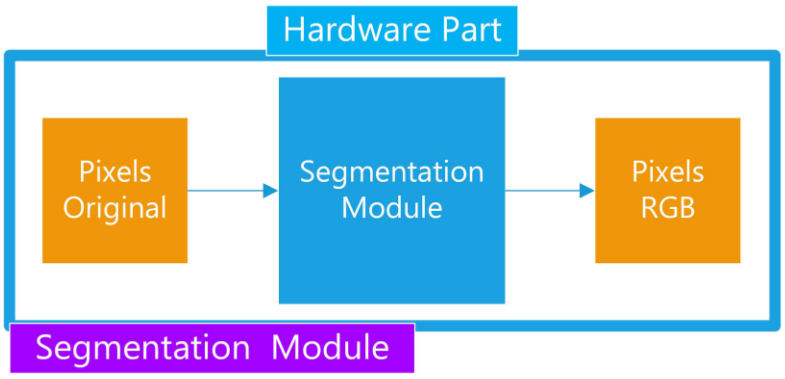
FOURTH PAGE: PLAY PAGE

In this page, ‘KeyCHAR_DISP.sv’ is used to convert the number or alphabet to the 7-segment display. It shows the notes number of keys on the bottom of the key.

When the user pressed the keys, the pressed keys give a feedback using changing color to a darker blue with a black out boundary.



Segmentation module



What software transmits to hardware is a line of pixels. Each pixels have their own value. In real world, the pixels captured by the camera could be in any color. In order to make the finger detection more robust, we want the pixels to have only 3 colors, namely red, green, and blue. So any other color will be recognized as red, green, or blue according its own RGB value. This algorithm is called segmentation.

Here, we segmented the pixel color by the following rule. Basically, each RGB value is between 0 and 255. If the value of G is larger than the value of R and B, the pixel will be recognized as green. If the value of B is larger than the value of R and G, the pixel will be recognized as blue. As for red, it is a little complicated since the finger's color is more close to red thus we must test it in practice to maintain robustness. Here, we will recognize the pixel as red when the value of R and G is larger than 130 and the value of R is larger than G and B.

The flow chart of segmentation is demonstrated in figure.

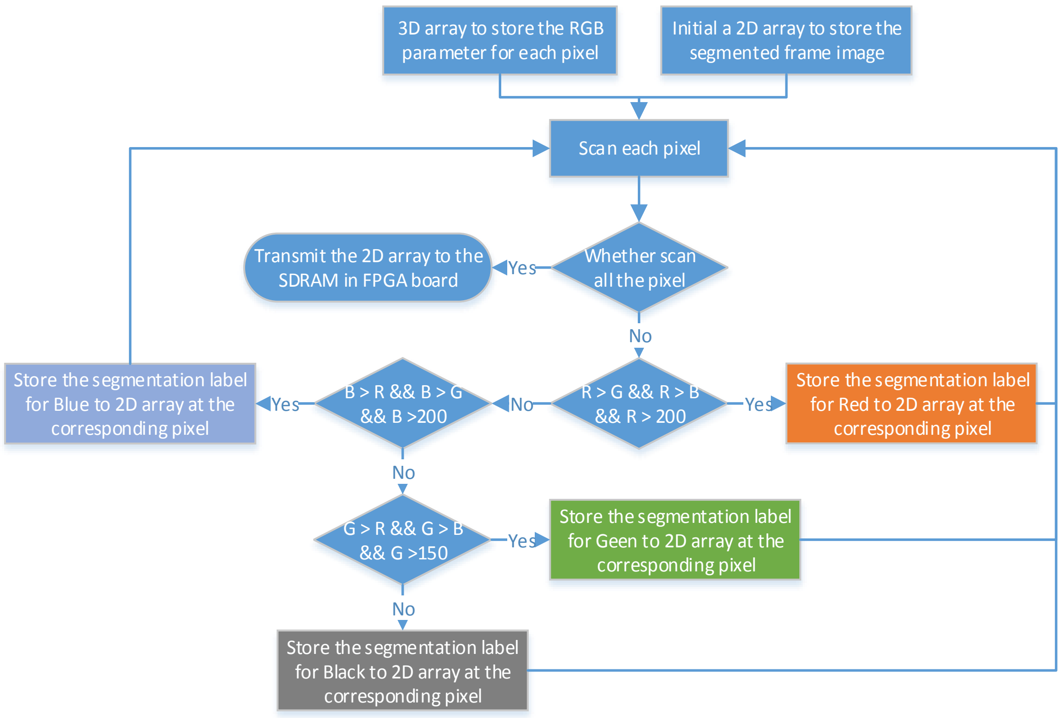

3D array to store the RGB parameter for each pixel
Initial a 2D array to store the segmented frame image
Scan each pixel
Whether scan all the pixel
Yes
Transmit the 2D array to the SDRAM in FPGA board
No
R > G && R > B && R > 200
Yes
Store the segmentation label for Red to 2D array at the corresponding pixel
No
B > R && B > G && B >200
Yes
Store the segmentation label for Blue to 2D array at the corresponding pixel
No
G > R && G > B && G >150
Yes
Store the segmentation label for Geen to 2D array at the corresponding pixel
No
Store the segmentation label for Black to 2D array at the corresponding pixel

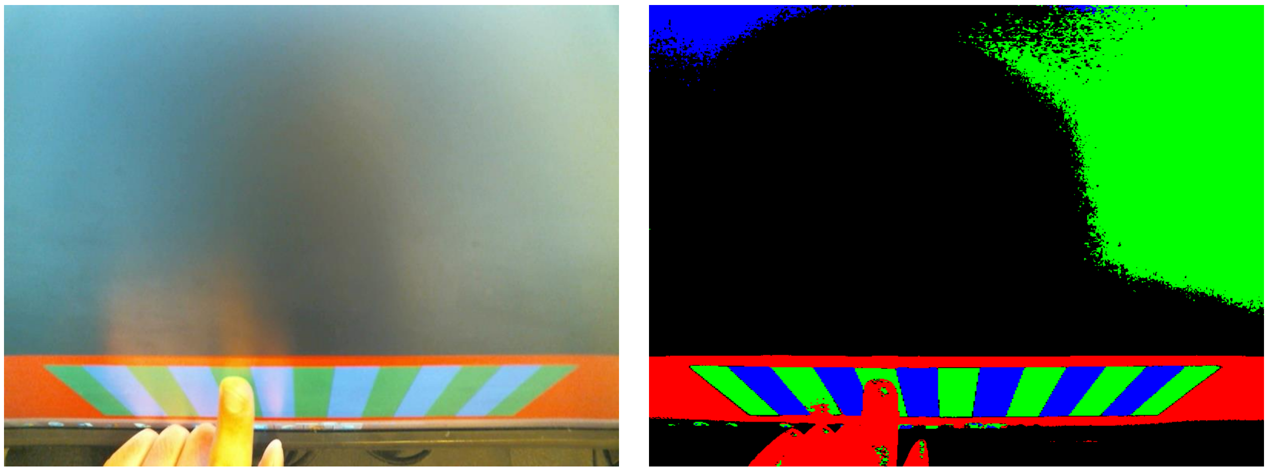
FIGURE 8: BLOCK DIAGRAM FOR THE ALGORITHM OF "PREPROCESSING THE CAPTURED IMAGE"

Firstly, we will use the RGB parameter for each pixel to segment the frame image into four kinds of colors, red, blue, green and black. The rule of this segmentation is presented in the table below.

TABLE 1: THE RULE OF THE SEGMRNTATION

Color	Condition	Segmentation label
Red	R > G && R > B && R > 200	1
Blue	B > R && B > G && B >200	2
Green	G > R && G > B && G >150	3
Black	Otherwise	0

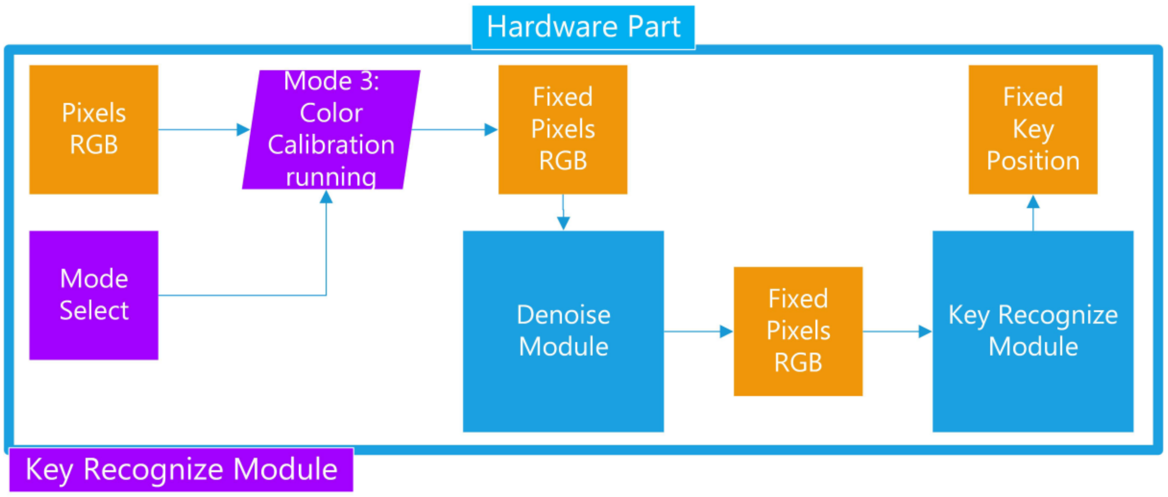
The Figure 9 show the result of the segmentation of a frame image. However, in order to compress the image data which will be stored in registers in FPGA, we will create a 2D array to store the segmentation label for each pixel and transmit this to the "finger detection" part in hardware implementation.



(a)The frame image captured by Web-Camera. (b)The frame image after segmentation.

FIGURE 9: SEGMENTATION OF THE FRAME IMAGE

Key recognition module



The software transmits a line of pixels with RGB value to hardware. In hardware, we finished the noise reduction, segmentation, key recognition and finally finger detection.

The flow chart of key recognition is demonstrated in figure . The original line of pixels with RGB value is transmitted from software. If we go through mode 3 (color calibration), we will get the calibrated pixels and store them in a line of fixed pixels with RGB value. It records the RGB value of each pixel in the key area. Then we go through the denoise module to get ride of uncertain variation. The key recognition module is used to get the key position (1, 2, …, 16) of each pixel in the target line.

DENOISE MODULE

The denoise module basically takes the array of fixed pixels as input and output an array of denoised fixed pixels. The algorithm is: For every pixel in the fixed pixels array, compare it with the pixels right before and after it. If it is different from both the pixel before it and the pixel after it, we consider it as a noise and assign the value of the previous pixel to it.

This approach will effectively reduce the uncertain jitter in the middle of the target pixel line.

KEY RECOGNITION MODULE

The key recognition module basically scans the fixed pixels with RGB value from left to right. Since our keyboard in the recognition mode is either blue or green, we could judge which key area the pixel belongs to by scanning the target pixel line from left to right. If the pixel is blue, it belongs to key one; Continue scanning and if the pixel is green, it belongs to key 2. Continue scanning like this and we will get the key position of each pixel.

Key Label module

The label module takes the image data and the 10-bit target horizontal line value as input. It outputs up to 32 10-bit signals telling the beginning and ending position of each label.

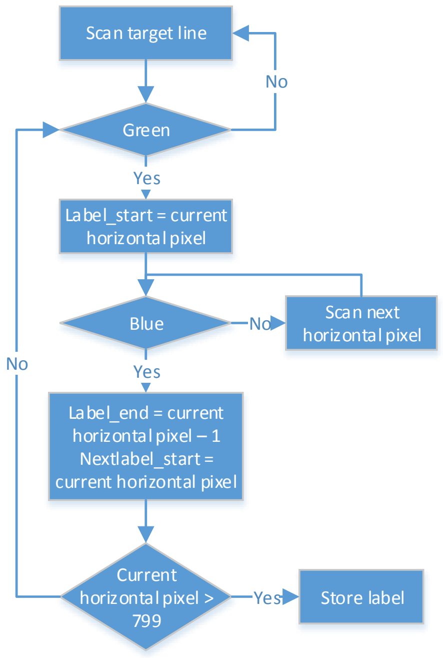

Scan target line
No
Green
Yes
Label_start = current horizontal pixel
Blue
No
Scan next horizontal pixel
Yes
No
Label_end = current horizontal pixel – 1
Nextlabel_start = current horizontal pixel
Current horizontal pixel > 799
Yes
Store label

FIGURE 10: LABEL MODULE

Keyboard detection module

The keyboard detection module takes the image array as input and outputs a 10-bit signal telling which horizontal line we are using to label and detect finger positions. The image array has 800 * 600 pixels and each pixel is implemented using 2 bits. 00 stands for black, 01 stands for red, 10 stands for green, and 11 stands for blue.

At the rising edge of clock, compare the color of each vertical pixel with red. If it is not red, the vertical counter increases by 1. If it is red, assign the target signal with the current vertical counter plus 30. We also need a FSM to control the flow of operation.

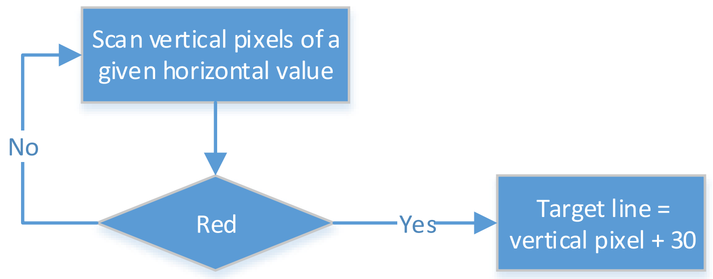

Scan vertical pixels of a given horizontal value
No
Red
Yes
Target line = vertical pixel + 30

FIGURE 11: KEYBOARD DETECTION MODULE

Diff module

The diff module takes the label information as inputs and outputs a 1-bit signal and a 4-bit signal telling the sound and display block whether there is a key being pressed and which key is pressed.

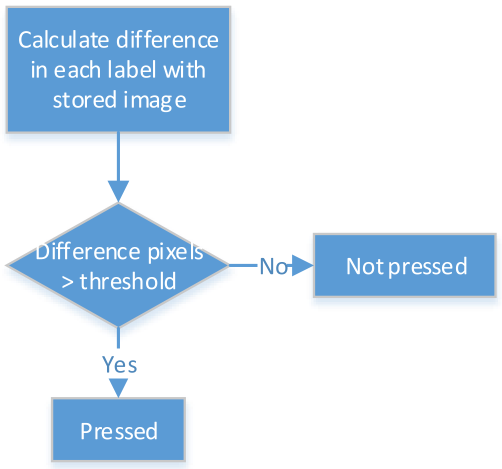
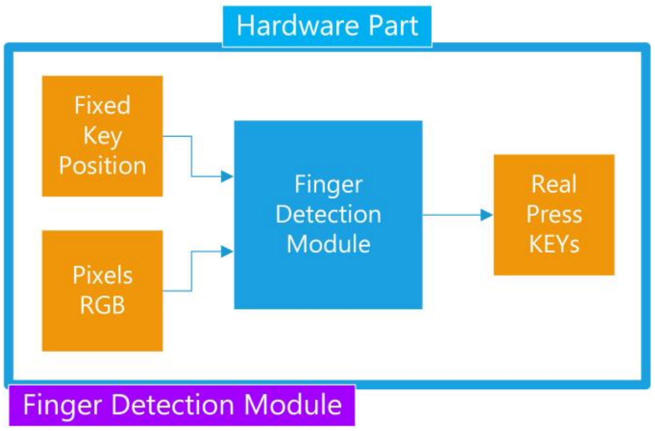

Calculate difference in each label with stored image
Difference pixels > threshold
No
Not pressed
Yes
Pressed

FIGURE 12: DIFF MODULE

Finger detection module



The key recognition module outputs an array of fixed key position of each pixel. Using this array, we could know which key area does each pixel belongs to. With the fixed key position and the real pixel RGB value, we could detect when the key is pressed and which key is pressed. The reason this will work relies on the fact that the color of finger after segmentation is red. In our play interface, the keys are blue with no red information. So wherever there is a couple of continuous red, we will consider it as a finger. The flow chart is demonstrated in figure 1.

The finger detection algorithm could be described as below: First, scan the real pixels from left to right. If it is red and is in key area, start to count the red pixel in this key area. If the number of red pixels is less than a given value, we will consider it as some kind of noise. If the number of red pixels is bigger than the given value, we will consider that this particular key is pressed. Then we output an array of press information indicating which key is presses.

The finger detection algorithm takes a 352 array as input and outputs two signals:

- 1-bit signal telling if there is a key being pressed
- 4-bit signal telling which key is pressed since we have 16 keys displayed on screen.

The image data we get from the registers only has four colors: black, red, green, and blue. The background is almost black during calibration. Red is used to separate the background and keyboard area. Green and blue are used as two alternating colors of keys. The actual image is shown in figure 10.

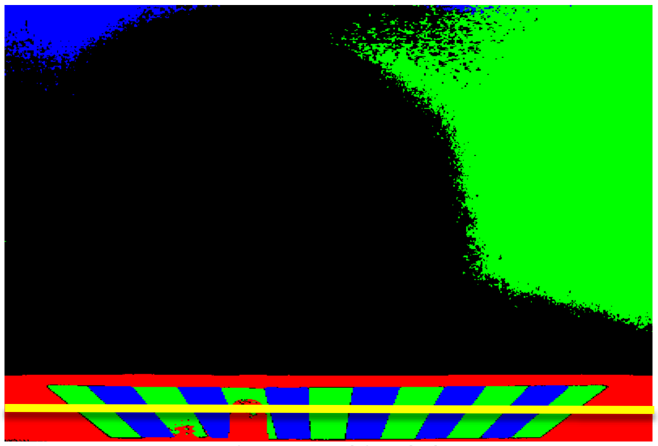


FIGURE 13: IMAGE DATA

The basic idea can be described below.

1. Scan the vertical pixels, if we find several consecutive red pixels (There may be noise pixels and a single red pixel cannot tell anything), we will know that the area below these red pixels are keyboard area. Then we assign the horizontal line, say 30 pixels below the last detected red pixel, as the target horizontal line. This target horizontal line is used to label different keys and detect if there is a key being pressed.
2. Scan the target line from left to right. If there are several pixels that are green, we will assign the position of the first detected green pixel to the label0_start signal. Continue scan the target line, if there are several blue pixels, we will assign the position of the first detected blue pixel to the label1_start signal. The label0_end will be assigned as label1_start – 1. After scanning the whole line, we compare the number of labels with the number of keyboards. If they are different, we must do it again. The label information is used as a reference when trying to detect if there is a finger. (Here we need to design a finite state machine to get rid of noise, since there may be some noise pixels in the middle of a key and will cause disaster. This is also for the purpose of robustness.)
3. Finally, we compare the difference in each label of our real time image with the original color (Green or blue). If the different pixels exceed the threshold value, we could say that the corresponding key is pressed.

   The detailed design is below.

The whole algorithm should be divided into 3 modules: keyboard detection, label, diff. Figure 11 shows a high level block diagram of the connection between these three parts.

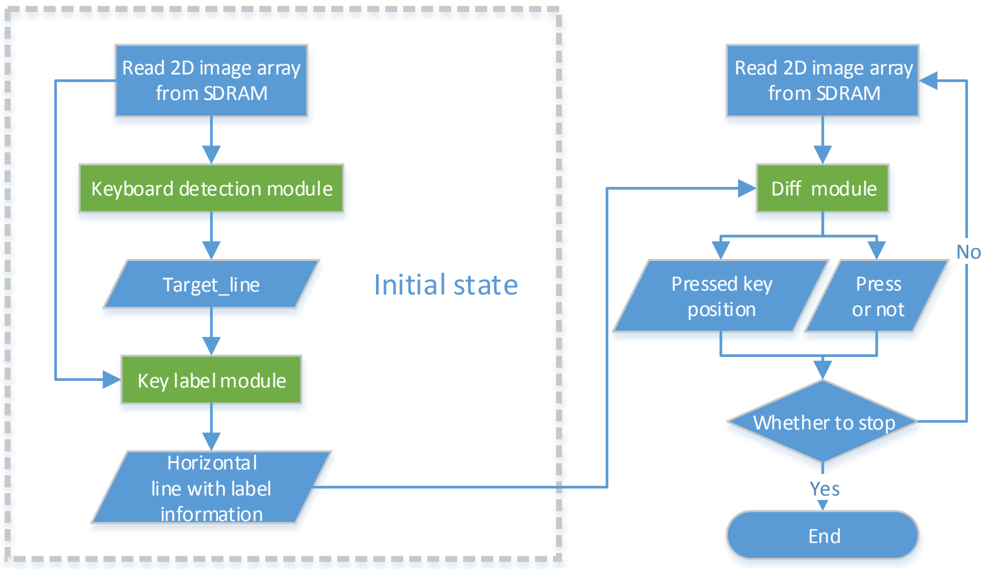

Read 2D image array from SDRAM
Keyboard detection module
Target_line
Initial state
Key label module
Horizontal line with label information
Read 2D image array from SDRAM
Diff module
Pressed key position
Press or not
No
Whether to stop
Yes
End

FIGURE 14: FINGER DETECTION BLOCK DIAGRAM

CRITICAL PATH ANALYSIS

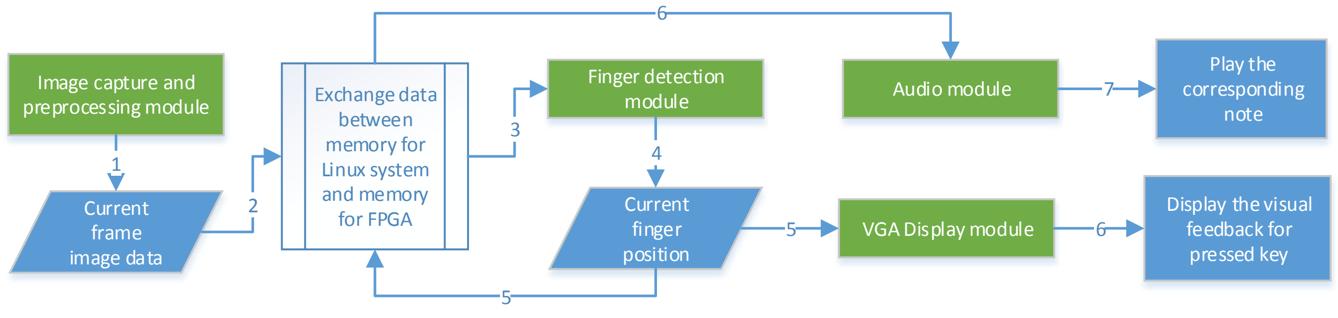


Image capture and preprocessing module
Current frame image data
Exchange data between memory for Linux system and memory for FPGA
Finger detection module
Current finger position
Audio module
VGA Display module
Play the corresponding note
Display the visual feedback for pressed key
1
2
3
4
5
5
6
6
7

FIGURE 15: THE BLOCK DIAGRAM FOR CRITICAL PATH

The response time for the whole system should be the time interval between the finger pressing a key and the audio feedback or visual feedback of the pressed key. As Figure 16 showed, the critical path should be the longer one between two paths. The one is from the beginning of capturing image to the visual feedback is displayed in monitor, and the other one is from the beginning of capturing image to the note of that pressed key is played.

RESULTS

1) Our design implement the basic function of a 16 keys piano. Users can play the key freely. Also, Users also can use the piano as a song player to play the song automatically. So it can be used widely in piano games and so on.
2) We implement the function of touching screen using camera on a normal screen. From the image got from the camera, through our hardware part, the piano keys and our fingers can be recognized correctly. And our calibration can adjust automatically according to the illumination changing. It has high accuracy, precision, and applicability.
3) We use Karplus-Strong algorithm to play the sound of a piano with one key pressed, or two keys pressed together which means the mix sound. And we implement it using FPGA. It is a very useful method to create songs, and can be used broadly.
4) We make a very humanistic human-computer interaction. Our user interface is very user friendly. We have five pages for user interface, the screen calibration, the color calibration, the welcome page, the play page, and the goodbye page. Every page has detailed description about what to do and how to do. Therefore, everyone can use it according to the manual shown on the screen without difficulty.

CONTRIBUTION AND TEAM WORK

Daran Cai

- Compile Linux kernel
- Camera driver
- Calibration (screen & color)
- Finger detection and key recognition algorithm
- Software control logic

Linjun Kuang & Wei Xia

- VGA display for keyboard and background
- RGB controller
- Matlab code for converting picture to mif file
- Drawing bit map and creating text using matlab
- Calibration and text display

Wenyuan Zhao

- Finger detection in hardware
- Karplus-strong algorithm
- Audio codec part and i2c configuration
- Mixing two sounds to support press two keys

MILESTONE

Milestone 1 (April 1st)

- Buy a camera and fix it on the top of the screen.
- Compile the linux kernel to make it support camera capturing image.
- Realize image capturing and preprocessing in Ubuntu operating system, develop algorithms for finger detection.
- Draw a simple keyboard using sprites.

Milestone 2 (April 15th)

- Complete image capturing and processing on the board, modify the algorithm and realizing finger detection.
- Finish all sprites for user interface, including keyboard, background, shadow feedback and blocks.
- Develop audio synthesis algorithm and implement it in hardware.

Milestone 3 (April 29th)

- Connect all parts together and do joint debugging.(camera & sprite & sound).
- Screen calibration and color calibration.
- User interface.
- Try to realize changing modes and changing tones.

CHALENGES AND LESSONS LEARNT

Challenges

- Compile Linux kernel to support camera
- Storage problem
  The biggest problem for us is about where to store the pictures and the sound, because there is not enough space in on-chip memory, and we have no time to figure out how to use off-chip DDR3 memory. Hence, we figure out several ways to handle this problem.
  For the storage of pictures captured by camera, we use registers on board; for the storage of background pictures, we chop a whole big picture into a 128 * 340 picture and then duplicate it five times, and this saves us a lot space.
- Robustness of algorithm
  Camera is a very important key part of our project, and it directly affects the subsequent algorithms such as finger detection and key recognition. Hence, if something wrong happens to data sent from camera, the whole system would crash.
  Our way to deal with this problem is doing as much calibration as possible to make the camera robust. For example, in the screen calibration, we will guide the user to adjust the camera until it can capture a full view of keyboard area and the base line for other algorithms.
- Karplus-Strong algorithm
  Another big challenge for us is about audio part. In the beginning, we want to do audio part in software, store the sound in memory and send signals from hardware to call corresponding sound. However, as we mentioned before, storage is also a big problem. We don't have space to store all 16 notes into on-chip memory.
  So we gave up this method and started to consider doing audio part in hardware, and Karplus-Strong algorithm is widely used in synthesize real instrument sound. The principle is easy to understand from DSP view, but how to implement it in hardware is very hard, especially the low-pass filter and the delay line. We figured out that the delay line can be achieved by a shift register, and the low-pass filter is built by a mathematic model.

Lessons learned

- Plan/think ahead
  Doing software modeling and verification before implementing in hardware. Typically, hardware takes more time to compile and simulate, while software mocking up is very fast and easy to debug. Hence, if we use software in the beginning to verify the feasibility of our project, it will ensure that our algorithms indeed work. Actually we did this for our project. Before milestone 1, we completed all simulation in software and this provided us a good base for moving to hardware.
- Use tools like signaltap to do hardware debugging
  When we start to do the audio part, we had a really tough time because there was no sound coming out no matter what we did, and we couldn't figure out where the problem was. However, the TA taught us to use signal tap to check the output of modules, and when we looked at the output, we found out there existed output. This discovery made us realize that maybe the problem came from the board, which we never thought before. Hence, we changed a board and ran the

same project, and then the sound came out. This lesson tells us that when you encounter some problem, using tools to help debugging is very important, because it will help you find out the source of the problem quickly, instead of checking your correct code again and again.

- Think as a user
  Before milestone 3, we basically finished all basic function we want to realize for our project and we were all excited about the progress we made. However, the professor said that we should make it more user friendly and add some instructions for users, which we never though before. After that, when we began to modify our user interface, we realized that the previous user interface was really awful, because no one can play the game except us. So from that time, we try to think as users instead of designers, adding many instructions to give users full guide to play. We even write a user manual in this report.
- Pay attention to the completeness of project
  The project is not finished even if you've done all parts of the project, especially for video game. As mentioned before, you have to make it more user friendly so that anyone can play it without difficulty. Also, we should make it more like a mature product, not only a project for one class.
- Watch out for multiple levels of logic
  In our vga display module, we have many nest logic, and the screen often have problems showing what we want. And it turns out that many times it is because that the logic level is not correct, which leads to the coverage and overlapping between different logic controls. Hence, if you have multiple levels of logic, you must be very careful about the order of these levels, which is the top level logic, which is the bottom level of logic, etc.

FUTURE WORK

1) We can add more application for our system. We can add play mode, which users can press the key according to the dropping blocks.
2) We can make the system support 3 keys or more pressing together.
3) Furthermore, the automatically adjust according to illumination changing can be more accurate.
4) We can use the depth of field camera to replace the normal camera to get a better result.

CONCLUSION

We implement the function of touching screen using camera on a normal screen which has high accuracy, precision, and applicability. We use the function of touching screen to implement a 16 keys piano player which have free play mode, and audio play mode. We make a very complete and user friendly system which can be played without difficulty by anyone. We use Karplus-strong to generate the sound. Our system is very applicable in many fields. It can be used for a piano play game. It can also be used as an audio player like a MP3. Moreover, it can be used to make games or players for other instruments. A very accuracy touch screen is expensive. But a good camera is much cheaper. So our successful implement of touch screen using normal screen using camera gives a broad range of users a new way to enjoy the feeling of touch screen on a much lesser cost.

REFERENCE

1. Cornell project link:
http://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/s2013/cl972_rh523/cl972_rh523/index.html

2. Sockit user manual.

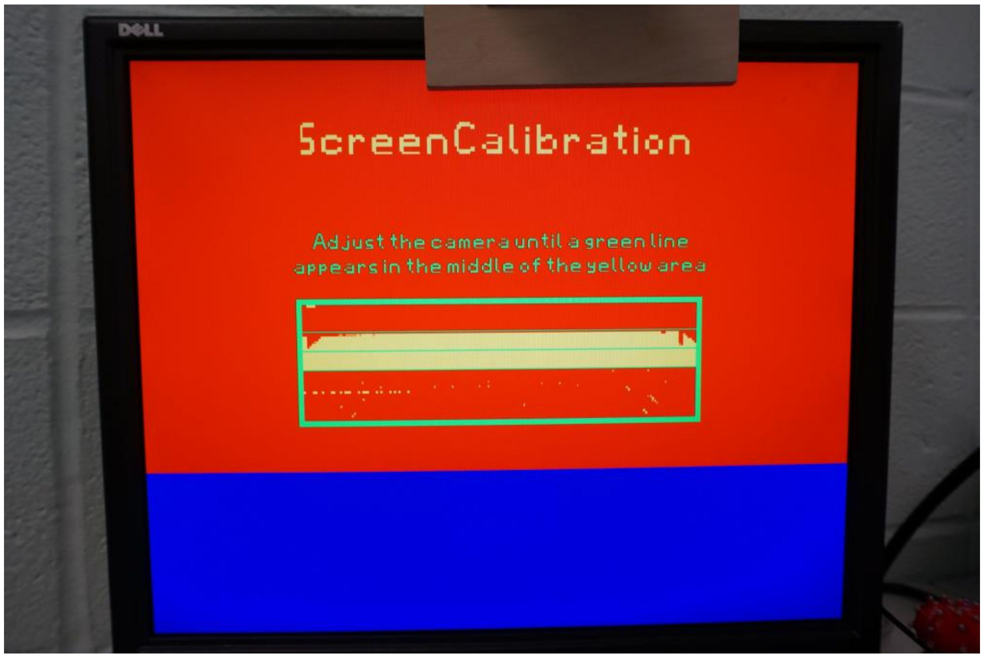
3. Audio codec tutorial: http://zhehaomao.com/blog/fpga/2014/01/15/sockit-8.html

4. SSM2603 datasheet.

APPENDIX

User manual

First page: screen calibration


DELL
ScreenCalibration
Adjust the camera until a green line
appears in the middle of the yellow area

Once the program is started, the first page appears to user is screen calibration page. In this page, the upper 340*640 pixels are red, while the lower 140*640 pixels are blue.

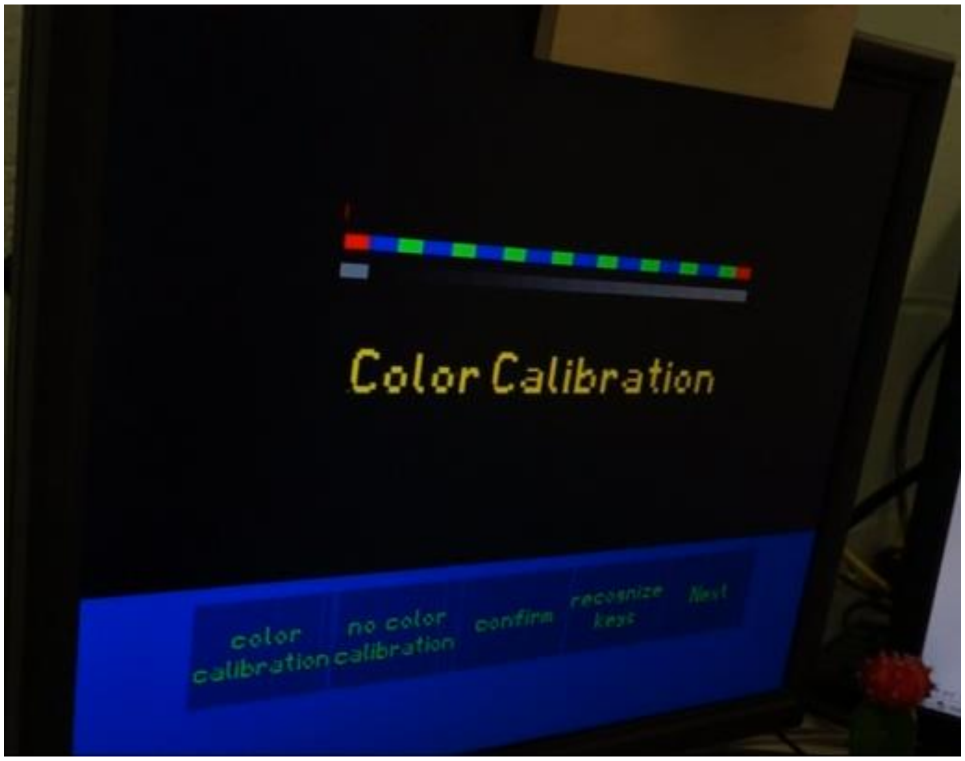
There is a green frame in the middle of the screen, which displays the compressed picture from the camera. In this frame, there is a rectangular yellow area, which represents the blue area on screen, and this is also the future keyboard area. This design is used to ensure that the camera can capture the full view the keyboard area, so that we won't get into a situation that some keys cannot be recognized.

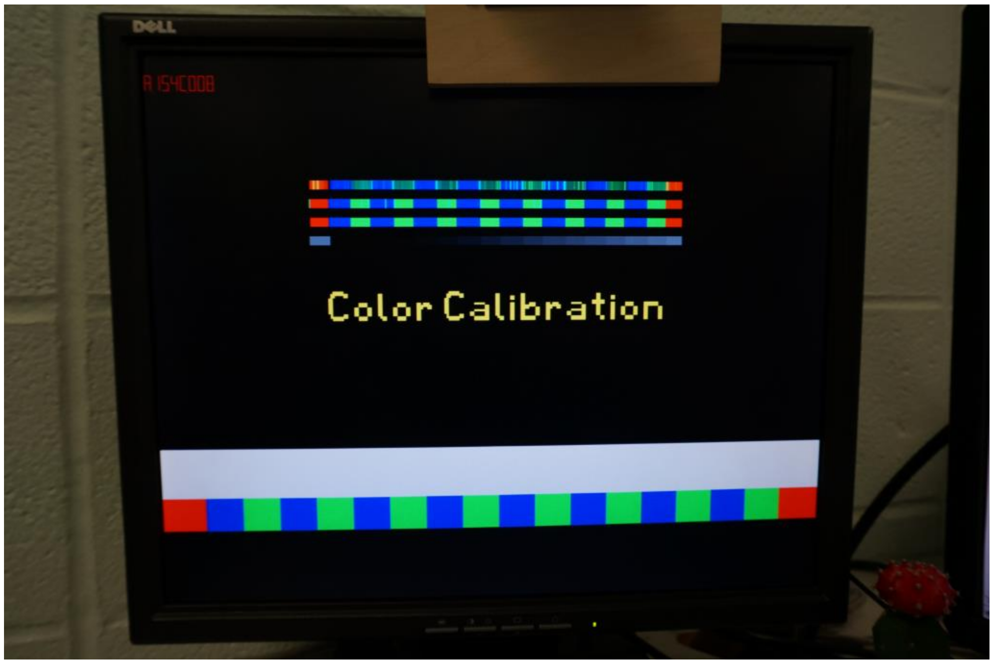
Above the green frame, there are two lines says that, "Adjust the camera until a green line appears in the middle of the yellow area". And now users should adjust the angle or position of the camera to make the green line appears in the middle. This step ensures that the camera can capture the base line for subsequent finger detection algorithm. If this step is done, there will be a button appears in the blue area denoted by "next". Click this button and we will enter the second page.

Also, users should pay attention that in order to get better experience of this project, this screen calibration is very important. In the process of adjusting the camera, there will appear two more green line except the middle on, and these two green line will appear near the upper bound and lower bound of the yellow area, and users should carefully adjust the camera until these two lines fit perfect to the upper and lower bound.

Second page: color calibration

Entering this page, users will see five buttons and some blue and green areas on lower screen and some rectangular area on the top of the screen, we will introduce the function of these components below.





- There are five buttons in the middle of the screen, which are denoted by ‘color calibration’, ‘no color calibration’, ‘confirm’, ‘recognize keys’, ‘next’, respectively.

- When users touch the ‘color calibration’ button, the screen will enter color calibration mode, and the name of this mode will appear in the upper screen. Also, there will be some blue and green area appears on the bottom of the screen. These area represent for 16 keys and are used for segmentation and recognize the position of each key. After finishing color calibration, the screen will return to the original second page.
- Above those keys, there is a white line, and below the keys, there is a black line, which are used to do color calibration, which is useful to make the system adjust the change of the light and correctly judge the range and the color of keys.
- There are several narrow lines on the upper area. The first line is original pixels of the base line sent from the camera. The second line is the RGB value for the base line after segmentation.
- When users touch the ‘no color calibration’ button, lines appears without color calibration, and this is not recommended for users.
- When users touch the ‘confirm’ button, the system actually will do the denoise work for the calibration results.
- When users touch the ‘recognize keys’ button, a third line will appear under the existing two lines on upper screen. This line shows the fixed key positions and different keys are represented by different gray value.
- When users touch the ‘next’ button, they will enter the third page.

It is strongly recommended that the user should click buttons in the order of ‘color calibration’, ‘denoise’ and ‘recognize keys’, and then finally this ‘next button’, in order to make the system do sufficient calibration and decrease the possibility of errors.

Also, in the process of doing calibration, since the original pixels information sent from camera and the results for segmentation and recognition of keys will appear on the screen, users can adjust the process more flexible, such as doing multiple times of color calibration and denoising to maximally eliminate the unstable factors in the environment.

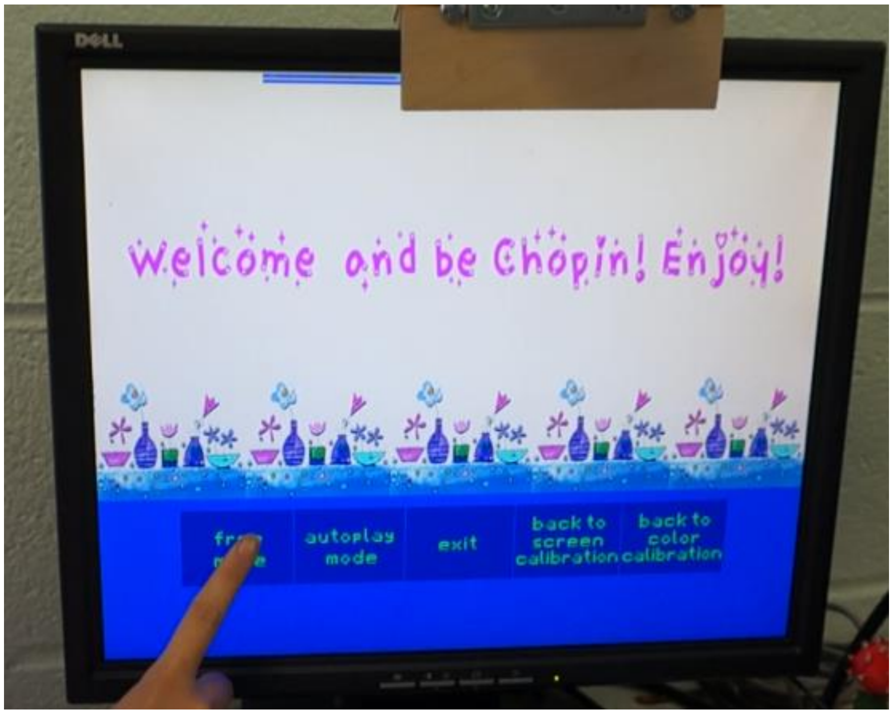
Third page: welcome page

The third page is welcome page. After users clicking the ‘next’ button in the second page, they will enter this page. There will be a lovely background and a welcome sentence displayed on screen. Since we support two different types of modes, this page is designed for user to choose the mode they want to play.

Similar to the second page, there are five buttons at the bottom part of the screen, which are denoted by ’ free mode’, ‘audio mode’, ‘exit’, ‘back to screen calibration’, ‘back to color calibration’ respectively.

a) When users touch the ‘free mode’ button, it will go to the fourth page, and users can play the piano freely. If users want to exit this mode, they can put their hand near the camera, and if the camera recognize that all keys are being pressed, it will return to the original third picture.

b) When users touch the 'autoplay mode' button, it will go to the fourth page, and people can choose a song name to type in. Then the song will be played automatically. In this mode, except for the songs that we put in the system, we also support user-defined songs. Users can type the numbered score of their favorite songs into a specific file and store, and then this song can be indexed and played by its name.
c) When users touch the 'exit' button, the program will terminate and users will exit from the game.
d) When users touch the 'back to screen calibration' button, we go back to the first page.
e) When users touch the 'back to color calibration' button, we go back to the second page.

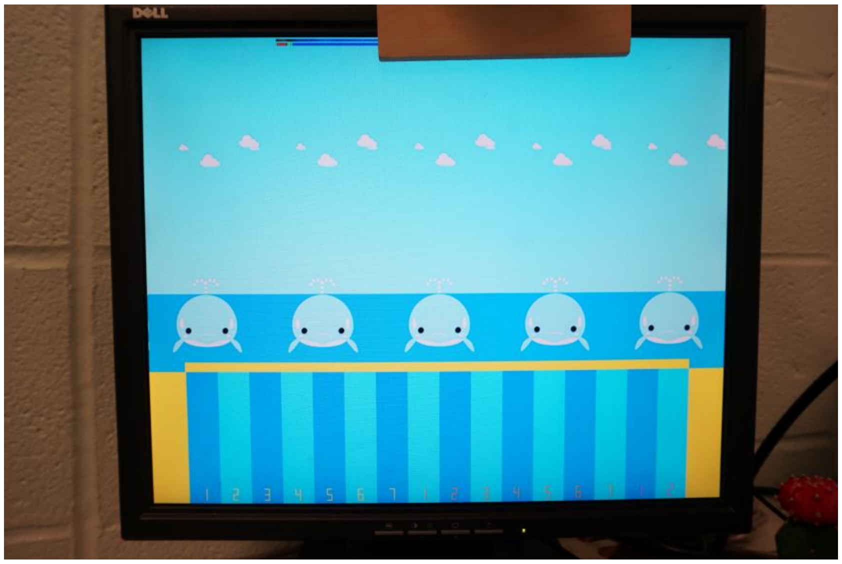

Welcome and be Chopin! Enjoy!
free mode
autoplay mode
exit
back to screen calibration
back to color calibration

Fourth page: play page

This page is the real page for play. The lower part of our screen is piano keys. And there is number shows on each key to help users easily find the key they want to play.

In this play page, we have two kinds of mode to play. Just as described above in the description of the third page.

When users press keys, pressed keys will change color to a darker blue with a black out boundary. And we support one key play sound, or two keys play sound together and play the mixing sound.



Source code

Software Control

```
/*
 * capturing from UVC cam
 * requires: libjpeg-dev
 * build: gcc -std=c99 piano_v5.c -ljpeg -o piano_v5
 */

#include <unistd.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <asm/types.h>
#include <linux/videodev2.h>

#include <sys/time.h>
#include <sys/types.h>

#include <jpeglib.h>
#include <inttypes.h>

#include "vga_led.h"
#include <sys/stat.h>

#include <linux/types.h>

#include <stdio.h>
#include <termios.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/time.h>

void changemode(int dir)
{
  static struct termios oldt, newt;

  if ( dir == 1 )
  {
    tcgetattr( STDIN_FILENO, &oldt);
    newt = oldt;
    newt.c_lflag &= ~( ICANON | ECHO );
    tcsetattr( STDIN_FILENO, TCSANOW, &newt);
  }
  else
    tcsetattr( STDIN_FILENO, TCSANOW, &oldt);
}

int kbhit (void)
{
  struct timeval tv;
  fd_set rdfs;

  tv.tv_sec = 0;
  tv.tv_usec = 0;

  FD_ZERO(&rdfs);
  FD_SET (STDIN_FILENO, &rdfs);

  select(STDIN_FILENO+1, &rdfs, NULL, NULL, &tv);
  return FD_ISSET(STDIN_FILENO, &rdfs);

}

int vga_led_fd;

/* Write the contents of the array to the display */
void write_segments(const unsigned char segs[1063])
{
	vga_led_arg_t vla;
	__u32 i;
	for (i = 1 ; i < VGA_LED_DIGITS ; i++) {
		vla.digit = i;
		vla.segments = segs[i];
		if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla)) {
			perror("ioctl(VGA_LED_WRITE_DIGIT) failed");
			return;
		}
	}
}

void quit(const char * msg)
{
	fprintf(stderr, "[%s] %d: %s\n", msg, errno, strerror(errno));
	exit(EXIT_FAILURE);
}

int xioctl(int fd, int request, void* arg)
```

```
{
	for (int i = 0; i < 100; i++) {
		int r = ioctl(fd, request, arg);
		if (r != -1 || errno != EINTR) return r;
	}
	return -1;
}

typedef struct {
	uint8_t* start;
	size_t length;
} buffer_t;

typedef struct {
	int fd;
	uint32_t width;
	uint32_t height;
	size_t buffer_count;
	buffer_t* buffers;
	buffer_t head;
} camera_t;


camera_t* camera_open(const char * device, uint32_t width, uint32_t height)
{
	int fd = open(device, O_RDWR | O_NONBLOCK, 0);
	if (fd == -1) quit("open");
	camera_t* camera = malloc(sizeof (camera_t));
	camera->fd = fd;
	camera->width = width;
	camera->height = height;
	camera->buffer_count = 0;
	camera->buffers = NULL;
	camera->head.length = 0;
	camera->head.start = NULL;
	return camera;
}

void camera_init(camera_t* camera)
{
	struct v4l2_capability cap;
	if (xioctl(camera->fd, VIDIOC_QUERYCAP, &cap) == -1) quit("VIDIOC_QUERYCAP");
	if (!(cap.capabilities & V4L2_CAP_VIDEO_CAPTURE)) quit("no capture");
	if (!(cap.capabilities & V4L2_CAP_STREAMING)) quit("no streaming");
	struct v4l2_cropcap cropcap;
	memset(&cropcap, 0, sizeof cropcap);
	cropcap.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
	if (xioctl(camera->fd, VIDIOC_CROPCAP, &cropcap) == 0) {
		struct v4l2_crop crop;
		crop.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
		crop.c = cropcap.defrect;
		if (xioctl(camera->fd, VIDIOC_S_CROP, &crop) == -1) {
		}
	}
	struct v4l2_format format;
	memset(&format, 0, sizeof format);
	format.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
	format.fmt.pix.width = camera->width;
	format.fmt.pix.height = camera->height;
	format.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
	format.fmt.pix.field = V4L2_FIELD_NONE;
	if (xioctl(camera->fd, VIDIOC_S_FMT, &format) == -1) quit("VIDIOC_S_FMT");
	struct v4l2_requestbuffers req;
	memset(&req, 0, sizeof req);
	req.count = 4;
	req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
	req.memory = V4L2_MEMORY_MMAP;
	if (xioctl(camera->fd, VIDIOC_REQBUFS, &req) == -1) quit("VIDIOC_REQBUFS");
	camera->buffer_count = req.count;
	camera->buffers = calloc(req.count, sizeof (buffer_t));
	size_t buf_max = 0;
	for (size_t i = 0; i < camera->buffer_count; i++) {
		struct v4l2_buffer buf;
		memset(&buf, 0, sizeof buf);
		buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
		buf.memory = V4L2_MEMORY_MMAP;
		buf.index = i;
		if (xioctl(camera->fd, VIDIOC_QUERYBUF, &buf) == -1)
quit("VIDIOC_QUERYBUF");
		if (buf.length > buf_max) buf_max = buf.length;
		camera->buffers[i].length = buf.length;
		camera->buffers[i].start =  mmap(NULL, buf.length, PROT_READ | PROT_WRITE,
MAP_SHARED, camera->fd, buf.m.offset);
		if (camera->buffers[i].start == MAP_FAILED) quit("mmap");
	}
	camera->head.start = malloc(buf_max);
}


void camera_start(camera_t* camera)
{
	for (size_t i = 0; i < camera->buffer_count; i++) {
		struct v4l2_buffer buf;
		memset(&buf, 0, sizeof buf);
```

```
                buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                buf.memory = V4L2_MEMORY_MMAP;
                buf.index = i;
                if (xioctl(camera->fd, VIDIOC_QBUF, &buf) == -1) quit("VIDIOC_QBUF");
        }
        enum v4l2_buf_type type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        if (xioctl(camera->fd, VIDIOC_STREAMON, &type) == -1) quit("VIDIOC_STREAMON");
}

void camera_stop(camera_t* camera)
{
        enum v4l2_buf_type type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        if (xioctl(camera->fd, VIDIOC_STREAMOFF, &type) == -1) quit("VIDIOC_STREAMOFF");
}

void camera_finish(camera_t* camera)
{
        for (size_t i = 0; i < camera->buffer_count; i++) {
                munmap(camera->buffers[i].start, camera->buffers[i].length);
        }
        free(camera->buffers);
        camera->buffer_count = 0;
        camera->buffers = NULL;
        free(camera->head.start);
        camera->head.length = 0;
        camera->head.start = NULL;
}

void camera_close(camera_t* camera)
{
        if (close(camera->fd) == -1) quit("close");
        free(camera);
}


int camera_capture(camera_t* camera)
{
        struct v4l2_buffer buf;
        memset(&buf, 0, sizeof buf);
        buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        buf.memory = V4L2_MEMORY_MMAP;
        if (xioctl(camera->fd, VIDIOC_DQBUF, &buf) == -1) return FALSE;
        memcpy(camera->head.start, camera->buffers[buf.index].start, buf.bytesused);
        camera->head.length = buf.bytesused;
        if (xioctl(camera->fd, VIDIOC_QBUF, &buf) == -1) return FALSE;
        return TRUE;
}
```

```
int camera_frame(camera_t* camera, struct timeval timeout) {
	fd_set fds;
	FD_ZERO(&fds);
	FD_SET(camera->fd, &fds);
	int r = select(camera->fd + 1, &fds, 0, 0, &timeout);
	if (r == -1) quit("select");
	if (r == 0) return FALSE;
	return camera_capture(camera);
}

void jpeg(FILE* dest, uint8_t* rgb, uint32_t width, uint32_t height, int quality)
{
	JSAMPARRAY image;
	image = calloc(height, sizeof (JSAMPROW));
	for (size_t i = 0; i < height; i++) {
		image[i] = calloc(width * 3, sizeof (JSAMPLE));
		for (size_t j = 0; j < width; j++) {
			image[i][j * 3 + 0] = rgb[(i * width + j) * 3 + 0];
			image[i][j * 3 + 1] = rgb[(i * width + j) * 3 + 1];
			image[i][j * 3 + 2] = rgb[(i * width + j) * 3 + 2];
		}
	}
	struct jpeg_compress_struct compress;
	struct jpeg_error_mgr error;
	compress.err = jpeg_std_error(&error);
	jpeg_create_compress(&compress);
	jpeg_stdio_dest(&compress, dest);
	compress.image_width = width;
	compress.image_height = height;
	compress.input_components = 3;
	compress.in_color_space = JCS_RGB;
	jpeg_set_defaults(&compress);
	jpeg_set_quality(&compress, quality, TRUE);
	jpeg_start_compress(&compress, TRUE);
	jpeg_write_scanlines(&compress, image, height);
	jpeg_finish_compress(&compress);
	jpeg_destroy_compress(&compress);
	for (size_t i = 0; i < height; i++) {
		free(image[i]);
	}
	free(image);
}

int minmax(int min, int v, int max)
{
	return (v < min) ? min : (max < v) ? max : v;
```

```
}

uint8_t *disrgb(uint8_t* rgb, uint32_t width, uint32_t height)
{
        int count, upline, downline, up, down;
        int blackline_begin, blackline_end, whiteline_begin, whiteline_end;
        int whitelinesum, blacklinesum;
        int bitcount = 0;
        uint8_t* baseline = calloc(width * 3, sizeof (uint8_t));
        upline = 0;
        downline = 0;
        up = 0;
        down = 0;
        count = 0;
        blackline_begin = 1;
        whiteline_end = 1;

        for (size_t i = 0; i < width * 3; i ++) {
                baseline[i] = 0;
        }

        for (size_t i = 114; i < 209; i += 2) {
                for (size_t j = 0; j < width * 3; j += 6) {
                        size_t index = i * width * 3 + 1053 - j;
                        if ((rgb[index] > rgb[index+2])) {
                                if (bitcount == 0) baseline[count] += 1;
                                else if (bitcount == 1) baseline[count] += 2;
                                else if (bitcount == 2) baseline[count] += 4;
                                else if (bitcount == 3) baseline[count] += 8;
                                else if (bitcount == 4) baseline[count] += 16;
                                else if (bitcount == 5) baseline[count] += 32;
                                else if (bitcount == 6) baseline[count] += 64;
                                else baseline[count] += 128;
                        }
                        if (bitcount == 7) {
                                bitcount = 0;
                                count += 1;
                        }
                        else bitcount += 1;
                }
        }

        for (size_t i = 114; i < 209; i += 2) {
                count = 0;
           for (size_t j = 0; j < width * 3; j += 9) {
                   size_t index = i * width * 3 + 1053 - j;
                        if (rgb[index] > rgb[index+2]) {
```

```
                                    count += 1;
                              }
               }
               if (count > width * 3 / 9 * 0.8) {
                              count = 2;
                              if (up == 0) up = i;
                      }
               else {
                      count = 0;
                      if (down == 0 && up != 0) down = i;
            }
            }

            baseline[792] = (up + down) / 2 /2;
            baseline[793] = (up + down) / 2 /2;
            blacklinesum = (up + down) / 2 - (down - up) / 3;
            whitelinesum = (up + down) / 2 + (down - up) / 3;

            baseline[794] = blacklinesum / 2;
            baseline[795] = blacklinesum / 2;
            baseline[796] = whitelinesum / 2;
            baseline[797] = whitelinesum / 2;
            //printf("Base line is: %d. From %d to %d.\n", baseline[792] + baseline[793], up, down);
            //printf("black line is: %d.\n", blacklinesum);
            //printf("white line is: %d.\n", whitelinesum);
            //printf("\n\n");
            return baseline;
}

uint8_t *finger_detection(uint8_t* rgb, uint32_t width, uint32_t height, int captureline)
{
            int finger_count[5], allbutton;
            allbutton = 0;
            uint8_t* finger_position = calloc(6, sizeof (uint8_t));
            int finger_detect[1056];
            for (size_t i = 0; i < 6; i++) finger_count[i] = 0;
            for (size_t i = 0; i < width * 3; i += 3) {
                      size_t index = (captureline + 1) * width * 3 - i;
               if ((rgb[index - 2] > 130) & (rgb[index - 1] > 130)) {
                              finger_detect[i] = 1;
                              allbutton++;
                      }
                      else
                              finger_detect[i] = 0;
                      if ((i >= 39 * 3) & (i < 94 * 3) & (finger_detect[i] == 1)) finger_count[0]++;
                      else if ((i >= 94 * 3) & (i < 149 * 3) & (finger_detect[i] == 1)) finger_count[1]++;
                      else if ((i >= 149 * 3) & (i < 204 * 3) & (finger_detect[i] == 1)) finger_count[2]++;
```

```
                else if ((i >= 204 * 3) & (i < 259 * 3) & (finger_detect[i] == 1)) finger_count[3]++;
                else if ((i >= 259 * 3) & (i < 314 * 3) & (finger_detect[i] == 1)) finger_count[4]++;
        }
        for (size_t i = 0; i < 5; i++) {
                if (finger_count[i] > 8) {
                        finger_position[i] = 1;
                        //printf("Button %d is pressed!\n", i);
                }
                else finger_position[i] = 0;
        }
        finger_position[5] = 0;
        if (allbutton > 300) finger_position[5] = 1;
        //printf("%d is pressed!\n", finger_position[5]);
        return finger_position;
}

uint8_t* color(uint8_t* rgb, uint32_t width, uint32_t height, int baseline, int blackline, int whiteline)
{
        double whitevalue = 255, blackvalue = 0, pixelvalue = 0, uppervalue = 255;
        double rate;
      blackline = blackline * width * 3;
        whiteline = whiteline * width * 3;
        baseline = baseline * width *3;
        for (size_t i = 0; i < width * 3; i++)
        {
                whitevalue = rgb[whiteline + i];
                blackvalue = rgb[blackline + i];
                pixelvalue = rgb[baseline + i];
                uppervalue = whitevalue - blackvalue;
                rate = 255 / uppervalue;
                rgb[baseline + i] = minmax(0, (int)((pixelvalue - blackvalue) * rate), 255);
        }
        return rgb;
}

uint8_t* yuyv2rgb(uint8_t* yuyv, uint32_t width, uint32_t height)
{
        uint8_t* rgb = calloc(width * height * 3, sizeof (uint8_t));
        for (size_t i = 0; i < height; i++) {
                for (size_t j = 0; j < width; j += 2) {
                        size_t index = i * width + j;
                        int y0 = yuyv[index * 2 + 0]-0;
                        int u = yuyv[index * 2 + 1] - 148;
                        int y1 = yuyv[index * 2 + 2]-0;
                        int v = yuyv[index * 2 + 3] - 128;
                        rgb[index * 3 + 2] = minmax(0, y0 + ((359 * v) >> 8), 255);
                        rgb[index * 3 + 1] = minmax(0, y0 - ((88 * v + 183 * u) >> 8), 255);
                        rgb[index * 3 + 0] = minmax(0, y0 + ((454 * u) >> 8), 255);
                        rgb[index * 3 + 5] = minmax(0, y1 + ((359 * v) >> 8), 255);
                        rgb[index * 3 + 4] = minmax(0, y1 - ((88 * v + 183 * u) >> 8), 255);
                        rgb[index * 3 + 3] = minmax(0, y1 + ((454 * u) >> 8), 255);
                }
        }
        return rgb;
}


uint8_t* coloryuyv2rgb(uint8_t* yuyv, uint8_t* black, uint8_t* white, uint32_t width, int capture)
{
        double whitevalue = 255, blackvalue = 0, pixelvalue = 0, uppervalue = 255;
        double rate;
        uint8_t* rgb = calloc(width * 3, sizeof (uint8_t));
        for (size_t i = 0; i < width; i += 2) {
                size_t index = capture * width + i;
                int y0 = yuyv[index * 2 + 0]-0;
                int u = yuyv[index * 2 + 1] - 148;
                int y1 = yuyv[index * 2 + 2]-0;
                int v = yuyv[index * 2 + 3] - 128;
                rgb[i * 3 + 2] = minmax(0, y0 + ((359 * v) >> 8), 255);
                rgb[i * 3 + 1] = minmax(0, y0 - ((88 * v + 183 * u) >> 8), 255);
                rgb[i * 3 + 0] = minmax(0, y0 + ((454 * u) >> 8), 255);
                rgb[i * 3 + 5] = minmax(0, y1 + ((359 * v) >> 8), 255);
                rgb[i * 3 + 4] = minmax(0, y1 - ((88 * v + 183 * u) >> 8), 255);
                rgb[i * 3 + 3] = minmax(0, y1 + ((454 * u) >> 8), 255);
        }
        for (size_t i = 0; i < width * 3; i++)
        {
                whitevalue = white[i];
                blackvalue = black[i];
                pixelvalue = rgb[i];
                uppervalue = whitevalue - blackvalue;
                rate = 255 / uppervalue;
                rgb[i] = minmax(0, (int)((pixelvalue - blackvalue) * rate), 255);
        }
        return rgb;
}
int main()
{
        int width = 352;
        int height = 288;
        int pause = 0;

        camera_t* camera = camera_open("/dev/video0", width, height);
        camera_init(camera);
```

```
	camera_start(camera);

	struct timeval timeout;
	timeout.tv_sec = 1;
	timeout.tv_usec = 0;

	/* skip 5 frames for booting a cam */
	for (int i = 0; i < 50; i++) camera_frame(camera, timeout);

	unsigned char* rgb, rgb_simple;
	uint8_t* black = calloc(width * 3, sizeof (uint8_t));
	uint8_t* white = calloc(width * 3, sizeof (uint8_t));
	uint8_t* baseline = calloc(width * 3, sizeof (uint8_t));

	vga_led_arg_t vla;
	int i, display_row, captureline = 0, captureline_1 = 0, captureline_2 = 0, blackline = 0, whiteline
= 0;
	int mode_select;

	static const char filename[] = "/dev/vga_led";
	static unsigned char message[1063];

	int charcount = 0, stringcount = 0, stringnum = 0, backcharcount = 0, backstringcount = 0;
	char ch;
	char songtimechar[3];
	int songtime = 0, backsongtime = 0;
	char string[2];
      int songstring[100];
	int backsongstring[60];

	int screen_pressed[6];
	for (i = 0; i < 6; i++) {
		screen_pressed[i] = 0;
	}

	uint8_t* finger_position = calloc(6, sizeof (uint8_t));

	if ( (vga_led_fd = open(filename, O_RDWR)) == -1) {
		fprintf(stderr, "could not open %s\n", filename);
		return -1;
	}

	FILE *play_song;
	play_song = fopen("mo_li_hua.txt", "r");

	charcount = 0;
	ch = fgetc(play_song);
```

```
while (ch != ' ') {
            songtimechar[charcount] = ch;
            charcount += 1;
    ch = fgetc(play_song);
}
backsongtime = atoi(songtimechar);

stringcount = 0;

while (ch != '\n') {
            charcount = 0;
            ch = fgetc(play_song);
            while (ch != ' ' && ch != '\n') {
                        string[charcount] = ch;
                        charcount += 1;
            ch = fgetc(play_song);
            }
            backsongstring[stringcount] = atoi(string);
            stringcount += 1;
}

fclose(play_song);
stringcount = 0;
charcount = 0;


rgb = yuyv2rgb(camera->head.start, camera->width, camera->height);
baseline = disrgb(rgb, camera->width, camera->height);

mode_select = 0;
while (1) {

            pause = 0;

            if (mode_select == 0)
            {
                        while (1) {
                                    camera_frame(camera, timeout);
                                    rgb = yuyv2rgb(camera->head.start, camera->width, camera->height);

                                    baseline = disrgb(rgb, camera->width, camera->height);

                                    captureline = baseline[792] + baseline[793];
                                    blackline = baseline[794] + baseline[795];
                                    whiteline = baseline[796] + baseline[797];

                                    if ((captureline >= 114) && (captureline <= 209)) {
```

```
					finger_position = finger_detection(rgb, camera->width, camera-
>height, captureline);
					if (finger_position[2] == 1) {
						message[1058] = 2;
						screen_pressed[2] = 1;
					}
					else if ((finger_position[2] == 0) && (screen_pressed[2] == 1)) {
						message[1058] = 3;
						mode_select = 3;
						screen_pressed[2] = 0;
						write_segments(message);
						free(rgb);
						break;
					}
					else message[1058] = 1;
					//printf("%d\n", finger_position[2]);
				}
				else message[1058] = 0;

				display_row = width * 3;
				for (i = 0; i < display_row; i++) {
					message[i] = baseline[i];
				}
				message[1056] = baseline[792];
				message[1057] = baseline[793];
				write_segments(message);
				free(rgb);
			}
			printf("Screen Calibration finished.\n\n");
			continue;
		}
		else if (mode_select == 4)
		{
			for (i = 0; i < 20; i++) camera_frame(camera, timeout);
			printf("Entry Color Calibration.\n");

			while (1) {
				camera_frame(camera, timeout);
				rgb = coloryuyv2rgb(camera->head.start, black, white, camera->width,
captureline);

				finger_position = finger_detection(rgb, camera->width, camera->height,
0);

				for (i = 0; i < width * 3; i++) {
					message[i] = rgb[width * 3 - 1 - i];
				}
```

```
if (finger_position[0] == 1) {
        message[1058] = 5;
        screen_pressed[0] = 1;
}
else if ((finger_position[0] == 0) && (screen_pressed[0] == 1)) {
        message[1058] = 3;
        mode_select = 3;
        screen_pressed[0] = 0;
        write_segments(message);
        free(rgb);
        break;
}
else if (finger_position[1] == 1) {
        message[1058] = 6;
        screen_pressed[1] = 1;
}
else if ((finger_position[1] == 0) && (screen_pressed[1] == 1)) {
        message[1058] = 3;
        mode_select = 6;
        screen_pressed[1] = 0;
        write_segments(message);
        free(rgb);
        break;
}
else if (finger_position[2] == 1) {
        message[1058] = 7;
        screen_pressed[2] = 1;
}
else if ((finger_position[2] == 0) && (screen_pressed[2] == 1)) {
        message[1058] = 4;
        mode_select = 4;
        screen_pressed[2] = 0;
        free(rgb);
        printf("Denoise pixel.\n\n");
        break;
}
else if (finger_position[3] == 1) {
        message[1058] = 8;
        screen_pressed[3] = 1;
}
else if ((finger_position[3] == 0) && (screen_pressed[3] == 1)) {
        message[1058] = 4;
        mode_select = 4;
        screen_pressed[3] = 0;
        free(rgb);
        printf("Recognize keys.\n\n");
```

```
                        break;
                }
                else if (finger_position[4] == 1) {
                        message[1058] = 9;
                        screen_pressed[4] = 1;
                }
                else if ((finger_position[4] == 0) && (screen_pressed[4] == 1)) {
                        message[1058] = 10;
                        mode_select = 10;
                        screen_pressed[4] = 0;
                        write_segments(message);
                        free(rgb);
                        printf("Color Calibration function select finished.\n\n");
                        break;
                }

                write_segments(message);
                free(rgb);
        }
        continue;
}
else if (mode_select == 3)
{
        while (pause < 50)
        {
                pause += 1;
                camera_frame(camera, timeout);
                rgb = yuyv2rgb(camera->head.start, camera->width, camera->height);

                rgb = color(rgb, camera->width, camera->height, captureline, blackline,
whiteline);

                display_row = captureline * 1056;
                //display_row = blackline * 1056;
                //display_row = whiteline * 1056;
                for (i = display_row; i < display_row + 1056; i++) {
                        message[i - display_row] = rgb[display_row + display_row + 1055
- i];
                }
                message[1058] = 3;
                write_segments(message);

                for (size_t i = 0; i < width * 3; i++)
                {
                        black[i] = rgb[blackline * width * 3 + i];
                        white[i] = rgb[whiteline * width * 3 + i];
                }
```

```
                        free(rgb);
                    }
                    message[1058] = 4;
                    mode_select = 4;
                    write_segments(message);
                    printf("Color Calibration finished.\n\n");
                    continue;
                }
                else if (mode_select == 6)
                {
                    while (pause < 50)
                    {
                        pause += 1;
                        camera_frame(camera, timeout);
                        rgb = yuyv2rgb(camera->head.start, camera->width, camera->height);

                        display_row = captureline * 1056;
                        for (i = display_row; i < display_row + 1056; i++) {
                            message[i - display_row] = rgb[display_row + display_row + 1055
- i];
                        }
                        message[1058] = 3;
                        write_segments(message);
                        free(rgb);
                    }
                    message[1058] = 4;
                    mode_select = 4;
                    write_segments(message);
                    printf("Display original pixel finished.\n\n");
                    continue;
                }
                else if (mode_select == 10)
                {
                    for (i = 0; i < 20; i++) camera_frame(camera, timeout);
                    printf("Entry Welcome page.\n");
                    while (1)
                    {
                        camera_frame(camera, timeout);
                        //rgb = yuyv2rgb(camera->head.start, camera->width, camera->height);
                        rgb = coloryuyv2rgb(camera->head.start, black, white, camera->width,
captureline);

                        finger_position = finger_detection(rgb, camera->width, camera->height,
0);

                        for (i = 0; i < width * 3; i++) {
                            message[i] = rgb[width * 3 - 1 - i];
```

```
				}

				if (finger_position[0] == 1) {
					message[1058] = 11;
					message[1059] = 0;
					screen_pressed[0] = 1;
				}
				else if ((finger_position[0] == 0) && (screen_pressed[0] == 1)) {
					message[1058] = 16;
					message[1059] = 0;
					mode_select = 16;
					screen_pressed[0] = 0;
					write_segments(message);
					free(rgb);
					printf("Entry to the freedom mode.\n\n");
					break;
				}
				else if (finger_position[1] == 1) {
					message[1058] = 12;
					message[1059] = 0;
					write_segments(message);

					for (i = 0; i < 100; i++) {
						songstring[i] = -1;
					}

					FILE *play_song;
					while (1) {
						char inputfile[50];
						printf("Welcome to audio player mode.\n");
						printf("You can choose any song in below list or type your
defined song file.\n*\n");

						printf("mo_li_hua.txt\n");
						printf("my_heart.txt\n");
						printf("littlestar.txt\n");
						printf("let_it_go.txt\n");
						printf("\nPlease type the song file name: ");
						scanf("%s", inputfile);
						printf("Song file: %s\n", inputfile);
						play_song = fopen(inputfile, "r");
						if (play_song != NULL) break;
				}

					charcount = 0;
					ch = fgetc(play_song);
					while (ch != ' ') {
						songtimechar[charcount] = ch;
```

```
                        charcount += 1;
                ch = fgetc(play_song);
                    }
                    songtime = atoi(songtimechar);

                    stringcount = 0;

                    while (ch != '\n') {
                        charcount = 0;
                        ch = fgetc(play_song);
                        while (ch != ' ' && ch != '\n') {
                            string[charcount] = ch;
                            charcount += 1;
                        ch = fgetc(play_song);
                        }
                        songstring[stringcount] = atoi(string);
                        stringcount += 1;
                    }

                    fclose(play_song);
                    printf("Song time is: %d\n", songtime);
                    printf("Test string is: ");
                    stringnum = 0;
                    while (songstring[stringnum] != -1) {
                        printf("%d ", songstring[stringnum]);
                        stringnum += 1;
                    }
                    printf("\n");

                    stringcount = 0;
                    charcount = 0;

                    message[1058] = 18;
                    mode_select = 18;
                    write_segments(message);
                    free(rgb);
                    printf("Entry to the Song player mode.\n\n");
                    break;
                }
                else if (finger_position[2] == 1) {
                    message[1058] = 13;
                    message[1059] = 0;
                    screen_pressed[2] = 1;
                }
                else if ((finger_position[2] == 0) && (screen_pressed[2] == 1)) {
                    message[1058] = 20;
                    message[1059] = 0;
```

```
        mode_select = 20;
        screen_pressed[2] = 0;
        write_segments(message);
        free(rgb);
        printf("Exit.\n\n");
        break;
}
else if (finger_position[3] == 1) {
        message[1058] = 14;
        screen_pressed[3] = 1;
}
else if ((finger_position[3] == 0) && (screen_pressed[3] == 1)) {
        message[1058] = 0;
        mode_select = 0;
        screen_pressed[3] = 0;
        write_segments(message);
        free(rgb);
        printf("Entry to the Screen Calibration mode.\n\n");
        break;
}
else if (finger_position[4] == 1) {
        message[1058] = 15;
        screen_pressed[4] = 1;
}
else if ((finger_position[4] == 0) && (screen_pressed[4] == 1)) {
        message[1058] = 4;
        mode_select = 4;
        screen_pressed[4] = 0;
        write_segments(message);
        free(rgb);
        printf("Entry to the Color Calibration mode.\n\n");
        break;
}

if (backcharcount < backsongtime / 2) {
        message[1059] = backsongstring[backstringcount];
        backcharcount += 1;
}
else if (backcharcount == backsongtime) {
        message[1059] = 0;
        backstringcount += 1;
        backcharcount = 0;
}
else {
        message[1059] = 0;
        backcharcount += 1;
}
```

```
                    if (backstringcount == 60) backstringcount = 0;
                    write_segments(message);
                    free(rgb);
                }
                continue;
            }
            else if (mode_select == 16)
            {
                while (1)
                {
                    camera_frame(camera, timeout);

                    rgb = coloryuyv2rgb(camera->head.start, black, white, camera->width,
capturelіne);

                    finger_position = finger_detection(rgb, camera->width, camera->height,
0);

                    if (finger_position[5] == 1) {
                        message[1058] = 17;
                        screen_pressed[5] = 1;
                        write_segments(message);
                    }
                    else if ((finger_position[5] == 0) && (screen_pressed[5] == 1)) {
                        message[1058] = 10;
                        mode_select = 10;
                        screen_pressed[5] = 0;
                        write_segments(message);
                        printf("Entry to the Welcome.\n\n");
                        break;
                    }
                    else
                    {
                        message[1058] = 16;

                        for (i = 0; i < width * 3; i++) {
                            message[i] = rgb[width * 3 - 1 - i];
                        }
                        write_segments(message);
                    }
                    free(rgb);
                }
                message[1058] = 10;
                mode_select = 10;
                write_segments(message);
                printf("Free mode terminaled.\n\n");
                continue;
            }
            else if (mode_select == 18)
            {

                while ((!kbhit()) && (stringcount < stringnum)) {
                    camera_frame(camera, timeout);
                    rgb = coloryuyv2rgb(camera->head.start, black, white, camera->width,
capturelinel);

                    finger_position = finger_detection(rgb, camera->width, camera->height,
0);

                    if (finger_position[5] == 1) {
                        message[1058] = 19;
                        screen_pressed[5] = 1;
                        write_segments(message);
                    }
                    else if ((finger_position[5] == 0) && (screen_pressed[5] == 1)) {
                        message[1058] = 10;
                        mode_select = 10;
                        screen_pressed[5] = 0;
                        write_segments(message);
                        printf("Entry to the Welcome.\n\n");
                        break;
                    }
                    else
                    {
                        for (i = 0; i < width * 3; i++) {
                            message[i] = rgb[width * 3 - 1 - i];
                        }

                        if (charcount < songtime / 2) {
                            message[1059] = songstring[stringcount];
                            charcount += 1;
                        }
                        else if (charcount == songtime) {
                            message[1059] = 0;
                            stringcount += 1;
                            charcount = 0;
                        }
                        else {
                            message[1059] = 0;
                            charcount += 1;
                        }
                        printf("%d %d\n", stringcount, charcount);
                        write_segments(message);
```

```
                                        }
                                        free(rgb);
                                }
                                message[1058] = 10;
                                mode_select = 10;
                                write_segments(message);
                                printf("Song Player mode terminaled.\n\n");
                                continue;
                        }
                        else
                        {
                                printf("Program terminaled.\n");
                                camera_stop(camera);
                                camera_finish(camera);
                                camera_close(camera);
                                return 0;
                        }
                }
}
```

SocKit top file

```
//
======================================================================
==
// Copyright (c) 2013 by Terasic Technologies Inc.
//
======================================================================
==
//
// Permission:
//
//   Terasic grants permission to use and modify this code for use
//   in synthesis for all Terasic Development Boards and Altera Development
//   Kits made by Terasic.  Other use of this code, including the selling
//   ,duplication, or modification of any portion is strictly prohibited.
//
// Disclaimer:
//
//   This VHDL/Verilog or C/C++ source code is intended as a design reference
//   which illustrates how these types of functions can be implemented.
//   It is the user's responsibility to verify their design for
//   consistency and functionality through the use of formal
//   verification methods.  Terasic provides no warranty regarding the use
//   or functionality of this code.
```

```
//
//
==============================================================================
==
//
//  Terasic Technologies Inc
//  9F., No.176, Sec.2, Gongdao 5th Rd, East Dist, Hsinchu City, 30070. Taiwan
//
//
//                    web: http://www.terasic.com/
//                    email: support@terasic.com
//
//
==============================================================================
==
//
==============================================================================
==
//
// Major Functions:     SoCKit_Default
//
//
==============================================================================
==
// Revision History :
//
==============================================================================
==
//   Ver  :| Author              :| Mod. Date :| Changes Made:
//   V1.0 :| xinxian              :| 04/02/13  :| Initial Revision
//
==============================================================================
==

//`define ENABLE_DDR3
//`define ENABLE_HPS
//`define ENABLE_HSMC_XCVR

module SoCKit_top(

                      ///////////AUD////////////
                      AUD_ADCDAT,
                      AUD_ADCLRCK,
                      AUD_BCLK,
                      AUD_DACDAT,
                      AUD_DACLRCK,
                      AUD_I2C_SCLK,
                      AUD_I2C_SDAT,
                      AUD_MUTE,
                      AUD_XCK,

`ifdef ENABLE_DDR3
                      /////////DDR3/////////
                      DDR3_A,
                      DDR3_BA,
                      DDR3_CAS_n,
                      DDR3_CKE,
                      DDR3_CK_n,
                      DDR3_CK_p,
                      DDR3_CS_n,
                      DDR3_DM,
                      DDR3_DQ,
                      DDR3_DQS_n,
                      DDR3_DQS_p,
                      DDR3_ODT,
                      DDR3_RAS_n,
                      DDR3_RESET_n,
                      DDR3_RZQ,
                      DDR3_WE_n,
`endif /*ENABLE_DDR3*/

                      /////////FAN/////////
                      FAN_CTRL,

`ifdef ENABLE_HPS
                      /////////HPS/////////
                      HPS_CLOCK_25,
                      HPS_CLOCK_50,
                      HPS_CONV_USB_n,
                      HPS_DDR3_A,
                      HPS_DDR3_BA,
                      HPS_DDR3_CAS_n,
                      HPS_DDR3_CKE,
                      HPS_DDR3_CK_n,
                      HPS_DDR3_CK_p,
                      HPS_DDR3_CS_n,
                      HPS_DDR3_DM,
                      HPS_DDR3_DQ,
                      HPS_DDR3_DQS_n,
                      HPS_DDR3_DQS_p,
                      HPS_DDR3_ODT,
                      HPS_DDR3_RAS_n,
                      HPS_DDR3_RESET_n,
                      HPS_DDR3_RZQ,
			HPS_DDR3_WE_n,
			HPS_ENET_GTX_CLK,
			HPS_ENET_INT_n,
			HPS_ENET_MDC,
			HPS_ENET_MDIO,
			HPS_ENET_RESET_n,
			HPS_ENET_RX_CLK,
			HPS_ENET_RX_DATA,
			HPS_ENET_RX_DV,
			HPS_ENET_TX_DATA,
			HPS_ENET_TX_EN,
			HPS_FLASH_DATA,
			HPS_FLASH_DCLK,
			HPS_FLASH_NCSO,
			HPS_GSENSOR_INT,
			HPS_I2C_CLK,
			HPS_I2C_SDA,
			HPS_KEY,
			HPS_LCM_D_C,
			HPS_LCM_RST_N,
			HPS_LCM_SPIM_CLK,
			HPS_LCM_SPIM_MISO,
			HPS_LCM_SPIM_MOSI,
			HPS_LCM_SPIM_SS,
			HPS_LED,
			HPS_LTC_GPIO,
			HPS_RESET_n,
			HPS_SD_CLK,
			HPS_SD_CMD,
			HPS_SD_DATA,
			HPS_SPIM_CLK,
			HPS_SPIM_MISO,
			HPS_SPIM_MOSI,
			HPS_SPIM_SS,
			HPS_SW,
			HPS_UART_RX,
			HPS_UART_TX,
			HPS_USB_CLKOUT,
			HPS_USB_DATA,
			HPS_USB_DIR,
			HPS_USB_NXT,
			HPS_USB_RESET_PHY,
			HPS_USB_STP,
			HPS_WARM_RST_n,
`endif /*ENABLE_HPS*/

			/////////HSMC/////////
```

```
                    HSMC_CLKIN_n,
                    HSMC_CLKIN_p,
                    HSMC_CLKOUT_n,
                    HSMC_CLKOUT_p,
                    HSMC_CLK_IN0,
                    HSMC_CLK_OUT0,
                    HSMC_D,

`ifdef ENABLE_HSMC_XCVR

                    HSMC_GXB_RX_p,
                    HSMC_GXB_TX_p,
                    HSMC_REF_CLK_p,
`endif
                    HSMC_RX_n,
                    HSMC_RX_p,
                    HSMC_SCL,
                    HSMC_SDA,
                    HSMC_TX_n,
                    HSMC_TX_p,

                    /////////IRDA/////////
                    IRDA_RXD,

                    /////////KEY/////////
                    KEY,

                    /////////LED/////////
                    LED,

                    /////////OSC/////////
                    OSC_50_B3B,
                    OSC_50_B4A,
                    OSC_50_B5B,
                    OSC_50_B8A,

                    /////////PCIE/////////
                    PCIE_PERST_n,
                    PCIE_WAKE_n,

                    /////////RESET/////////
                    RESET_n,

                    /////////SI5338/////////
                    SI5338_SCL,
                    SI5338_SDA,
                    /////////SW/////////
                    SW,

                    /////////TEMP/////////
                    TEMP_CS_n,
                    TEMP_DIN,
                    TEMP_DOUT,
                    TEMP_SCLK,

                    /////////USB/////////
                    USB_B2_CLK,
                    USB_B2_DATA,
                    USB_EMPTY,
                    USB_FULL,
                    USB_OE_n,
                    USB_RD_n,
                    USB_RESET_n,
                    USB_SCL,
                    USB_SDA,
                    USB_WR_n,

                    /////////VGA/////////
                    VGA_B,
                    VGA_BLANK_n,
                    VGA_CLK,
                    VGA_G,
                    VGA_HS,
                    VGA_R,
                    VGA_SYNC_n,
                    VGA_VS,
                    //////////hps/////////
                    memory_mem_a,
                    memory_mem_ba,
                    memory_mem_ck,
                    memory_mem_ck_n,
                    memory_mem_cke,
                    memory_mem_cs_n,
                    memory_mem_ras_n,
                    memory_mem_cas_n,
                    memory_mem_we_n,
                    memory_mem_reset_n,
                    memory_mem_dq,
                    memory_mem_dqs,
                    memory_mem_dqs_n,
                    memory_mem_odt,
                    memory_mem_dm,
                    memory_oct_rzqin,
                    hps_io_hps_io_emac1_inst_TX_CLK,
                    hps_io_hps_io_emac1_inst_TXD0,
                    hps_io_hps_io_emac1_inst_TXD1,
                    hps_io_hps_io_emac1_inst_TXD2,
                    hps_io_hps_io_emac1_inst_TXD3,
                    hps_io_hps_io_emac1_inst_RXD0,
                    hps_io_hps_io_emac1_inst_MDIO,
                    hps_io_hps_io_emac1_inst_MDC,
                    hps_io_hps_io_emac1_inst_RX_CTL,
                    hps_io_hps_io_emac1_inst_TX_CTL,
                    hps_io_hps_io_emac1_inst_RX_CLK,
                    hps_io_hps_io_emac1_inst_RXD1,
                    hps_io_hps_io_emac1_inst_RXD2,
                    hps_io_hps_io_emac1_inst_RXD3,
                    hps_io_hps_io_qspi_inst_IO0,
                    hps_io_hps_io_qspi_inst_IO1,
                    hps_io_hps_io_qspi_inst_IO2,
                    hps_io_hps_io_qspi_inst_IO3,
                    hps_io_hps_io_qspi_inst_SS0,
                    hps_io_hps_io_qspi_inst_CLK,
                    hps_io_hps_io_sdio_inst_CMD,
                    hps_io_hps_io_sdio_inst_D0,
                    hps_io_hps_io_sdio_inst_D1,
                    hps_io_hps_io_sdio_inst_CLK,
                    hps_io_hps_io_sdio_inst_D2,
                    hps_io_hps_io_sdio_inst_D3,
                    hps_io_hps_io_usb1_inst_D0,
                    hps_io_hps_io_usb1_inst_D1,
                    hps_io_hps_io_usb1_inst_D2,
                    hps_io_hps_io_usb1_inst_D3,
                    hps_io_hps_io_usb1_inst_D4,
                    hps_io_hps_io_usb1_inst_D5,
                    hps_io_hps_io_usb1_inst_D6,
                    hps_io_hps_io_usb1_inst_D7,
                    hps_io_hps_io_usb1_inst_CLK,
                    hps_io_hps_io_usb1_inst_STP,
                    hps_io_hps_io_usb1_inst_DIR,
                    hps_io_hps_io_usb1_inst_NXT,
                    hps_io_hps_io_spim0_inst_CLK,
                    hps_io_hps_io_spim0_inst_MOSI,
                    hps_io_hps_io_spim0_inst_MISO,
                    hps_io_hps_io_spim0_inst_SS0,
                    hps_io_hps_io_spim1_inst_CLK,
                    hps_io_hps_io_spim1_inst_MOSI,
                    hps_io_hps_io_spim1_inst_MISO,
                    hps_io_hps_io_spim1_inst_SS0,
                    hps_io_hps_io_uart0_inst_RX,
                        hps_io_hps_io_uart0_inst_TX,
                        hps_io_hps_io_i2c1_inst_SDA,
                        hps_io_hps_io_i2c1_inst_SCL,
                        hps_io_hps_io_gpio_inst_GPIO00
                        );

  //=======================================================
  //  PORT declarations
  //=======================================================

  ///////// AUD /////////
  input                         AUD_ADCDAT;
  inout                         AUD_ADCLRCK;
  inout                         AUD_BCLK;
  output                         AUD_DACDAT;
  inout                         AUD_DACLRCK;
  output                         AUD_I2C_SCLK;
  inout                         AUD_I2C_SDAT;
  output                         AUD_MUTE;
  output                         AUD_XCK;

`ifdef ENABLE_DDR3
  ///////// DDR3 /////////
  output [14:0]                           DDR3_A;
  output [2:0]                       DDR3_BA;
  output                         DDR3_CAS_n;
  output                         DDR3_CKE;
  output                         DDR3_CK_n;
  output                         DDR3_CK_p;
  output                         DDR3_CS_n;
  output [3:0]                       DDR3_DM;
  inout [31:0]                       DDR3_DQ;
  inout [3:0]                             DDR3_DQS_n;
  inout [3:0]                             DDR3_DQS_p;
  output                         DDR3_ODT;
  output                         DDR3_RAS_n;
  output                         DDR3_RESET_n;
  input                         DDR3_RZQ;
  output                         DDR3_WE_n;
`endif /*ENABLE_DDR3*/

  ///////// FAN /////////
  output                         FAN_CTRL;

`ifdef ENABLE_HPS
  ///////// HPS /////////
  input                         HPS_CLOCK_25;
  input                           HPS_CLOCK_50;
  input                           HPS_CONV_USB_n;
  output [14:0]                               HPS_DDR3_A;
  output [2:0]                         HPS_DDR3_BA;
  output                           HPS_DDR3_CAS_n;
  output                           HPS_DDR3_CKE;
  output                           HPS_DDR3_CK_n;
  output                           HPS_DDR3_CK_p;
  output                           HPS_DDR3_CS_n;
  output [3:0]                         HPS_DDR3_DM;
  inout [31:0]                         HPS_DDR3_DQ;
  inout [3:0]                                 HPS_DDR3_DQS_n;
  inout [3:0]                                 HPS_DDR3_DQS_p;
  output                           HPS_DDR3_ODT;
  output                           HPS_DDR3_RAS_n;
  output                           HPS_DDR3_RESET_n;
  input                           HPS_DDR3_RZQ;
  output                           HPS_DDR3_WE_n;
  input                           HPS_ENET_GTX_CLK;
  input                           HPS_ENET_INT_n;
  output                           HPS_ENET_MDC;
  inout                           HPS_ENET_MDIO;
  output                           HPS_ENET_RESET_n;
  input                           HPS_ENET_RX_CLK;
  input [3:0]                                 HPS_ENET_RX_DATA;
  input                           HPS_ENET_RX_DV;
  output [3:0]                         HPS_ENET_TX_DATA;
  output                           HPS_ENET_TX_EN;
  inout [3:0]                                 HPS_FLASH_DATA;
  output                           HPS_FLASH_DCLK;
  output                           HPS_FLASH_NCSO;
  input                           HPS_GSENSOR_INT;
  inout                           HPS_I2C_CLK;
  inout                           HPS_I2C_SDA;
  inout [3:0]                                 HPS_KEY;
  output                           HPS_LCM_D_C;
  output                           HPS_LCM_RST_N;
  input                           HPS_LCM_SPIM_CLK;
  inout                           HPS_LCM_SPIM_MISO;
  output                           HPS_LCM_SPIM_MOSI;
  output                           HPS_LCM_SPIM_SS;
  output [3:0]                         HPS_LED;
  inout                           HPS_LTC_GPIO;
  input                           HPS_RESET_n;
  output                           HPS_SD_CLK;
  inout                           HPS_SD_CMD;
  inout [3:0]                                 HPS_SD_DATA;
```

```
  output                         HPS_SPIM_CLK;
  input                         HPS_SPIM_MISO;
  output                         HPS_SPIM_MOSI;
  output                         HPS_SPIM_SS;
  input [3:0]                              HPS_SW;
  input                         HPS_UART_RX;
  output                         HPS_UART_TX;
  input                         HPS_USB_CLKOUT;
  inout [7:0]                              HPS_USB_DATA;
  input                         HPS_USB_DIR;
  input                         HPS_USB_NXT;
  output                         HPS_USB_RESET_PHY;
  output                         HPS_USB_STP;
  input                         HPS_WARM_RST_n;
`endif /*ENABLE_HPS*/

  ///////// HSMC /////////
  input [2:1]                              HSMC_CLKIN_n;
  input [2:1]                              HSMC_CLKIN_p;
  output [2:1]                       HSMC_CLKOUT_n;
  output [2:1]                       HSMC_CLKOUT_p;
  input                         HSMC_CLK_IN0;
  output                         HSMC_CLK_OUT0;
  inout [3:0]                              HSMC_D;
`ifdef ENABLE_HSMC_XCVR
  input [7:0]                              HSMC_GXB_RX_p;
  output [7:0]                       HSMC_GXB_TX_p;
  input                         HSMC_REF_CLK_p;
`endif
  inout [16:0]                       HSMC_RX_n;
  inout [16:0]                       HSMC_RX_p;
  output                         HSMC_SCL;
  inout                         HSMC_SDA;
  inout [16:0]                       HSMC_TX_n;
  inout [16:0]                       HSMC_TX_p;

  ///////// IRDA /////////
  input                         IRDA_RXD;

  ///////// KEY /////////
  input [3:0]                              KEY;

  ///////// LED /////////
  output [3:0]                       LED;

  ///////// OSC /////////
  input                         OSC_50_B3B;
  input                          OSC_50_B4A;
  input                          OSC_50_B5B;
  input                          OSC_50_B8A;

  ///////// PCIE /////////
  input                          PCIE_PERST_n;
  input                          PCIE_WAKE_n;

  ///////// RESET /////////
  input                          RESET_n;

  ///////// SI5338 /////////
  inout                          SI5338_SCL;
  inout                          SI5338_SDA;

  ///////// SW /////////
  input [3:0]                               SW;

  ///////// TEMP /////////
  output                          TEMP_CS_n;
  output                          TEMP_DIN;
  input                          TEMP_DOUT;
  output                          TEMP_SCLK;

  ///////// USB /////////
  input                          USB_B2_CLK;
  inout [7:0]                               USB_B2_DATA;
  output                          USB_EMPTY;
  output                          USB_FULL;
  input                          USB_OE_n;
  input                          USB_RD_n;
  input                          USB_RESET_n;
  inout                          USB_SCL;
  inout                          USB_SDA;
  input                          USB_WR_n;

  ///////// VGA /////////
  output [7:0]                         VGA_B;
  output                          VGA_BLANK_n;
  output                          VGA_CLK;
  output [7:0]                         VGA_G;
  output                          VGA_HS;
  output [7:0]                         VGA_R;
  output                          VGA_SYNC_n;
  output                          VGA_VS;

  /////////hps pin///////
```

```
output wire [14:0]                      memory_mem_a;
output wire [2:0]                       memory_mem_ba;
output wire                             memory_mem_ck;
output wire                             memory_mem_ck_n;
output wire                             memory_mem_cke;
output wire                             memory_mem_cs_n;
output wire                             memory_mem_ras_n;
output wire                             memory_mem_cas_n;
output wire                             memory_mem_we_n;
output wire                             memory_mem_reset_n;
inout  wire [31:0]                      memory_mem_dq;
inout  wire [3:0]                       memory_mem_dqs;
inout  wire [3:0]                       memory_mem_dqs_n;
output wire                             memory_mem_odt;
output wire [3:0]                       memory_mem_dm;
input  wire                             memory_oct_rzqin;
output wire                             hps_io_hps_io_emac1_inst_TX_CLK;
output wire                             hps_io_hps_io_emac1_inst_TXD0;
output wire                             hps_io_hps_io_emac1_inst_TXD1;
output wire                             hps_io_hps_io_emac1_inst_TXD2;
output wire                             hps_io_hps_io_emac1_inst_TXD3;
input  wire                             hps_io_hps_io_emac1_inst_RXD0;
inout  wire                             hps_io_hps_io_emac1_inst_MDIO;
output wire                             hps_io_hps_io_emac1_inst_MDC;
input  wire                             hps_io_hps_io_emac1_inst_RX_CTL;
output wire                             hps_io_hps_io_emac1_inst_TX_CTL;
input  wire                             hps_io_hps_io_emac1_inst_RX_CLK;
input  wire                             hps_io_hps_io_emac1_inst_RXD1;
input  wire                             hps_io_hps_io_emac1_inst_RXD2;
input  wire                             hps_io_hps_io_emac1_inst_RXD3;
inout  wire                             hps_io_hps_io_qspi_inst_IO0;
inout  wire                             hps_io_hps_io_qspi_inst_IO1;
inout  wire                             hps_io_hps_io_qspi_inst_IO2;
inout  wire                             hps_io_hps_io_qspi_inst_IO3;
output wire                             hps_io_hps_io_qspi_inst_SS0;
output wire                             hps_io_hps_io_qspi_inst_CLK;
inout  wire                             hps_io_hps_io_sdio_inst_CMD;
inout  wire                             hps_io_hps_io_sdio_inst_D0;
inout  wire                             hps_io_hps_io_sdio_inst_D1;
output wire                             hps_io_hps_io_sdio_inst_CLK;
inout  wire                             hps_io_hps_io_sdio_inst_D2;
inout  wire                             hps_io_hps_io_sdio_inst_D3;
inout  wire                             hps_io_hps_io_usb1_inst_D0;
inout  wire                             hps_io_hps_io_usb1_inst_D1;
inout  wire                             hps_io_hps_io_usb1_inst_D2;
inout  wire                             hps_io_hps_io_usb1_inst_D3;
inout  wire                             hps_io_hps_io_usb1_inst_D4;
inout  wire                                        hps_io_hps_io_usb1_inst_D5;
inout  wire                                        hps_io_hps_io_usb1_inst_D6;
inout  wire                                        hps_io_hps_io_usb1_inst_D7;
input  wire                                        hps_io_hps_io_usb1_inst_CLK;
output wire                                        hps_io_hps_io_usb1_inst_STP;
input  wire                                        hps_io_hps_io_usb1_inst_DIR;
input  wire                                        hps_io_hps_io_usb1_inst_NXT;
output wire                                        hps_io_hps_io_spim0_inst_CLK;
output wire                                        hps_io_hps_io_spim0_inst_MOSI;
input  wire                                        hps_io_hps_io_spim0_inst_MISO;
output wire                                        hps_io_hps_io_spim0_inst_SS0;
output wire                                        hps_io_hps_io_spim1_inst_CLK;
output wire                                        hps_io_hps_io_spim1_inst_MOSI;
input  wire                                        hps_io_hps_io_spim1_inst_MISO;
output wire                                        hps_io_hps_io_spim1_inst_SS0;
input  wire                                        hps_io_hps_io_uart0_inst_RX;
output wire                                        hps_io_hps_io_uart0_inst_TX;
inout  wire                                        hps_io_hps_io_i2c1_inst_SDA;
inout  wire                                        hps_io_hps_io_i2c1_inst_SCL;
inout  wire                                        hps_io_hps_io_gpio_inst_GPIO00;
//=======================================================
//  REG/WIRE declarations
//=======================================================

//      For Audio CODEC
wire                                  AUD_CTRL_CLK;         //        For Audio Controller

reg [31:0]                                  Cont;
wire                                  VGA_CTRL_CLK;
wire [9:0]                                  mVGA_R;
wire [9:0]                                  mVGA_G;
wire [9:0]                                  mVGA_B;
wire [19:0]                                 mVGA_ADDR;
wire                                  DLY_RST;

//      For VGA Controller
wire                                  mVGA_CLK;
wire [9:0]                                  mRed;
wire [9:0]                                  mGreen;
wire [9:0]                                  mBlue;
wire                                  VGA_Read;        //        VGA data request

wire [9:0]                                  recon_VGA_R;
wire [9:0]                                  recon_VGA_G;
wire [9:0]                                  recon_VGA_B;

//      For Down Sample
```

```
wire [3:0]                                   Remain;
wire [9:0]                                   Quotient;

wire                                 AUD_MUTE;

// Drive the LEDs with the switches
assign LED = SW;

// Make the FPGA reset cause an HPS reset
reg [19:0]                                   hps_reset_counter = 20'h0;
reg                                          hps_fpga_reset_n = 0;

always @(posedge OSC_50_B4A) begin
  if (hps_reset_counter == 20'h ffffff) hps_fpga_reset_n <= 1;
  hps_reset_counter <= hps_reset_counter + 1;
end

lab3 u0 (
      .clk_clk                        (OSC_50_B4A),                  //             clk.clk
      .reset_reset_n                  (hps_fpga_reset_n),            //             reset.reset_n
      .memory_mem_a                   (memory_mem_a),                //             memory.mem_a
      .memory_mem_ba                  (memory_mem_ba),               //             .mem_ba
      .memory_mem_ck                  (memory_mem_ck),               //             .mem_ck
      .memory_mem_ck_n                (memory_mem_ck_n),             //             .mem_ck_n
      .memory_mem_cke                 (memory_mem_cke),              //             .mem_cke
      .memory_mem_cs_n                (memory_mem_cs_n),             //             .mem_cs_n
      .memory_mem_ras_n               (memory_mem_ras_n),            //             .mem_ras_n
      .memory_mem_cas_n               (memory_mem_cas_n),            //             .mem_cas_n
      .memory_mem_we_n                (memory_mem_we_n),             //             .mem_we_n
      .memory_mem_reset_n             (memory_mem_reset_n),          //             .mem_reset_n
      .memory_mem_dq                  (memory_mem_dq),               //             .mem_dq
      .memory_mem_dqs                 (memory_mem_dqs),              //             .mem_dqs
      .memory_mem_dqs_n               (memory_mem_dqs_n),            //             .mem_dqs_n
      .memory_mem_odt                 (memory_mem_odt),              //             .mem_odt
      .memory_mem_dm                  (memory_mem_dm),               //             .mem_dm
      .memory_oct_rzqin               (memory_oct_rzqin),            //             .oct_rzqin
      .hps_io_hps_io_emac1_inst_TX_CLK               (hps_io_hps_io_emac1_inst_TX_CLK), //
                          .hps_0_hps_io.hps_io_emac1_inst_TX_CLK
      .hps_io_hps_io_emac1_inst_TXD0           (hps_io_hps_io_emac1_inst_TXD0),
//          .hps_io_emac1_inst_TXD0
      .hps_io_hps_io_emac1_inst_TXD1           (hps_io_hps_io_emac1_inst_TXD1),
//          .hps_io_emac1_inst_TXD1
      .hps_io_hps_io_emac1_inst_TXD2           (hps_io_hps_io_emac1_inst_TXD2),
//          .hps_io_emac1_inst_TXD2
      .hps_io_hps_io_emac1_inst_TXD3           (hps_io_hps_io_emac1_inst_TXD3),
//          .hps_io_emac1_inst_TXD3
```

```
		.hps_io_hps_io_emac1_inst_RXD0			(hps_io_hps_io_emac1_inst_RXD0),
//		.hps_io_emac1_inst_RXD0
		.hps_io_hps_io_emac1_inst_MDIO				(hps_io_hps_io_emac1_inst_MDIO),
//		.hps_io_emac1_inst_MDIO
		.hps_io_hps_io_emac1_inst_MDC			(hps_io_hps_io_emac1_inst_MDC),
//		.hps_io_emac1_inst_MDC
		.hps_io_hps_io_emac1_inst_RX_CTL				(hps_io_hps_io_emac1_inst_RX_CTL),
//		.hps_io_emac1_inst_RX_CTL
		.hps_io_hps_io_emac1_inst_TX_CTL				(hps_io_hps_io_emac1_inst_TX_CTL),
//		.hps_io_emac1_inst_TX_CTL
		.hps_io_hps_io_emac1_inst_RX_CLK				(hps_io_hps_io_emac1_inst_RX_CLK),
//		.hps_io_emac1_inst_RX_CLK
		.hps_io_hps_io_emac1_inst_RXD1			(hps_io_hps_io_emac1_inst_RXD1),
//		.hps_io_emac1_inst_RXD1
		.hps_io_hps_io_emac1_inst_RXD2			(hps_io_hps_io_emac1_inst_RXD2),
//		.hps_io_emac1_inst_RXD2
		.hps_io_hps_io_emac1_inst_RXD3			(hps_io_hps_io_emac1_inst_RXD3),
//		.hps_io_emac1_inst_RXD3
		.hps_io_hps_io_qspi_inst_IO0			(hps_io_hps_io_qspi_inst_IO0),
//		.hps_io_qspi_inst_IO0
		.hps_io_hps_io_qspi_inst_IO1			(hps_io_hps_io_qspi_inst_IO1),
//		.hps_io_qspi_inst_IO1
		.hps_io_hps_io_qspi_inst_IO2			(hps_io_hps_io_qspi_inst_IO2),
//		.hps_io_qspi_inst_IO2
		.hps_io_hps_io_qspi_inst_IO3			(hps_io_hps_io_qspi_inst_IO3),
//		.hps_io_qspi_inst_IO3
		.hps_io_hps_io_qspi_inst_SS0			(hps_io_hps_io_qspi_inst_SS0),
//		.hps_io_qspi_inst_SS0
		.hps_io_hps_io_qspi_inst_CLK			(hps_io_hps_io_qspi_inst_CLK),
//		.hps_io_qspi_inst_CLK
		.hps_io_hps_io_sdio_inst_CMD			(hps_io_hps_io_sdio_inst_CMD),
//		.hps_io_sdio_inst_CMD
		.hps_io_hps_io_sdio_inst_D0			(hps_io_hps_io_sdio_inst_D0),
//		.hps_io_sdio_inst_D0
		.hps_io_hps_io_sdio_inst_D1			(hps_io_hps_io_sdio_inst_D1),
//		.hps_io_sdio_inst_D1
		.hps_io_hps_io_sdio_inst_CLK			(hps_io_hps_io_sdio_inst_CLK),
//		.hps_io_sdio_inst_CLK
		.hps_io_hps_io_sdio_inst_D2			(hps_io_hps_io_sdio_inst_D2),
//		.hps_io_sdio_inst_D2
		.hps_io_hps_io_sdio_inst_D3			(hps_io_hps_io_sdio_inst_D3),
//		.hps_io_sdio_inst_D3
		.hps_io_hps_io_usb1_inst_D0			(hps_io_hps_io_usb1_inst_D0),
//		.hps_io_usb1_inst_D0
		.hps_io_hps_io_usb1_inst_D1			(hps_io_hps_io_usb1_inst_D1),
//		.hps_io_usb1_inst_D1
```

```
        .hps_io_hps_io_usb1_inst_D2                 (hps_io_hps_io_usb1_inst_D2),
//          .hps_io_usb1_inst_D2
        .hps_io_hps_io_usb1_inst_D3                 (hps_io_hps_io_usb1_inst_D3),
//          .hps_io_usb1_inst_D3
        .hps_io_hps_io_usb1_inst_D4                 (hps_io_hps_io_usb1_inst_D4),
//          .hps_io_usb1_inst_D4
        .hps_io_hps_io_usb1_inst_D5                 (hps_io_hps_io_usb1_inst_D5),
//          .hps_io_usb1_inst_D5
        .hps_io_hps_io_usb1_inst_D6                 (hps_io_hps_io_usb1_inst_D6),
//          .hps_io_usb1_inst_D6
        .hps_io_hps_io_usb1_inst_D7                 (hps_io_hps_io_usb1_inst_D7),
//          .hps_io_usb1_inst_D7
        .hps_io_hps_io_usb1_inst_CLK                (hps_io_hps_io_usb1_inst_CLK),
//          .hps_io_usb1_inst_CLK
        .hps_io_hps_io_usb1_inst_STP                (hps_io_hps_io_usb1_inst_STP),
//          .hps_io_usb1_inst_STP
        .hps_io_hps_io_usb1_inst_DIR                (hps_io_hps_io_usb1_inst_DIR),
//          .hps_io_usb1_inst_DIR
        .hps_io_hps_io_usb1_inst_NXT                (hps_io_hps_io_usb1_inst_NXT),
//          .hps_io_usb1_inst_NXT
        .hps_io_hps_io_spim0_inst_CLK               (hps_io_hps_io_spim0_inst_CLK),
//          .hps_io_spim0_inst_CLK
        .hps_io_hps_io_spim0_inst_MOSI              (hps_io_hps_io_spim0_inst_MOSI),
//          .hps_io_spim0_inst_MOSI
        .hps_io_hps_io_spim0_inst_MISO              (hps_io_hps_io_spim0_inst_MISO),
//          .hps_io_spim0_inst_MISO
        .hps_io_hps_io_spim0_inst_SS0               (hps_io_hps_io_spim0_inst_SS0),
//          .hps_io_spim0_inst_SS0
        .hps_io_hps_io_spim1_inst_CLK               (hps_io_hps_io_spim1_inst_CLK),
//          .hps_io_spim1_inst_CLK
        .hps_io_hps_io_spim1_inst_MOSI              (hps_io_hps_io_spim1_inst_MOSI),
//          .hps_io_spim1_inst_MOSI
        .hps_io_hps_io_spim1_inst_MISO              (hps_io_hps_io_spim1_inst_MISO),
//          .hps_io_spim1_inst_MISO
        .hps_io_hps_io_spim1_inst_SS0               (hps_io_hps_io_spim1_inst_SS0),
//          .hps_io_spim1_inst_SS0
        .hps_io_hps_io_uart0_inst_RX                (hps_io_hps_io_uart0_inst_RX),
//          .hps_io_uart0_inst_RX
        .hps_io_hps_io_uart0_inst_TX                (hps_io_hps_io_uart0_inst_TX),
//          .hps_io_uart0_inst_TX
        .hps_io_hps_io_i2c1_inst_SDA                (hps_io_hps_io_i2c1_inst_SDA),
//          .hps_io_i2c1_inst_SDA
        .hps_io_hps_io_i2c1_inst_SCL                (hps_io_hps_io_i2c1_inst_SCL),
//          .hps_io_i2c1_inst_SCL
              .piano_VGA_R (VGA_R),
                              .piano_VGA_G (VGA_G),
                              .piano_VGA_B (VGA_B),
                              .piano_VGA_CLK (VGA_CLK),
                              .piano_VGA_HS (VGA_HS),
                              .piano_VGA_VS (VGA_VS),
                              .piano_VGA_BLANK_n (VGA_BLANK_n),
                        .piano_VGA_SYNC_n (VGA_SYNC_n),
                              .piano_SW (SW),
                              .piano_KEY (KEY),
                              .piano_AUD_ADCLRCK (AUD_ADCLRCK),
                              .piano_AUD_ADCDAT (AUD_ADCDAT),
                              .piano_AUD_DACLRCK (AUD_DACLRCK),
                              .piano_AUD_DACDAT (AUD_DACDAT),
                              .piano_AUD_XCK (AUD_XCK),
                              .piano_AUD_BCLK (AUD_BCLK),
                              .piano_AUD_I2C_SCLK (AUD_I2C_SCLK),
                              .piano_AUD_I2C_SDAT (AUD_I2C_SDAT),
                              .piano_AUD_MUTE (AUD_MUTE)
                );

endmodule
```

Audio

I2C CONFIGURATION

```
module i2c_av_config (
   input clk,
   input reset,

   output i2c_sclk,
   inout  i2c_sdat
);

reg [23:0] i2c_data;
reg [15:0] lut_data;
reg [3:0]  lut_index = 4'd0;

parameter LAST_INDEX = 4'ha;

reg  i2c_start = 1'b0;
wire i2c_done;
wire i2c_ack;

i2c_controller control (
   .clk (clk),
   .i2c_sclk (i2c_sclk),
   .i2c_sdat (i2c_sdat),
   .i2c_data (i2c_data),
   .start (i2c_start),
   .done (i2c_done),
   .ack (i2c_ack)
);

always @(*) begin
   case (lut_index)
       4'h0: lut_data <= 16'h0c10; // power on everything except out
       4'h1: lut_data <= 16'h0017; // left input
       4'h2: lut_data <= 16'h0217; // right input
       4'h3: lut_data <= 16'h0479; // left output
       4'h4: lut_data <= 16'h0679; // right output
       4'h5: lut_data <= 16'h08d4; // analog path
       4'h6: lut_data <= 16'h0a04; // digital path
       4'h7: lut_data <= 16'h0e01; // digital IF
       4'h8: lut_data <= 16'h1020; // sampling rate
       4'h9: lut_data <= 16'h0c00; // power on everything
       4'ha: lut_data <= 16'h1201; // activate
       default: lut_data <= 16'h0000;
   endcase
end

reg [1:0] control_state = 2'b00;

always @(posedge clk) begin
   if (reset) begin
       lut_index <= 4'd0;
       i2c_start <= 1'b0;
       control_state <= 2'b00;
   end else begin
       case (control_state)
           2'b00: begin
               i2c_start <= 1'b1;
               i2c_data <= {8'h34, lut_data};
               control_state <= 2'b01;
           end
           2'b01: begin
               i2c_start <= 1'b0;
               control_state <= 2'b10;
           end
           2'b10: if (i2c_done) begin
               if (i2c_ack) begin
                   if (lut_index == LAST_INDEX)
                       control_state <= 2'b11;
                   else begin
                       lut_index <= lut_index + 1'b1;
                       control_state <= 2'b00;
                   end
               end else
                   control_state <= 2'b00;
           end
       endcase
   end
end

endmodule
```

I2C CONTROLLER

```
module i2c_controller (
    input  clk,

    output i2c_sclk,
    inout  i2c_sdat,

    input  start,
    output done,
    output ack,

    input [23:0] i2c_data
);

reg [23:0] data;

reg [4:0] stage;
reg [6:0] sclk_divider;
reg clock_en = 1'b0;

// don't toggle the clock unless we're sending data
// clock will also be kept high when sending START and STOP symbols
assign i2c_sclk = (!clock_en) || sclk_divider[6];
wire midlow = (sclk_divider == 7'h1f);

reg sdat = 1'b1;
// rely on pull-up resistor to set SDAT high
assign i2c_sdat = (sdat) ? 1'bz : 1'b0;

reg [2:0] acks;

parameter LAST_STAGE = 5'd29;

assign ack = (acks == 3'b000);
assign done = (stage == LAST_STAGE);
```

```
always @(posedge clk) begin
   if (start) begin
      sclk_divider <= 7'd0;
      stage <= 5'd0;
      clock_en = 1'b0;
      sdat <= 1'b1;
      acks <= 3'b111;
      data <= i2c_data;
   end else begin
      if (sclk_divider == 7'd127) begin
         sclk_divider <= 7'd0;

         if (stage != LAST_STAGE)
            stage <= stage + 1'b1;

         case (stage)
            // after start
            5'd0:  clock_en <= 1'b1;
            // receive acks
            5'd9:  acks[0] <= i2c_sdat;
            5'd18: acks[1] <= i2c_sdat;
            5'd27: acks[2] <= i2c_sdat;
            // before stop
            5'd28: clock_en <= 1'b0;
         endcase
      end else
         sclk_divider <= sclk_divider + 1'b1;

      if (midlow) begin
         case (stage)
            // start
            5'd0:  sdat <= 1'b0;
            // byte 1
            5'd1:  sdat <= data[23];
            5'd2:  sdat <= data[22];
            5'd3:  sdat <= data[21];
            5'd4:  sdat <= data[20];
            5'd5:  sdat <= data[19];
            5'd6:  sdat <= data[18];
            5'd7:  sdat <= data[17];
            5'd8:  sdat <= data[16];
            // ack 1
            5'd9:  sdat <= 1'b1;
            // byte 2
            5'd10: sdat <= data[15];
            5'd11: sdat <= data[14];
```

```
                5'd12: sdat <= data[13];
                5'd13: sdat <= data[12];
                5'd14: sdat <= data[11];
                5'd15: sdat <= data[10];
                5'd16: sdat <= data[9];
                5'd17: sdat <= data[8];
                // ack 2
                5'd18: sdat <= 1'b1;
                // byte 3
                5'd19: sdat <= data[7];
                5'd20: sdat <= data[6];
                5'd21: sdat <= data[5];
                5'd22: sdat <= data[4];
                5'd23: sdat <= data[3];
                5'd24: sdat <= data[2];
                5'd25: sdat <= data[1];
                5'd26: sdat <= data[0];
                // ack 3
                5'd27: sdat <= 1'b1;
                // stop
                5'd28: sdat <= 1'b0;
                5'd29: sdat <= 1'b1;
            endcase
        end
    end
end

endmodule
```

AUDIO CODEC

```
module audio_codec (
    input  clk,
    input  reset,
    output [1:0]  sample_end,
    output [1:0]  sample_req,
    input  [15:0] audio_output,
    output [15:0] audio_input,
    // 1 - left, 0 - right
    input  [1:0] channel_sel,

    output AUD_ADCLRCK,
    input AUD_ADCDAT,
    output AUD_DACLRCK,
    output AUD_DACDAT,
    output AUD_BCLK
);
```

```
reg [7:0] lrck_divider;
reg [1:0] bclk_divider;

reg [15:0] shift_out;
reg [15:0] shift_temp;
reg [15:0] shift_in;

wire lrck = !lrck_divider[7];

assign AUD_ADCLRCK = lrck;
assign AUD_DACLRCK = lrck;
assign AUD_BCLK = bclk_divider[1];
assign AUD_DACDAT = shift_out[15];

always @(posedge clk) begin
    if (reset) begin
        lrck_divider <= 8'hff;
        bclk_divider <= 2'b11;
    end else begin
        lrck_divider <= lrck_divider + 1'b1;
        bclk_divider <= bclk_divider + 1'b1;
    end
end

assign sample_end[1] = (lrck_divider == 8'h40);
assign sample_end[0] = (lrck_divider == 8'hc0);
assign audio_input = shift_in;
assign sample_req[1] = (lrck_divider == 8'hfe);
assign sample_req[0] = (lrck_divider == 8'h7e);

wire clr_lrck = (lrck_divider == 8'h7f);
wire set_lrck = (lrck_divider == 8'hff);
// high right after bclk is set
wire set_bclk = (bclk_divider == 2'b10 && !lrck_divider[6]);
// high right before bclk is cleared
wire clr_bclk = (bclk_divider == 2'b11 && !lrck_divider[6]);

always @(posedge clk) begin
    if (reset) begin
        shift_out <= 16'h0;
        shift_in <= 16'h0;
        shift_in <= 16'h0;
    end else if (set_lrck || clr_lrck) begin
        // check if current channel is selected
        if (channel_sel[set_lrck]) begin
            shift_out <= audio_output;
            shift_temp <= audio_output;
            shift_in <= 16'h0;
        // repeat the sample from the other channel if not
        end else shift_out <= shift_temp;
    end else if (set_bclk == 1) begin
        // only read in if channel is selected
        if (channel_sel[lrck])
            shift_in <= {shift_in[14:0], AUD_ADCDAT};
    end else if (clr_bclk == 1) begin
        shift_out <= {shift_out[14:0], 1'b0};
    end
end

endmodule
```

KARPLUS-STRONG ALGORITHM

```
(control[3] == 0) ? ((control[2] == 0) ? 3'b000 : 3'b010): ((control[2] == 0) ? 3'b011 : 3'b100);
//Karplus-strong algorithm module to synthesis piano notes
//Refer to 5760 DSP example code of guitar string pluck synthesis:
//https://instruct1.cit.cornell.edu/courses/ece576/DE2/fpgaDSP.html
module karplus_note (clock50, audiolrclk, reset, press, audio_output, audio_input, control);

input clock50;  //clock as reference
input audiolrclk;  //sample frequency
input reset;     //reset signal
input [16:0] press;   //represent which keys are pressed
output[15:0] audio_output;  //output audio signal
input [15:0] audio_input;
input [3:0] control;

wire [16:0] combination;
reg [16:0] combination_last;
assign combination = press;

reg [17:0] Out;  //middle string
reg [17:0] OutS,OutH;    //lower string and higher string
reg [17:0] OutSum;   //the sum of three strings
assign audio_output = OutSum[17:2];   //take the higher 15 bits to output

reg [10:0] note;   //define the length of the middle string shiftregister
reg [10:0] noteS;   //define the length of the lower string shiftregister
reg [10:0] noteH;                    //define the length of the higher string shiftregister

reg pluck ;   //pluck the string, and counts for three strings
reg last_pluck;
reg [11:0] pluck_count,pluck_countS,pluck_countH;
```

```
// state variable 0=reset, 1=readinput,
// 2=readoutput, 3=writeinput, 4=write 5=updatepointers,
// 9=stop
reg [2:0] state ;
reg last_clk ; //oneshot gen

wire [17:0] gain ;   // constant for gain

//pointers into the shift register
//4096 at 48kHz imples 12 Hz
reg [11:0] ptr_in, ptr_out,ptr_inS,ptr_outS,ptr_inH,ptr_outH;

//memory control
reg we,weS,weH; //write enable--active high
wire [17:0] sr_data,sr_dataS,sr_dataH;
reg [17:0]  write_data,write_dataS,write_dataH;
reg [11:0] addr_reg,addr_regS,addr_regH;

//data registers for arithmetic
reg [17:0] in_data, out_data;
reg [17:0] in_dataS, out_dataS;
reg [17:0] in_dataH, out_dataH;
wire [17:0] new_out, new_outH;

//random number generator and lowpass filter
wire x_low_bit ;    // random number gen low-order bit
reg [30:0] x_rand ;   //  rand number
wire [17:0] new_lopass ;
reg [17:0]  lopass ;

wire [2:0] alpha;   //alpha that is used in filter
assign alpha = (control[3] == 0) ? 3'b100 : 3'b001;

// pluck control by combination
always @ (posedge clock50)
begin
        pluck <= (combination==combination_last)?((combination==17'd0)?1'b0:1'b1):0;
        combination_last<=combination;
end

//generate a random number at audio rate
// --AUD_DACLRCK toggles once per left/right pair
// --so it is the start signal for a random number update
// --at audio sample rate
//right-most bit for rand number shift regs
assign x_low_bit = x_rand[27] ^ x_rand[30];
```

```
// newsample = (1-alpha)*oldsample + (random+/-1)*alpha
// rearranging:
// newsample = oldsample + ((random+/-1)-oldsample)*alpha
// alpha is set from 1 to 1/128 using switches
// alpha==1 means no lopass at all. 1/128 loses almost all the input bits
assign new_lopass = lopass + ((( (x_low_bit)?18'h1_0000:18'h3_0000) - lopass)>>>alpha);

//your basic XOR random # gen
always @ (posedge audiolrclk)
begin
        if (reset)
        begin
                x_rand <= 31'h55555555;
                lopass <= 18'h0 ;
        end
        else begin
                x_rand <= {x_rand[29:0], x_low_bit} ;
                lopass <= new_lopass;
        end
end

//when user pushes a button transfer rand number to circular buffer
//treat each bit of rand register as +/-1, 18-bit, 2'comp
//when loading to circ buffer
//shift buffer, apply filter, update indexes
//once per audio clock tick
assign gain = 18'h0_7FF8 ;

//Run the state machine FAST so that it completes in one
//audio cycle
always @ (posedge clock50)
begin
        if (reset)
        begin
                ptr_out <= 12'h1 ; //output beginning of shift register
                ptr_outS <=12'h1;
                ptr_outH <=12'h1;
                ptr_in <= 12'h0 ;   //input beginning of shift register
                ptr_inS<=12'h0;
                ptr_inH <= 12'h0 ;
                we <= 1'h0 ;    //write enable signal
                weS<= 1'h0;
                weH<= 1'h0;
                state <= 3'd7; //turn off the update state machine
                last_clk <= 1'h1;

        end
        else begin

    //frequency(Hz) and the notes they correspond                                  if
(combination==17'b10000000000000000)//587-d2,16
      note <= 11'd79;
    else if (combination==17'b01000000000000000)//523-c2,15
      note <= 11'd89;
    else if (combination==17'b00100000000000000)//493-b1,14
      note <= 11'd94;
    else if (combination==17'b00010000000000000)//440-a1,13
      note <= 11'd106;
    else if (combination==17'b00001000000000000)//391-g1,12
      note <= 11'd119;
    else if (combination==17'b00000100000000000)//349-f1,11
      note <= 11'd133;
    else if (combination==17'b00000010000000000)//329-e1,10
      note <= 11'd142;
    else if (combination==17'b00000001000000000)//293-d1,9
      note <= 11'd160;
    else if (combination==17'b00000000100000000)//261-c1,8
      note <= 11'd179;
    else if (combination==17'b00000000010000000)//246-b0,7
      note <= 11'd188;
    else if (combination==17'b00000000001000000)//220-a0,6
      note <= 11'd210;
    else if (combination==17'b00000000000100000)//196-g0,5
      note <= 11'd238;
    else if (combination==17'b00000000000010000)//174-f0,4
      note <= 11'd261;
    else if (combination==17'b00000000000001000)//164-e0,3
            note<= 11'd280;
    else if (combination==17'b00000000000000100)//146-d0,2
      note <= 11'd315;
    else if (combination==17'b00000000000000010)//130.8-c0,1
      note<= 11'd350;

                case (state)

                        1:
                        begin
                                // set up read ptr_out data
                                addr_reg <= ptr_out;
                                addr_regS<= ptr_outS;
                                addr_regH <= ptr_outH;
                                we <= 1'h0;
                                weS<= 1'h0;
                                weH<= 1'h0;
                                state <= 3'd2;
                        end

                        2:
                        begin
                                //get ptr_out data
                                out_data <= sr_data;
                                out_dataS<= sr_dataS;
                                out_dataH <= sr_dataH;
                                // set up read ptr_in data
                                addr_reg <= ptr_in;
                                addr_regS<= ptr_inS;
                                addr_regH <= ptr_inH;
                                we <= 1'h0;
                                weS<= 1'h0;
                                weH<= 1'h0;
                                state <= 3'd3;
                        end

                        3:
                        begin
                                //get prt_in data
                                in_data <= sr_data;
                                in_dataS <= sr_dataS;
                                in_dataH <= sr_dataH;
                                noteS<=note+2'd2;  //define the length of the lower string shiftregister
                                noteH<=note-2'd2;   //define the length of the higher string shiftregister
                                state <= 3'd4 ;
                        end

                        4:
                        begin
                                //write ptr_in data:
                                // -- can be either computed feedback, or noise from pluck
                                Out <= new_out;
                                OutS<= new_outS;
                                OutH<= new_outH;
                                OutSum<=Out+OutS+OutH;
                                addr_reg <= ptr_in;
                                addr_regS<= ptr_inS;
                                addr_regH <= ptr_inH;
                                we <= 1'h1 ;
                                weS<= 1'h1;
                                weH<= 1'h1;
                                // feedback or new pluck
                                if (pluck )
```

```
begin
        // is this a new pluck? (part of the debouncer)
        //middle string
        if (last_pluck==0)
        begin
                // if so, reset the count
                pluck_count <= 12'd0;
                ptr_out<=12'd1;
                ptr_in<=12'd0;
                // and debounce pluck
                last_pluck <= 1'd1;
        end
        // have the correct number of random numbers been loaded?
        else if (pluck_count<note)
        begin
                //if less, load lowpass output into memory
                pluck_count <= pluck_count + 12'd1 ;
                write_data <= new_lopass;

        end
        //update feedback if not actually loading random numbers
        else
        //slow human holds button down, but feedback is still necessary
                write_data <= new_out ;


//lower string
if (last_pluck==0)
        begin
                // if so, reset the count
                pluck_countS <= 12'd0;
                ptr_inS<=12'd0;
                ptr_outS<=12'd1;
                // and debounce pluck
                last_pluck <= 1'd1;
        end
        // have the correct number of random numbers been loaded?
        else if (pluck_countS<noteS)
        begin
                //if less, load lowpass output into memory
                pluck_countS <= pluck_countS + 12'd1 ;
                write_dataS<= new_lopass;
        end
        //update feedback if not actually loading random numbers
        else
        //slow human holds button down, but feedback is still necessary
                write_dataS <=new_outS;
```

```
                                    //higher string
                                    if (last_pluck==0)
                                    begin
                                            // if so, reset the count
                                            ptr_outH<=12'd1;
                                            ptr_inH<=12'd0;
                                            pluck_countH <= 12'd0;
                                            // and debounce pluck
                                            last_pluck <= 1'd1;
                                    end
                                    // have the correct number of random numbers been loaded?
                                    else if (pluck_countH<noteH)
                                    begin
                                            //if less, load lowpass output into memory
                                            pluck_countH <= pluck_countH + 12'd1 ;
                                            write_dataH <= new_lopass;

                                    end
                                    //update feedback if not actually loading random numbers
                                    else
                                            //slow human holds button down, but feedback is still
necessary

                                            write_dataH <= new_outH ;


                    end
                            else begin
                                    // update feedback if pluck button is not pushed
                                    // and get ready for next pluck since the button is released
                                    last_pluck = 1'h0;
                                    write_data <= new_out;
                                    write_dataS <= new_outS;
                                    write_dataH <= new_outH;
                            end
                            state <= 3'd5;
                    end

                    5:
                    begin
                            we <= 0;
                            weS<= 0 ;
                            weH<= 0;
                            //update 2 ptrs for middle string
                            if (ptr_in == note)
                                    ptr_in <= 12'h0;
                    else
                              ptr_in <= ptr_in + 12'h1 ;

                    if (ptr_out == note)
                              ptr_out <= 12'h0;
                    else
                              ptr_out <= ptr_out + 12'h1 ;

                    //update 2 ptrs for lower string
                    if (ptr_inS == noteS)
                              ptr_inS <= 12'h0;
                    else
                              ptr_inS <= ptr_inS + 12'h1 ;

                    if (ptr_outS == noteS)
                              ptr_outS <= 12'h0;
                    else
                              ptr_outS <= ptr_outS + 12'h1 ;


//update 2 ptrs for higher string
                    if (ptr_inH == noteH)
                              ptr_inH <= 12'h0;
                    else
                              ptr_inH <= ptr_inH + 12'h1 ;

                    if (ptr_outH == noteH)
                              ptr_outH <= 12'h0;
                    else
                              ptr_outH <= ptr_outH + 12'h1 ;

                    state <= 3'd7;
          end

          7:
          begin
          //judge if there is another strike
                    if (audiolrclk && last_clk)
                    begin
                              state <= 3'd1 ;
                              last_clk <= 1'h0 ;
                    end
                    else if (~audiolrclk)
                    begin
                              last_clk <= 1'h1 ;
                              state<= 3'd7;
                    end
```

```
                        end

                endcase
        end
end

//make the shift register
ram_infer KS(sr_data, addr_reg, write_data, we, clock50);
ram_infer KS2(sr_dataS, addr_regS, write_dataS, weS, clock50);
ram_infer KS3(sr_dataH, addr_regH, write_dataH, weH, clock50);

//make a multiplier and compute gain*(in+out)
signed_mult gainfactor(new_out, gain, (out_data + in_data));
signed_mult gainfactor2(new_outS, gain, (out_dataS + in_dataS));
signed_mult gainfactor3(new_outH, gain, (out_dataH + in_dataH));


endmodule
// M10k ram for circular buffer
// Synchronous RAM with Read-Through-Write Behavior
// and modified for 18 bit access
// of 109 words to tune for 440 Hz
module ram_infer (q, a, d, we, clk);
output  [17:0] q;
input [17:0] d;
input [11:0] a;
input we, clk;
reg [11:0] read_add;
// define the length of the shiftregister
// 48000/2000 is 24 Hz. Should be long enough
// for any reasonable note
parameter note = 2047 ;
reg [17:0] mem [note:0];
        always @ (posedge clk)
        begin
                if (we) mem[a] <= d;
                read_add <= a;
        end
        assign q = mem[read_add];
endmodule

//signed mult of 2.16 format 2'comp
module signed_mult (out, a, b);

        output          [17:0]  out;
        input   signed  [17:0]  a;
        input   signed  [17:0]  b;
```

```
	wire	signed	[17:0]	out;
	wire	signed	[35:0]	mult_out;

	assign mult_out = a * b;
	assign out = {mult_out[35], mult_out[32:16]};
endmodule
```

MIXING SOUNDS

```
module MULTI_KEY (input logic			clk50,
						input logic [16:0]	press,

						output logic [16:0]	press1,
						output logic [16:0]	press2);

logic [4:0]	first_key;
logic [4:0]	second_key;
logic [4:0]	first_key_last;
logic [4:0]	second_key_last;

logic [4:0]	pressed_number;

logic [16:0]	press1_tmp;
logic [16:0]	press2_tmp;
logic [17:0]	press_new;
assign press_new = {1'b0, press};
integer				i;
integer			j;

always_ff @(posedge clk50)
begin
	i <= i + 1;
	if (i == 17) begin
		i <= 0;
		pressed_number <= 0;
		//first_key_last <= first_key;
		//second_key_last <= second_key;
	end
	else if (press_new == 0) begin
		first_key <= 0;
		second_key <= 0;
	end
	else if (press_new[i] == 1)
	begin
		pressed_number <= pressed_number + 1;
		if (pressed_number == 0) begin
```

```
                        first_key <= i;
                end else if (pressed_number == 1) begin
                        second_key <= i;
                end
        end

        for (j = 0; j < 17; j = j + 1)
        begin
                if (first_key == second_key) begin
                        press1_tmp <= 0;
                        press2_tmp <= 0;
                end else if (j == first_key) begin
                        press1_tmp[j] <= 1;
                end else if (j == second_key) begin
                        press2_tmp[j] <= 1;
                end else begin
                        press1_tmp[j] <= 0;
                        press2_tmp[j] <= 0;
                end
        end
end

assign press1 = {press1_tmp[16:1], 1'b0};
assign press2 = {press2_tmp[16:1], 1'b0};

endmodule
```

Autopress

```
module PRESS(input logic         clk50,
                                 input logic  [7:0]   audio_demo,
                                 input logic  [7:0]   mode_select,
                                 input logic  [16:0]  press_real,
                                 output logic [16:0]  press
                                );

logic [16:0] press_sim;
assign press_sim[0] = 1'b0;
assign press_sim[1] = (audio_demo == 8'd1) ? 1'b1 : 1'b0;
assign press_sim[2] = (audio_demo == 8'd2) ? 1'b1 : 1'b0;
assign press_sim[3] = (audio_demo == 8'd3) ? 1'b1 : 1'b0;
assign press_sim[4] = (audio_demo == 8'd4) ? 1'b1 : 1'b0;
assign press_sim[5] = (audio_demo == 8'd5) ? 1'b1 : 1'b0;
assign press_sim[6] = (audio_demo == 8'd6) ? 1'b1 : 1'b0;
assign press_sim[7] = (audio_demo == 8'd7) ? 1'b1 : 1'b0;
assign press_sim[8] = (audio_demo == 8'd8) ? 1'b1 : 1'b0;
assign press_sim[9] = (audio_demo == 8'd9) ? 1'b1 : 1'b0;
assign press_sim[10] = (audio_demo == 8'd10) ? 1'b1 : 1'b0;
assign press_sim[11] = (audio_demo == 8'd11) ? 1'b1 : 1'b0;
assign press_sim[12] = (audio_demo == 8'd12) ? 1'b1 : 1'b0;
assign press_sim[13] = (audio_demo == 8'd13) ? 1'b1 : 1'b0;
assign press_sim[14] = (audio_demo == 8'd14) ? 1'b1 : 1'b0;
assign press_sim[15] = (audio_demo == 8'd15) ? 1'b1 : 1'b0;
assign press_sim[16] = (audio_demo == 8'd16) ? 1'b1 : 1'b0;

logic [16:0] press_mode;
assign press_mode = ((mode_select == 8'd16) | (mode_select == 8'd18)) ? 17'b11111111111111111 :
17'b0;

assign press = ((press_real & press_mode) | press_sim);

endmodule
```

VGA Display

PIANO DISPLAY

```
/*
 * Seven-segment LED emulator
 *
 * Stephen A. Edwards, Columbia University
 */

module PIANO_DISPLAY(input logic          clk50, reset,
               input logic [7:0]   pixel [0:1057],
                                           input logic [1:0]   pixel_RGB [0:351],
                                           input logic [1:0]   fix_pixel_RGB [0:351],
                                           input logic [1:0]   fix_pixel_RGB_denise [0:351],
                                           input logic [4:0]   fix_keys_position [0:351],
                                    input logic [8:0]   row_num,
                                           input logic [3:0]   SW,
                                           input logic [3:0]   KEY,
                                           input logic [4:0]   key_num,
                                           input logic [7:0]   mode_select,
                                           input logic [7:0]   audio_demo,
                                           input logic [16:0]  press,
                                           input logic [7:0]   in_R1, in_G1, in_B1,
                                           input logic [7:0]   in_R2, in_G2, in_B2,
                                           input logic [7:0]   in_R3, in_G3, in_B3,
                                           input logic [7:0]   in_R4, in_G4, in_B4,

                                           input logic [7:0]   in_R5, in_G5, in_B5,
               output logic [7:0]  VGA_R, VGA_G, VGA_B,
               output logic        VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n,
                                                                            output logic       clk_vga,
                    output logic [9:0]  h,v);
/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 * HCOUNT 1599 0             1279       1599 0
 *             _______________              ________
 * ___________|    Video     |____________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *       _______________________      ______________
 * |____|       VGA_HS          |____|
 */
   // Parameters for hcount
   parameter HACTIVE      = 11'd 1280,
             HFRONT_PORCH = 11'd 32,
             HSYNC        = 11'd 192,
             HBACK_PORCH  = 11'd 96,
             HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH; // 1600

   // Parameters for vcount
   parameter VACTIVE      = 10'd 480,
             VFRONT_PORCH = 10'd 10,
             VSYNC        = 10'd 2,
             VBACK_PORCH  = 10'd 33,
             VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH; // 525

   logic [10:0]                     hcount; // Horizontal counter
                                 // Hcount[10:1] indicates pixel column (0-639)
   logic                         endOfLine;

   logic [7:0] real_VGA_R, real_VGA_G, real_VGA_B;

         integer byecounter;
         logic byechange;

         always_ff @(posedge clk50) begin
                 if (byecounter == 30000000) begin
                         byecounter <= 0;
                         byechange <= byechange + 1;
                 end else byecounter <= byecounter + 1;
         end

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          hcount <= 0;
     else if (endOfLine) hcount <= 0;
```

```
  else
           begin
                    hcount <= hcount + 11'd 1;
                            real_VGA_R <= pixel[(hcount[10:1] - 10'd146) * 3];
             real_VGA_G <= pixel[(hcount[10:1] - 10'd146) * 3 + 1];
             real_VGA_B <= pixel[(hcount[10:1] - 10'd146) * 3 + 2];
                   end

assign endOfLine = hcount == HTOTAL - 1;

// Vertical counter
logic [9:0]                          vcount;
logic                          endOfField;

always_ff @(posedge clk50 or posedge reset)
  if (reset)          vcount <= 0;
  else if (endOfLine)
    if (endOfField)   vcount <= 0;
    else              vcount <= vcount + 10'd 1;

assign endOfField = vcount == VTOTAL - 1;

// Horizontal sync: from 0x520 to 0x5DF (0x57F)
// 101 0010 0000 to 101 1101 1111
assign VGA_HS = !( (hcount[10:8] == 3'b101) & !(hcount[7:5] == 3'b111));
assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

assign VGA_SYNC_n = 1; // For adding sync to video signals; not used for VGA

// Horizontal active: 0 to 1279     Vertical active: 0 to 479
// 101 0000 0000  1280              01 1110 0000  480
// 110 0011 1111  1599              10 0000 1100  524
assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
                        !( vcount[9] | (vcount[8:5] == 4'b1111) );

/* VGA_CLK is 25 MHz
 *             __    __    __
 * clk50    __|  |__|  |__|
 *
 *             _____       __
 * hcount[0]__|     |_____|
 */
assign VGA_CLK = hcount[0]; // 25 MHz clock: pixel latched on rising edge

        //word
        reg [43:0]  bit_next [0:13];
        assign bit_next[0]  = 44'b00000000000000000000000000000000011000011110;
		assign bit_next[1]  = 44'b00001100000000000000000000000000011000011110;
		assign bit_next[2]  = 44'b00001100000000000000000000000000011000110110;
		assign bit_next[3]  = 44'b00001100000000000000000000000000011000110110;
		assign bit_next[4]  = 44'b00111111000000000000000000000000011000110110;
		assign bit_next[5]  = 44'b00111111000110000110000011111000011001100110;
		assign bit_next[6]  = 44'b00001100000110000110001111111110011001100110;
		assign bit_next[7]  = 44'b00001100000011001100001100000110011001100110;
		assign bit_next[8]  = 44'b00001100000011111100001111111110011011000110;
		assign bit_next[9]  = 44'b00001100000000110000001111111110011011000110;
		assign bit_next[10]  = 44'b00001100000011001100000000000110011011000110;
		assign bit_next[11]  = 44'b00110000000111001110000000000110011110000110;
		assign bit_next[12]  = 44'b00110000000110000110001111111100011110000110;
		assign bit_next[13]  = 44'b00000000000000000000000000000000000000000000;


	reg [50:0]  bit_color [0:13];
		assign bit_color[0]  = 51'b000000000000000000000000000000000000000000000000000;
		assign bit_color[1]  = 51'b000000000000000000000000001100000000000000000000000;
		assign bit_color[2]  = 51'b000000000000000000000000001100000000000000000000000;
		assign bit_color[3]  = 51'b000000000000000000000000001100000000000000000000000;
		assign bit_color[4]  = 51'b000000000000000000000000001100000000000000000000000;
		assign bit_color[5]  = 51'b000111011000001111000000011000001111000000011111000;
		assign bit_color[6]  = 51'b001111111000001111000000011000001111000001111111110;
		assign bit_color[7]  = 51'b001100111000110000110000011000110000110001100000110;
		assign bit_color[8]  = 51'b000000011000110000110000011000110000110000000000110;
		assign bit_color[9]  = 51'b000000011000110000110000011000110000110000000000110;
		assign bit_color[10]  = 51'b000000011000110000110000011000110000110001100000110;
		assign bit_color[11]  = 51'b000000011000001111000001110000001111000001111111110;
		assign bit_color[12]  = 51'b000000011000001111000001100000001111000000011111000;
		assign bit_color[13]  = 51'b000000000000000000000000000000000000000000000000000;


		reg [96:0]  bit_calibration [0:13];
		assign bit_calibration[0]  =
97'b00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000;
		assign bit_calibration[1]  =
97'b0000000000000000000000000000110000000000000000000000000000000011000000001100000
00000000000000000;
		assign bit_calibration[2]  =
97'b0000000000000000000000110000011000000000000000000000000000000110011000011000000000
0000000000000000;
		assign bit_calibration[3]  =
97'b0000000000000000000000110000011000000000000000000000000000000110011000011000000000
0000000000000000;
```

```
	assign bit_calibration[4]  =
97'b0000000000000000000000000011111100000000000000000000000000000001100000000011000000
000000000000000;
	assign bit_calibration[5]  =
97'b0001111111000011110000110001111110000111110000011101100001110011001100001100001111
100000011111000;
	assign bit_calibration[6]  =
97'b0001111111000011110000110000011000011000000000111111100001111111001100001100110000
000001111111110;
	assign bit_calibration[7]  =
97'b011000001100110000110011000001100001100000000011001110011000111100110000110011000
0000001100000110;
	assign bit_calibration[8]  =
97'b0110000011001100001100110000011000011111110000000001100110000110011000011001111111
100000000000110;
	assign bit_calibration[9]  =
97'b011000001100110000110011000001100001100000110000000110011000011001100001100110000
0011000000000110;
	assign bit_calibration[10]  =
97'b011000001100110000110011000001100001100000110000000110011000011001100001100110000
0011001100000110;
	assign bit_calibration[11] =
97'b0110000011000011110000110001100000011111110000000001100001111110011001110001111111
100001111111110;
	assign bit_calibration[12]  =
97'b0110000011000011110000110001100000011111110000000001100001111110011001100001111111
100000011111000;
	assign bit_calibration[13]  =
97'b000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000;


	reg [72:0]  bit_confirm [0:13];
	assign bit_confirm[0]  =
73'b0000000000000000000000000000000000000000000000000000000000000000000000000;
	assign bit_confirm[1]  =
73'b0000000000000000000000000000000111000000000000000000000000000000000000000;
	assign bit_confirm[2]  =
73'b0000000000000000000000000001100011111000000000000000000000000000000000000;
	assign bit_confirm[3]  =
73'b0000000000000000000000000001100000011000000000000000000000000000000000000;
	assign bit_confirm[4]  =
73'b0000000000000000000000000000000001100000000000000000000000000000000000000;
	assign bit_confirm[5]  =
73'b0000110011000000111011000110001111111000001111111000001111000000011111000;
	assign bit_confirm[6]  =
73'b0000111111110001111111000110001111111000001111111000001111000001111111110;
```

```
        assign bit_confirm[7]  =
73'b0011001100110001100111000110000001100000110000011000110000110001100000110;
        assign bit_confirm[8]  =
73'b0011001100110000000011000110000001100000110000011000110000110000000000110;
        assign bit_confirm[9]  =
73'b0011001100110000000011000110000001100000110000011000110000110000000000110;
        assign bit_confirm[10]  =
73'b0011001100110000000011000110000001100000110000011000110000110001100000110;
        assign bit_confirm[11]  =
73'b0011001100110000000011000110000001100000110000011000001111000001111111110;
        assign bit_confirm[12]  =
73'b0011001100110000000011000110000001100000110000011000001111000000011111000;
        assign bit_confirm[13]  =
73'b0000000000000000000000000000000000000000000000000000000000000000000000000;

        reg [79:0]  bit_no_color [0:13];
        assign bit_no_color[0]  =
80'b00000000000000000000000000000000000000000000000000000000000000000000000000000000;
        assign bit_no_color[1]  =
80'b00000000000000000000000000110000000000000000000000000000000000000000000000000000;
        assign bit_no_color[2]  =
80'b00000000000000000000000000110000000000000000000000000000000000000000000000000000;
        assign bit_no_color[3]  =
80'b00000000000000000000000000110000000000000000000000000000000000000000000000000000;
        assign bit_no_color[4]  =
80'b00000000000000000000000000110000000000000000000000000000000000000000000000000000;
        assign bit_no_color[5]  =
80'b00011101100000111100000001100000111100000001111100000000000001111000000001111110;
        assign bit_no_color[6]  =
80'b00111111100000111100000001100000111100000111111110000000000001111000000001111110;
        assign bit_no_color[7]  =
80'b00110011100011000011000001100011000011000110000011000000000110000110001100000110;
        assign bit_no_color[8]  =
80'b00000001100011000011000001100011000011000000000011000000000110000110001100000110;
        assign bit_no_color[9]  =
80'b00000001100011000011000001100011000011000000000011000000000110000110001100000110;
        assign bit_no_color[10]  =
80'b00000001100011000011000001100011000011000110000011000000000110000110001100000110;
        assign bit_no_color[11]  =
80'b00000001100000111100000111000000111100000111111110000000000001111000001100000110;
        assign bit_no_color[12]  =
80'b00000001100000111100000110000000111100000001111100000000000001111000001100000110;
        assign bit_no_color[13]  =
80'b00000000000000000000000000000000000000000000000000000000000000000000000000000000;


    reg [94:0]  bit_recognize [0:13];
```

```
	assign bit_recognize[0]  =
95'b00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000;
	assign bit_recognize[1]  =
95'b00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000;
	assign bit_recognize[2]  =
95'b00000000000000000000000000110000000000000000000000000000000000000000000000000000
000000000000000;
	assign bit_recognize[3]  =
95'b00000000000000000000000000110000000000000000000000000000000000000000000000000000
000000000000000;
	assign bit_recognize[4]  =
95'b00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000;
	assign bit_recognize[5]  =
95'b00001111100000111111110001100000111111000011111000000011110000000111110000000111110
000001110110;
	assign bit_recognize[6]  =
95'b001111111110001111111100011000011111111000111111100000111100000111111111000111111111
00011111110;
	assign bit_recognize[7]  =
95'b0011000000110000110000000001100011000000110001100011000110000110001100000110001100000
01100011001110;
	assign bit_recognize[8]  =
95'b0011111111100000110000000011000110000011000110001100011000011000000000011000111111
1100000000110;
	assign bit_recognize[9]  =
95'b0011111111100000011000000110001100000110001111111000110000110000000000110001111111
1100000000110;
	assign bit_recognize[10]  =
95'b0000000001100000000110000110001100000110001111100000110000110001100000110000000000
01100000000110;
	assign bit_recognize[11]  =
95'b0000000001100011111111000110001100000110001100000000000111100000111111111000000000
1100000000110;
	assign bit_recognize[12]  =
95'b001111111100001111111100011000110000011000110000000000111100000000111110000011111111
000000000110;
	assign bit_recognize[13]  =
95'b0000000000000000000000000000000000000000000111110000000000000000000000000000000000
000000000000000;


	reg [42:0]  bit_keys [0:13];
	assign bit_keys[0]  = 43'b0000000000000000000000000000000000000000000;
	assign bit_keys[1]  = 43'b0000000000000000000000000000000000000000110;
assign bit_keys[2]  = 43'b0000000000000000000000000000000000000000000110;
assign bit_keys[3]  = 43'b0000000000000000000000000000000000000000000110;
assign bit_keys[4]  = 43'b0000000000000000000000000000000000000110000110;
assign bit_keys[5]  = 43'b001111100000011000110000011111000000110000110;
assign bit_keys[6]  = 43'b00111111100011000110001111111111000011100110;
assign bit_keys[7]  = 43'b000000011000110001100011000001100000001110110;
assign bit_keys[8]  = 43'b000000011000111111100011111111110000000011110;
assign bit_keys[9]  = 43'b000111110000111110000011111111110000000011110;
assign bit_keys[10]  = 43'b0011111100000110000000000000000110000001100110;
assign bit_keys[11]  = 43'b0011000000001100000000000000000011000111000110;
assign bit_keys[12]  = 43'b00111111100011111110001111111110000110000110;
assign bit_keys[13]  = 43'b0000000000000111111000000000000000000000000000;


reg [43:0]  bit_free [0:13];
assign bit_free[0]  = 44'b00000000000000000000000000000000000000000000000;
assign bit_free[1]  = 44'b00000000000000000000000000000000000000011100000;
assign bit_free[2]  = 44'b00000000000000000000000000000000000000011111000;
assign bit_free[3]  = 44'b00000000000000000000000000000000000000000011000;
assign bit_free[4]  = 44'b00000000000000000000000000000000000000000011000;
assign bit_free[5]  = 44'b0000111110000000011111000000011101100011111110;
assign bit_free[6]  = 44'b001111111110001111111110001111111000111111110;
assign bit_free[7]  = 44'b00110000001100011000001100011001110000000011000;
assign bit_free[8]  = 44'b0011111111100011111111100000000110000000011000;
assign bit_free[9]  = 44'b0011111111100011111111100000000110000000011000;
assign bit_free[10]  = 44'b00000000011000000000011000000000110000000011000;
assign bit_free[11]  = 44'b00000000011000000000011000000000110000000011000;
assign bit_free[12]  = 44'b0011111111000011111111000000000110000000011000;
assign bit_free[13]  = 44'b00000000000000000000000000000000000000000000000;


reg [47:0]  bit_mode [0:13];
assign bit_mode[0]  = 48'b0000000000000000000000000000000000000000000000000000;
assign bit_mode[1]  = 48'b0000000000000001100000000000000000000000000000000000;
assign bit_mode[2]  = 48'b0000000000000001100000000000000000000000000000000000;
assign bit_mode[3]  = 48'b0000000000000001100000000000000000000000000000000000;
assign bit_mode[4]  = 48'b0000000000000001100000000000000000000000000000000000;
assign bit_mode[5]  = 48'b000011111000000110011100000000111100000001110011100;
assign bit_mode[6]  = 48'b00111111111000111111100000000111100000011111111110;
assign bit_mode[7]  = 48'b00110000001100011110001100011000011000110011001100110;
assign bit_mode[8]  = 48'b001111111111000110000001100011000001100011001100110;
assign bit_mode[9]  = 48'b001111111111000110000001100011000001100011001100110;
assign bit_mode[10]  = 48'b0000000001100011000000110001100000110001100110011 0;
assign bit_mode[11]  = 48'b0000000001100011111110000000111100000011001100110;
assign bit_mode[12]  = 48'b0011111111000011111110000000111100000011001100110;
assign bit_mode[13]  = 48'b0000000000000000000000000000000000000000000000000000;
```

```
        reg [81:0]  bit_auto_play [0:13];
        assign bit_auto_play[0]  =
82'b00000000000000000000000000000000000000000000000000000000000000000000000000000000
00;
        assign bit_auto_play[1]  =
82'b00000000000000000000000000110000000000000000000000000011000000000000000000000000
00;
        assign bit_auto_play[2]  =
82'b00000000000000000000000000110000000000000000000000000011000000000000000000000000
00;
        assign bit_auto_play[3]  =
82'b00000000000000000000000000110000000000000000000000000011000000000000000000000000
00;
        assign bit_auto_play[4]  =
82'b000000000000000000000000001100000000000000000000000001111100000000000000000000000
0;
        assign bit_auto_play[5]  =
82'b0011000110000011111000000011000011101100000111100000111111000110000110000011111000
;
        assign bit_auto_play[6]  =
82'b001100011000110000000000001100001111110000011110000000110000011000011000110000000
0;
        assign bit_auto_play[7]  =
82'b001100011000110000000000001100011001110001100001100000110000011000011000110000000
0;
        assign bit_auto_play[8]  =
82'b0011111110001111111000000011000110001100011000011000001100000110000110001111111000;
        assign bit_auto_play[9]  =
82'b0011111000001100000110000011000011111100011000011000001100000110000110001100000110
;
        assign bit_auto_play[10]  =
82'b001100000000110000011000001100001111110001100001100000110000011000011000110000011
0;
        assign bit_auto_play[11]  =
82'b0011000000001111111000001110000000001100000111100000110000000111111000001111111000
;
        assign bit_auto_play[12]  =
82'b0011111110001111111000001100000000000110000011110000011000000011111100000111111000;
        assign bit_auto_play[13]  =
82'b000111110000000000000000000000000000110000000000000000000000000000000000000000000
0;



        reg [36:0] bit_exit [0:13];
        assign bit_exit[0] = 37'b0000000000000000000000000000000000000;
        assign bit_exit[1] = 37'b0000110000000000000000000000000000000;
        assign bit_exit[2] = 37'b0000110000011000000000000000000000000;
        assign bit_exit[3] = 37'b0000110000011000000000000000000000000;
        assign bit_exit[4] = 37'b0011111110000000000000000000000000000;
        assign bit_exit[5] = 37'b00111111100011000110000110000011111000;
        assign bit_exit[6] = 37'b0000110000011000110000110001111111110;
        assign bit_exit[7] = 37'b0000110000011000011001100001100000110;
        assign bit_exit[8] = 37'b0000110000011000011111100001111111110;
        assign bit_exit[9] = 37'b0000110000011000000110000001111111110;
        assign bit_exit[10] = 37'b0000110000011000011001100000000000110;
        assign bit_exit[11] = 37'b0011000000011000111001110000000000110;
        assign bit_exit[12] = 37'b0011000000011000110000110001111111100;
        assign bit_exit[13] = 37'b0000000000000000000000000000000000000;



        reg [69:0]  bit_back_to [0:13];
        assign bit_back_to[0]  =
70'b0000000000000000000000000000000000000000000000000000000000000000000000;
        assign bit_back_to[1]  =
70'b0000000000000001100000000000000011000000000000000000000000000000000110;
        assign bit_back_to[2]  =
70'b0000000000000001100000000000000011000000000000000000000000000000000110;
        assign bit_back_to[3]  =
70'b0000000000000001100000000000000011000000000000000000000000000000000110;
        assign bit_back_to[4]  =
70'b0000000000000111111000000110000110000000000000000000000000000000000110;
        assign bit_back_to[5]  =
70'b0000111100000111111000000110000110000011111000000011111000000011100110;
        assign bit_back_to[6]  =
70'b0000111100000000110000000000111001100011111111100011000000000000011111110;
        assign bit_back_to[7]  =
70'b0011000011000000110000000000111011000110000001100011000000000001100011110;
        assign bit_back_to[8]  =
70'b0011000011000000110000000000000111100000000000110001111111000000110000110;
        assign bit_back_to[9]  =
70'b0011000011000000110000000000000111100000000000110001100000011000110000110;
        assign bit_back_to[10]  =
70'b0011000011000000110000000000011001100011000000110001100000011000110000110;
        assign bit_back_to[11]  =
70'b0000111100000011000000000001110001100011111111100011111110000000011111110;
        assign bit_back_to[12]  =
70'b0000111100000011000000000001100001100000111110000011111110000000011111110;
        assign bit_back_to[13]  =
70'b0000000000000000000000000000000000000000000000000000000000000000000000;
```

```
        reg [67:0]  bit_screen [0:13];
        assign bit_screen[0]  =
68'b00000000000000000000000000000000000000000000000000000000000000000000;
        assign bit_screen[1]  =
68'b00000000000000000000000000000000000000000000000000000000000000000000;
        assign bit_screen[2]  =
68'b00000000000000000000000000000000000000000000000000000000000000000000;
        assign bit_screen[3]  =
68'b00000000000000000000000000000000000000000000000000000000000000000000;
        assign bit_screen[4]  =
68'b00000000000000000000000000000000000000000000000000000000000000000000;
        assign bit_screen[5]  =
68'b000011111100000011111000000011111000000111011000000111110000011111000;
        assign bit_screen[6]  =
68'b0001111111100011111111100011111111100011111110001111111110001111111110;
        assign bit_screen[7]  =
68'b0011000000110001100000011000110000001100011001110001100000011000000000110;
        assign bit_screen[8]  =
68'b001100000011000111111111000111111111000000000110000000000011000000000110;
        assign bit_screen[9]  =
68'b001100000011000111111111000111111111000000000110000000000011000011111100;
        assign bit_screen[10]  =
68'b0011000000110000000000011000000000001100000000110001100000011000111111100;
        assign bit_screen[11]  =
68'b0011000000110000000000011000000000001100000000110001111111110001100000000;
        assign bit_screen[12]  =
68'b001100000011000111111110000111111111000000000011000000111110000011111110;
        assign bit_screen[13]  =
68'b00000000000000000000000000000000000000000000000000000000000000000000;


        reg [174:0] bit_screen_calibration [0:13];
        assign bit_screen_calibration[0]  =
175'b0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000011111000000000000000000000000000000000000000000000000000000000
000011111000;
        assign bit_screen_calibration[1]  =
175'b0000000000000000000000000000000011000000000000000000000000000000000011000000000
011000000000000000011111111000000000000000000000000000000000000000000000000000000
000011111110;
        assign bit_screen_calibration[2]  =
175'b0000000000000000000000000011000001100000000000000000000000000000000001100011000 0
011000000000000000011000011000000000000000000000000000000000000000000000000000000000
000000000110;
        assign bit_screen_calibration[3]  =
175'b0000000000000000000000000011000001100000000000000000000000000000000001100011000 0
```

```
0110000000000000000000000110000000000000000000000000000000000000000000000000000000
000000000110;
	assign bit_screen_calibration[4]  =
175'b0000000000000000000000000000001111110000000000000000000000000000000000110000000000
11000000000000000000000011000000000000000000000000000000000000000000000000000000000
00000000110;
	assign bit_screen_calibration[5]  =
175'b000011111100000011110000011000111111000000111110000000111011000000111001100011000001
10000011111000000000000011000001111110000001111100000001111100000001110110000001111100000
000000110;
	assign bit_screen_calibration[6]  =
175'b000111111110000011110000011000001100000110000000000011111110000011111110001100000 1
1000110000000000000000000110000111111110001111111110001111111110001111111000111111111000
01111100;
	assign bit_screen_calibration[7]  =
175'b0011000001100011000001100011000001100000110000000000011001110001100011110001100000
1100011000000000000000000011000110000011000110000011000110000011000110011100011000001100
0011111100;
	assign bit_screen_calibration[8]  =
175'b0011000001100011000001100011000001100000111111100000000000011000110000011000110000 0
11000111111100000000000011000110000011000111111111000111111111000000000110000000000001100
011000000;
	assign bit_screen_calibration[9]  =
175'b00110000011000110000011000110000011000001100000110000000011000110000011000110000 0
110001100000110000000000110001100000110001111111110001111111110000000001100000000000110
0011000000;
	assign bit_screen_calibration[10]  =
175'b00110000011000110000011000110000011000001100000110000000011000110000011000110000 0
110001100000110001100000100001100000110000000000110000000000110000000011000110000011
00011000000;
	assign bit_screen_calibration[11]  =
175'b00110000011000000111100000011000110000000111111100000000000110000011111110001100011 1
00000111111100000111111110000110000011000000000011000000000011000000001100011111111100
011111110;
	assign bit_screen_calibration[12]  =
175'b0011000001100000011110000001100011000000011111110000000000011000001111111000110001 10
000011111110000000111110000011000001100011111110000111111110000000001100000111110000 0
01111110;
	assign bit_screen_calibration[13]  =
175'b000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000;


	reg [160:0] bit_color_calibration [0:13];
	assign bit_color_calibration[0]  =
161'b0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000011111000000000000000000000000000000000000000000000000011111000;
	assign bit_color_calibration[1]  =
161'b0000000000000000000000000000000011000000000000000000000000000000000011000000000
0110000000000000001111111100000000000000000000000000110000000000000000000111111100;
	assign bit_color_calibration[2]  =
161'b0000000000000000000000000110000011000000000000000000000000000000000011000110000
0110000000000000001100001100000000000000000000000000110000000000000000001100001100;
	assign bit_color_calibration[3]  =
161'b0000000000000000000000000110000011000000000000000000000000000000000011000110000
0110000000000000000000001100000000000000000000000000110000000000000000000000000110;
	assign bit_color_calibration[4]  =
161'b00000000000000000000000000011111100000000000000000000000000000000000110000000000
110000000000000000000001100000000000000000000000000110000000000000000000000000110;
	assign bit_color_calibration[5]  =
161'b0000111111100000111100000110001111110000011111000000111011000001110011000110000001
1000001111100000000000000110000000111011000001111000000001100000111100000000000000110;
	assign bit_color_calibration[6]  =
161'b0000111111100000111100000110000011000001100000000000111111100000111111100011000001
10001100000000000000000011000000111111100000111100000000110000011110000000000000110;
	assign bit_color_calibration[7]  =
161'b001100000011000110000011000110000001100000110000000000011001110001100011110001100000
1100011000000000000000000110000001100111000110000110000011000110000011000000000000110;
	assign bit_color_calibration[8]  =
161'b001100000011000110000011000110000001100000111111100000000000110001100000110001100000
110001111111000000000000011000000000000110001100001100000110001100000110000000000110;
	assign bit_color_calibration[9]  =
161'b00110000001100011000001100011000000110000011000000110000000001100011000000110001100000
1100011000001100000000000110000000000001100011000011000001100011000001100000000000110;
	assign bit_color_calibration[10]  =
161'b00110000001100011000001100011000000110000011000000110000000001100011000000110001100000
1100011000001100011000000100000000000001100011000011000001100011000001100011000000100;
	assign bit_color_calibration[11]  =
161'b0011000000110000011110000001100011000000011111110000000000011000000111111000110001111
000011111110000011111111000000000000011000000111100000011100000011110000011111111100;
	assign bit_color_calibration[12]  =
161'b0011000000110000011110000001100011000000011111110000000000011000000111111000110001110
0000111111100000001111100000000000000110000001111000000110000000111100000000111111000;
	assign bit_color_calibration[13]  =
161'b0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000;

	reg [334:0] bit_adjust1 [0:13];
	assign bit_adjust1[0]   =
335'b0000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000011110
0000;
	assign bit_adjust1[1]	=
335'b000000000000000000000000000000000110000000000000000000000000000000000000000000000
00000000000000000000000000000000000000110000000000110000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000001100000110000000000110000000000000000000000000000000000110000000000000011110
0000;
	assign bit_adjust1[2]	=
335'b00000000000000000000000000110000011000000000000000000000000000000000000000000000
000000000000000000000000000000000000001100011000001100000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000001100000110000000000110000000000000000000000001100000000110000000000000110011
0000;
	assign bit_adjust1[3]	=
335'b00000000000000000000000000110000011000000000000000000000000000000000000000000000
000000000000000000000000000000000000001100011000001100000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000001100000110000000000110000000000000000000000001100000000110000000000000110011
0000;
	assign bit_adjust1[4]	=
335'b000000000000000000000000000000000110000000000000000000000000000000000000000000000
00000000000000000000000000000000000000110000000011111100000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000011000111111000000111111000000000000000000000000000000001100000000000001100110
000;
	assign bit_adjust1[5]	=
335'b0000111110000000111111000011000001100000000111111000000111110000000111110000000111
0110001111100000000001111100000000000110001100011111000001111100001100001100000000011
1110000001110110000011111000000111001110000001111100000001111100000000001111100000001
1100110001111110000001111110001111100000110000110000000000000110011100000110000011000;
	assign bit_adjust1[6]	=
335'b0011111111100001111111100011000001100000001111111100011111111100011111111100011111
1100011111110000001100000000000000001100011000001100000011111111000110000110000011000
000000011111110001111111110001111111110001100000000001111111110000001111111110000001111
111000001100000000001100000111111100011000011000110000000011111110000011000011000;
	assign bit_adjust1[7]	=
335'b0011000000110001100000110001100000110000001100000110001100000110001100000011000110
0111000110001100000001100000000000000001100011000001100000110000011000110000110000011
000000000110011100011000001100011001100110001100000000001100000110000001100000011000
110001111000001100000000000110000000000110001100001100011000000000111100011000111111110
00;
	assign bit_adjust1[8]	=
335'b0011111111100011000001100011000001100000011000001100011111111100011111111100000000
01100011000110000001111111000000000001100011000001100000110000011000110000110000001111
```

```
1110000000000110001111111110001100110011000111111100000000000001100000011111111110001100
0001100000011000000000011000000000011000110000110001100000000110000011001111111111100;
	assign bit_adjust1[9]	=
335'b001111111111000110000011000110000011000000110000011000111111111000111111111000000000
011000111111100000011000001100000000110001100000110000011000001100011000011000000001100
00011000000000110001111111110001100110011000110000011000000000011000000111111111000110
0000110000011000000000011000000111110000110000110001100000000110000011001100000001100;
	assign bit_adjust1[10]	=
335'b000000000110001100000110001100000110000000110000011000000000001100000000000110000000
0011000111110000000001100000110000000001100011000001100000110000011000110000110000000110
000011000000000110000000000011000110011001100011000001100011000001100000000000000110001
1000001100000011000000000001100000111111000011000011000110000000011000001100110000000110
0;
	assign bit_adjust1[11]	=
335'b00000000011000110000011000110001110000000110000011000000000011000000000000110000000
0011000110000000000011111110000000001110000110001100000000110000011000111111000000000111
111100000000000110000000000011000110011001100011111110000011111111100000000000000001100011
000001100011000000000011000000011000000000111111000001100011000111111100011000000000110
;
	assign bit_adjust1[12]	=
335'b001111111110000110000011000110001100000000110000011000111111111000111111110000000000
011000110000000000001111111000000000110000011000110000000011000001100011111100000000001111
111000000000001100011111111000011001100110001111111000000001111100000000011111111000011000
000110001100000000000110000000111111100011111110000011111110001111111000110000000000110;
	assign bit_adjust1[13]	=
335'b0000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000111110000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000001111000000000000000000000000
0000;


	reg [367:0] bit_adjust2 [0:13];
	assign bit_adjust2[0]	=
368'b0000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000;
	assign bit_adjust2[1]	=
368'b0000000000000000000000000000000000000000000000000000000000000000000000000000011
000001100000000000000000000000000000000000000000000001100000110000000011100000000000
000000000000000000000000011000110000000000110000000000000000000000000000000000000000
000000000011000001100000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000;
	assign bit_adjust2[2]	=
368'b0000000000000000000000000000000000000000000000000000000000000000000000000000011
```

```
0000011000000000000000000000000000000000000000000000001100000110000000001111100000000
000000000000000000000000011000110000000000011000000000011000000000000000000000000000
0000000000110000011000000000000000000000110000000000000000000000000000000000000000000
000000000000000000000000000000000000;
	assign bit_adjust2[3]	=
368'b000000000000000000000000000000000000000000000000000000000000000000000000000000011
000001100000000000000000000000000000000000000000000011000001100000000000110000000
000000000000000000000000011000110000000000011000000000011000000000000000000000000000
0000000000110000011000000000000000000000110000000000000000000000000000000000000000000
000000000000000000000000000000000000;
	assign bit_adjust2[4]	=
368'b000000000000000000000000000000000000000000000000000000000000000000000000000000011
000001100000000000000000000000000000000000000000000011000111111000000000110000000
000000000000000000000000011000110000000000011000000000000000000000000000000000000000
0000000000110001111110000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000;
	assign bit_adjust2[5]	=
368'b0000111110000000111110000001110110000011111000000000110000001100000111100000001100
0001100000111110000011000110000000011111000000011100110001111110000001111111000011110
0000000001111100000000110001100111000001100111000001100001110011100000000011111000000000
111001100011111100000000111111000011000000111110000001110110000011111000000001111100000
0111011000011101100000011111000;
	assign bit_adjust2[6]	=
368'b00110000000000011111111100011111110001100000000000000110000001100000111100000001100
000110001111111100011000110000001111111110000011111110000011000000001111111000011110
00000001111111110000011000111111100000111111000001100011111111100000011111111000001
1111110000011000000000111111110001100000011111110001111111000110000000000111111110000
111111000011111100011000000000;
	assign bit_adjust2[7]	=
368'b00110000000000011000001100011001110001100000000000000110011001100011000011000000110
00001100011000001100011000110000001100000110001100011110000011000000000001100000110000
011000000011000001100000110001111000110001111000110001100011001100110000000110000011000
11000111100000110000000011000001100011000000000000110001100111000110000000000011000001 1
0001100111000110011100011000000000;
	assign bit_adjust2[8]	=
368'b0011111110000011111111100000000011000111111100000000110011001100011000011000001100
00011000111111110001111111000000111111111000110000011000001100000000000011000001100001
10000001111111110000011000110000011000110000011000110001100110011000000111111111100011
0000011000001100000000011000001100011000000000001100000000011000111111000001111111100
0110001100011000110001111111000;
	assign bit_adjust2[9]	=
368'b0011000000110001111111110000000011000110000001100000011001100110001100001100000001100
0001100011111111100011111000000000111111111000110000011000001100000000000011000000110000
110000001111111110000011000110000011000110000011000110001100110011000000111111111100011
000001100000110000000001100000110001100000000111110000000000110001100000110001111111100
0011111100001111110001100000110;
```

```
	assign bit_adjust2[10]	=
368'b0011000001100000000000110000000011000110000011000000110011001100011000011000000110
0000110000000000011000110000000000000000000011000110000011000001100000000000110000001100
0011000000000000000110000011000110000011000110000011000110001100110011000000000000000110
0011000000110000001100000000110000001100011000000111111000000000011000110000001100000000000
1100001111110000111111000110000011 0;
	assign bit_adjust2[11]	=
368'b0011111110000000000000110000000011000111111000000000111111110000000111100000111000
0111000000000000110001100000000000000000000110001100000110001100000000000000110000000111
1000000000000000011000111000011111110000011111110000011000110011001100000000000000011000
1100000110001100000000000110000011000110000001100000000000000110001111111000000000000011
000000001100000000110001111111000;
	assign bit_adjust2[12]	=
368'b0011111110000011111111000000000110001111111000000000011111100000001111000001100000
0110000011111111000011111110000001111111100001100000110001100000000000000110000000011110
000000011111111000011000001111111000001111111000001100011001100110000001111111100000110
0000110001100000000000110000011000110000001111110000000001100011111110000011111111000000
0000011000000000110001111111000;
	assign bit_adjust2[13]	=
368'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000011111100000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000001100000000110000000000000;


	reg [334:0] bit_autoplay [0:13];
	assign bit_autoplay[0] =
335'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000;
	assign bit_autoplay[1] =
335'b00001100000000000000000000000000000000000000000000000000000000000000000000000001100
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000001100000000111000000000000000000000000000011000001100000000000000000000000000
0000000000000000110000000000000000000000000000000000000000000000000000000000011000000000
0000;
	assign bit_autoplay[2] =
335'b00001100000000000000000000000000110000000000000000000000000000000000000000000001100
0000000000000000011000000000000000000000000000000000000000000000000000000000000000000000
000000001100011000111110000000000000000000000000000011000001100000000000000000000000000
0000000000000000110000000000000000000000000000000000000000000000000000000000011000000000
0000;
	assign bit_autoplay[3] =
335'b00001100000000000000000000000000110000000000000000000000000000000000000000000001100
```

```
00000000000000000001100000000000000000000000000000000000000000000000000000000000
000000001100011000000110000000000000000000000000000001100000110000000000000000000000000
00000000000000000011000000000000000000000000000000000000000000000000000000110000000
0000;
	assign bit_autoplay[4] =
335'b000011000000000000000000000000000000000000000000000000000000000000000001111110
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000001100000000000110000000000000000000000000000110001111110000000000000000000000
0000000000000001111110000000000000000000000000000000000000000000000001100000000
000;
	assign bit_autoplay[5] =
335'b000011000000111110000000011111100001100001110011100000111011000000111110000011111100
000000111111000011000000000111110000000111001110000000111110000000011111100000000011111000
000011000110001111111000000000111110000000011100110001111110000000001111100000011101100001
10001100011111100000000011111000000111110000000011111000000000111110000000001100001110110;
	assign bit_autoplay[6] =
335'b00001100011000000000000111111110001100011111111110001111111000111111111000001100000
000011111111000110000000111111111000111111111100011000000000001111111100000011111111000
0011000110001111111000000111111111000001111111000001100000000111111111000011111000110
0011000000110000000001111111110001111111000110000000000011111111100000110000111110;
	assign bit_autoplay[7] =
335'b000011000110000000000011000000110001100011001100110001100111000110000001100000011000
00000110000001100011000000011000000110001100110011000110000000000110000001100000011000000
110000001100011000000110000000011000000110001100011110000001100000000110000001100011001111
000110001100000110000000001100000011000000000110001100000000000110000001100000110001100111
10;
	assign bit_autoplay[8] =
335'b000011000111111000000110000001100011000110011001100000000011000111111111000000110000
00001100000011000110000000111111111000110011001100011111110000001100000011000000011111111110
000011000110000000110000000001111111110001100000011000000110000000001111111110001100011000
1111111000000110000000001111111110000111110000111111100000111111111000000110001100110;
	assign bit_autoplay[9] =
335'b000011000110000001100011000000110001100011001100110000000001100011111111100000110000
000011000000110001100000001111111110001100110011000110000001100011000000110000000111111111
000000110001100000011000000000111111111000110000001100000011000000000111111111000011111000
1111100000000110000000011111111100011111100001100000011000111111111000000110000011111110;
	assign bit_autoplay[10] =
335'b000011000110000001100011000000110001100011001100110000000001100000000000110000011000
0000011000000110001100000000000000110001100110011000110000001100011000000110000000000000
1100000011000110000000110000000000000000110001100000011000000110000000000000000011000011111
0001100000000000110000000000000000110001100000000110000001100000000000110000001100000111111
10;
	assign bit_autoplay[11] =
335'b00111000001111111000000110000001100011000110011001100000000011000000000001100011000000
00000110000001100011000000000000000110001100110011000111111100000110000001100000000000000001
10001110000011000000110000000000000000110001100000011000110000000000000000000110000000011
```

```
000110000000001100000000000000000011000111111100011111110000000000000110001110000000000011
0;
	assign bit_autoplay[12] =
335'b00110000001111111000000110000001100011000110011001100000000011000111111110000110000000
00001100000011000110000000111111110000110011001100011111110000011000000110000000111111100
00110000001100000011000000000111111110000110000001100011000000000011111111000000000011000
11111110001100000000001111111100000111111000111111100000111111110000110000000000110;
	assign bit_autoplay[13] =
335'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
1100001111110000000000000000000000000000000000000000000000000000000000000000000000000000000
0110;


	reg [205:0] bit_freemode [0:13];
	assign bit_freemode[0] =
206'b00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000;
	assign bit_freemode[1] =
206'b0000000000000000000000110000000000001100000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000011000000000000000000000000000110000000000000000
000000000000000000000000000000001100000000000;
	assign bit_freemode[2] =
206'b0000000000000000000000110001100000011000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000011000000000000000000000000000110000000000000000
000000000000000000000000000000001100000000000;
	assign bit_freemode[3] =
206'b0000000000000000000000110001100000011000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000011000000000000000000000000000110000000000000000
000000000000000000000000000000001100000000000;
	assign bit_freemode[4] =
206'b000000000000000110000110000000000001100000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000111111000000000000000000000000001100000000000000000
00000000000000000000000000000001100000000000;
	assign bit_freemode[5] =
206'b000011111100000011000011000110000001100000001100001100000111100000110001100000001110
110000001111100000110000110000011111000001111110000011111000000001110011000110000001100
0000110001100000011111000000001100001110110;
	assign bit_freemode[6] =
206'b001111111111000011100110001100000110000000110000110000011110000001100011000000111111
100011111111100011000011000111111111000000110000011000000000000011111110001100000001100000
0011000110001100000000000011000011111110;
	assign bit_freemode[7] =
206'b00110000001100000111011000110000001100000001100001100011000011000110001100000011001
```

```
110001100000110001100001100011000001100000110000011000000000110001111000110011001100
000011000110001100000000000001100011001110;
        assign bit_freemode[8] =
206'b001111111111000000011110001100000110000000110000110001100001100011111110000000000001
100011111111100011000011000111111111000001100000111111000001100000110001100110011000
001111111000111111100000001100011000110;
        assign bit_freemode[9] =
206'b001111111111000000011110001100000110000000110000110001100001100011111000000000000001
100011111111100011000011000111111111000001100000110000011000110000011000110011001100
0001111100000110000011000001100001111110;
        assign bit_freemode[10] =
206'b000000000110000011001100011000001100000011000011000110000110001100000000000000000
1100000000001100001100110000000000001100000110000011000001100011000001100011001100110
000001100000000110000011000001100001111110;
        assign bit_freemode[11] =
206'b000000000110001110001100011000111000000011111100000001111000001100000000000000000
110000000000110000011110000000000000110001100000001111111000001100000110000111111110000
000011000000000111111000000111000000000110;
        assign bit_freemode[12] =
206'b001111111110000110000110001100011000000000111111000000011110000011111110000000000001
10001111111100000001100000011111111000110000000111111100000110000011000000111111000000
0011111110001111111000001100000000000110;
        assign bit_freemode[13] =
206'b000000000000000000000000000000000000000000000000000000000000000001111110000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000011111100000000000000000000000000000110;


        reg[22:0] bit_dao [0:13];
        assign bit_dao[0] = 23'b00000000000000000000000;
        assign bit_dao[1] = 23'b000000000000001100000000;
        assign bit_dao[2] = 23'b000000000000001100000000;
        assign bit_dao[3] = 23'b000000000000001100000000;
        assign bit_dao[4] = 23'b000000000000001100000000;
        assign bit_dao[5] = 23'b000001111000001100111000;
        assign bit_dao[6] = 23'b000001111000001111111000;
        assign bit_dao[7] = 23'b00110000110001111000110;
        assign bit_dao[8] = 23'b00110000110001100000110;
        assign bit_dao[9] = 23'b00110000110001100000110;
        assign bit_dao[10] = 23'b00110000110001100000110;
        assign bit_dao[11] = 23'b000001111000001111111000;
        assign bit_dao[12] = 23'b000001111000001111111000;
        assign bit_dao[13] = 23'b00000000000000000000000;


        reg[21:0] bit_re [0:13];
        assign bit_re[0] = 22'b0000000000000000000000;
        assign bit_re[1] = 22'b0000000000000000000000;
        assign bit_re[2] = 22'b0000000000000000000000;
        assign bit_re[3] = 22'b0000000000000000000000;
        assign bit_re[4] = 22'b0000000000000000000000;
        assign bit_re[5] = 22'b0000111110000001110110;
        assign bit_re[6] = 22'b0011111111110001111110;
        assign bit_re[7] = 22'b0011000001100011001110;
        assign bit_re[8] = 22'b0011111111110000000110;
        assign bit_re[9] = 22'b0011111111110000000110;
        assign bit_re[10] = 22'b0000000000110000000110;
        assign bit_re[11] = 22'b0000000000110000000110;
        assign bit_re[12] = 22'b0011111111110000000110;
        assign bit_re[13] = 22'b0000000000000000000000;


        reg[17:0] bit_mi [0:13];
        assign bit_mi[0] = 18'b000000000000000000;
        assign bit_mi[1] = 18'b000000000000000000;
        assign bit_mi[2] = 18'b001100000000000000;
        assign bit_mi[3] = 18'b001100000000000000;
        assign bit_mi[4] = 18'b000000000000000000;
        assign bit_mi[5] = 18'b001100001110011100;
        assign bit_mi[6] = 18'b001100011111111110;
        assign bit_mi[7] = 18'b001100011001100110;
        assign bit_mi[8] = 18'b001100011001100110;
        assign bit_mi[9] = 18'b001100011001100110;
        assign bit_mi[10] = 18'b001100011001100110;
        assign bit_mi[11] = 18'b001100011001100110;
        assign bit_mi[12] = 18'b001100011001100110;
        assign bit_mi[13] = 18'b000000000000000000;


        reg[21:0] bit_fa [0:13];
        assign bit_fa[0] = 22'b0000000000000000000000;
        assign bit_fa[1] = 22'b0000000000000011100000;
        assign bit_fa[2] = 22'b0000000000000011111000;
        assign bit_fa[3] = 22'b0000000000000000011000;
        assign bit_fa[4] = 22'b0000000000000000011000;
        assign bit_fa[5] = 22'b0000111110000011111110;
        assign bit_fa[6] = 22'b0011000000000011111110;
        assign bit_fa[7] = 22'b0011000000000000011000;
        assign bit_fa[8] = 22'b0011111111000000011000;
        assign bit_fa[9] = 22'b0011000001100000011000;
        assign bit_fa[10] = 22'b0011000001100000011000;
        assign bit_fa[11] = 22'b0011111111000000011000;
        assign bit_fa[12] = 22'b0011111111000000011000;
        assign bit_fa[13] = 22'b0000000000000000000000;
```

```
reg[20:0] bit_so [0:13];
assign bit_so[0] = 21'b000000000000000000000;
assign bit_so[1] = 21'b000000000000000000000;
assign bit_so[2] = 21'b000000000000000000000;
assign bit_so[3] = 21'b000000000000000000000;
assign bit_so[4] = 21'b000000000000000000000;
assign bit_so[5] = 21'b000011110000011111000;
assign bit_so[6] = 21'b000011110000011111110;
assign bit_so[7] = 21'b001100001100000000110;
assign bit_so[8] = 21'b001100001100001111100;
assign bit_so[9] = 21'b001100001100011111100;
assign bit_so[10] = 21'b001100001100011000000;
assign bit_so[11] = 21'b000011110000011111110;
assign bit_so[12] = 21'b000011110000001111110;
assign bit_so[13] = 21'b000000000000000000000;


reg[18:0] bit_la [0:13];
assign bit_la[0] = 19'b0000000000000000000;
assign bit_la[1] = 19'b0000000000000000110;
assign bit_la[2] = 19'b0000000000000000110;
assign bit_la[3] = 19'b0000000000000000110;
assign bit_la[4] = 19'b0000000000000000110;
assign bit_la[5] = 19'b0000111110000000110;
assign bit_la[6] = 19'b0011000000000000110;
assign bit_la[7] = 19'b0011000000000000110;
assign bit_la[8] = 19'b0011111110000000110;
assign bit_la[9] = 19'b0011000001100000110;
assign bit_la[10] = 19'b0011000001100000110;
assign bit_la[11] = 19'b0011111110000011100;
assign bit_la[12] = 19'b0011111110000011000;
assign bit_la[13] = 19'b0000000000000000000;


reg[15:0] bit_xi [0:13];
assign bit_xi[0] = 16'b0000000000000000;
assign bit_xi[1] = 16'b0000000000000000;
assign bit_xi[2] = 16'b0011000000000000;
assign bit_xi[3] = 16'b0011000000000000;
assign bit_xi[4] = 16'b0000000000000000;
assign bit_xi[5] = 16'b0011000110000110;
assign bit_xi[6] = 16'b0011000110000110;
assign bit_xi[7] = 16'b0011000011001100;
assign bit_xi[8] = 16'b0011000011111100;
assign bit_xi[9] = 16'b0011000000110000;
```

```
        assign bit_xi[10] = 16'b0011000011001100;
        assign bit_xi[11] = 16'b0011000111001110;
        assign bit_xi[12] = 16'b0011000110000110;
        assign bit_xi[13] = 16'b0000000000000000;


  //Background & Animation
  assign h = hcount[10:1];
  assign v = vcount[9:0];

        logic animation_1, animation_2, animation_3, animation_4, animation_5, animation_6,
animation_7, animation_8, animation_9, animation_10, animation_11, animation_12, animation_13,
animation_14, animation_15, animation_16;

        assign animation_1 = (vcount <= 10'd295) & (hcount > 11'd80 & hcount <= 11'd150);
  assign animation_2 = (vcount <= 10'd295) & (hcount > 11'd150 & hcount <= 11'd220);
  assign animation_3 = (vcount <= 10'd295) & (hcount > 11'd220 & hcount <= 11'd290);
  assign animation_4 = (vcount <= 10'd295) & (hcount > 11'd290 & hcount <= 11'd360);
  assign animation_5 = (vcount <= 10'd295) & (hcount > 11'd360 & hcount <= 11'd430);
  assign animation_6 = (vcount <= 10'd295) & (hcount > 11'd430 & hcount <= 11'd500);
  assign animation_7 = (vcount <= 10'd295) & (hcount > 11'd500 & hcount <= 11'd570);
  assign animation_8 = (vcount <= 10'd295) & (hcount > 11'd570 & hcount <= 11'd640);
  assign animation_9 = (vcount <= 10'd295) & (hcount > 11'd640 & hcount <= 11'd710);
  assign animation_10 = (vcount <= 10'd295) & (hcount > 11'd710 & hcount <= 11'd780);
  assign animation_11 = (vcount <= 10'd295) & (hcount > 11'd780 & hcount <= 11'd850);
  assign animation_12 = (vcount <= 10'd295) & (hcount > 11'd850 & hcount <= 11'd920);
  assign animation_13 = (vcount <= 10'd295) & (hcount > 11'd920 & hcount <= 11'd990);
  assign animation_14 = (vcount <= 10'd295) & (hcount > 11'd990 & hcount <= 11'd1060);
  assign animation_15 = (vcount <= 10'd295) & (hcount > 11'd1060 & hcount <= 11'd1130);
  assign animation_16 = (vcount <= 10'd295) & (hcount > 11'd1130 & hcount <= 11'd1200);

        logic keychar1, keychar2, keychar3, keychar4, keychar5, keychar6, keychar7, keychar8,
keychar9, keychar10, keychar11, keychar12, keychar13, keychar14, keychar15, keychar16;

  assign keychar1 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd55 & hcount[10:1] < 11'd63);
  assign keychar2 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd90 & hcount[10:1] < 11'd98);
  assign keychar3 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd125 & hcount[10:1] < 11'd133);
  assign keychar4 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd160 & hcount[10:1] < 11'd168);
  assign keychar5 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd195 & hcount[10:1] < 11'd203);
  assign keychar6 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd230 & hcount[10:1] < 11'd238);
  assign keychar7 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd265 & hcount[10:1] < 11'd273);
  assign keychar8 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd300 & hcount[10:1] < 11'd308);
  assign keychar9 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd335 & hcount[10:1] < 11'd343);
  assign keychar10 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd370 & hcount[10:1] < 11'd378);
  assign keychar11 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd405 & hcount[10:1] < 11'd413);
  assign keychar12 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd440 & hcount[10:1] < 11'd448);
  assign keychar13 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd475 & hcount[10:1] < 11'd483);
assign keychar14 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd510 & hcount[10:1] < 11'd518);
assign keychar15 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd545 & hcount[10:1] < 11'd553);
assign keychar16 = (vcount >= 10'd464) & (hcount[10:1] >= 10'd580 & hcount[10:1] < 11'd588);

        logic [2:0] key_digit, key_char_offset;

        always_comb begin
                if (key1) begin key_digit = 3'd1; key_char_offset = 3'd7;end
                else if (key2) begin key_digit = 3'd2; key_char_offset = 3'd2;end
                else if (key3) begin key_digit = 3'd3; key_char_offset = 3'd5;end
                else if (key4) begin key_digit = 3'd4; key_char_offset = 3'd0;end
                else if (key5) begin key_digit = 3'd5; key_char_offset = 3'd3;end
                else if (key6) begin key_digit = 3'd6; key_char_offset = 3'd6;end
                else if (key7) begin key_digit = 3'd7; key_char_offset = 3'd1;end
                else if (key8) begin key_digit = 3'd1; key_char_offset = 3'd4;end
                else if (key9) begin key_digit = 3'd2; key_char_offset = 3'd7;end
                else if (key10) begin key_digit = 3'd3; key_char_offset = 3'd2;end
                else if (key11) begin key_digit = 3'd4; key_char_offset = 3'd5;end
                else if (key12) begin key_digit = 3'd5; key_char_offset = 3'd0;end
                else if (key13) begin key_digit = 3'd6; key_char_offset = 3'd3;end
                else if (key14) begin key_digit = 3'd7; key_char_offset = 3'd6;end
                else if (key15) begin key_digit = 3'd1; key_char_offset = 3'd1;end
                else if (key16) begin key_digit = 3'd2; key_char_offset = 3'd4;end
                else begin key_digit = 3'd0; key_char_offset = 3'd0;end
        end

        logic [7:0] keysegment;
logic [7:0] keycurSegs;
        KEYCHAR_DISP key_disp(.digit(key_digit), .offset(key_char_offset), .*);

        //Test char
HEX_CONVERT hex(.*);

logic inChar;
        assign inChar = (vcount[9:4] == 6'd1) & (hcount[10:1] < 10'd64);

        logic [7:0] segment;
logic [7:0] curSegs;
        logic [8:0] digit_1, digit_2;

        assign digit_1 = {3'b0, row_num};
        assign digit_2 = {1'b0, audio_demo};

        //Test pixel line
        logic real_line_area, seg_line_area, denoise_line_area, fix_keys_line_area;
        assign real_line_area = (vcount[9:3] == 7'b0001101) & (hcount[10:1] > 10'd145) &(hcount[10:1]
< 10'd496);
        assign seg_line_area = (vcount[9:3] == 7'b0001111) & (hcount[10:1] > 10'd145) & (hcount[10:1]
< 10'd496);
        assign denoise_line_area = (vcount[9:3] == 7'b0010001) & (hcount[10:1] > 10'd145) &
(hcount[10:1] < 10'd496);
        assign fix_keys_line_area = (vcount[9:3] == 7'b0010011) & (hcount[10:1] > 10'd145) &
(hcount[10:1] < 10'd496);

        logic real_line_area1, seg_line_area1;

        assign real_line_area1 = (vcount[9:1] == 9'b000000001) & (hcount[10:1] > 10'd145)
&(hcount[10:1] < 10'd496);
        assign seg_line_area1 = (vcount[9:1] == 9'b000000011) & (hcount[10:1] > 10'd145) &
(hcount[10:1] < 10'd496);

        //Calibration
        logic sprite1on, sprite2on, sprite3on, sprite4on, feedback, feedbackframe, keyboardarea;
        assign sprite1on = (vcount >= 10'd0 & vcount < 10'd340);
        assign sprite2on = (vcount >= 10'd100 & vcount < 10'd182 & hcount[10:1] >= 10'd20 &
hcount[10:1] < 10'd620);
        assign sprite3on = (vcount >= 10'd240 & vcount < 10'd340);
        assign sprite4on = (vcount >= 10'd240 & vcount < 10'd360 & hcount[10:1] >= 10'd275 &
hcount[10:1] < 10'd365);


        assign feedback = (vcount >= 10'd199 & vcount <= 10'd295) & (hcount[10:1] >= 10'd144) &
(hcount[10:1] <= 10'd495);
        assign feedbackframe = (vcount >= 10'd194 & vcount <= 10'd300) & (hcount[10:1] >= 10'd139)
& (hcount[10:1] <= 10'd500);
        assign keyboardarea = (vcount >= 10'd340);

        logic [9:0] feedback_x, feedback_y;
        logic [2:0] feedback_xx;
        assign feedback_x = (hcount[10:1] - 10'd144) >> 4;
        assign feedback_xx = hcount[4:2] - 3'b010;
        assign feedback_y = (vcount[9:0] - 10'd199) >> 1;

        logic [10:0] feedback_index;
        assign feedback_index = feedback_y * 22 + feedback_x;

        logic whitearea, blackarea, lineup, linedown;
        assign whitearea = (vcount >= 10'd360 & vcount < 10'd385);
        assign blackarea = (vcount >= 10'd415);
  assign lineup = (vcount >= 10'd340 & vcount <= 10'd360);
        assign linedown = (vcount >= 10'd450);
```

```
	logic fb1, fb2, fb3, fb4, fb5, fb6, fb7, fb8, fb9, fb10, fb11, fb12, fb13, fb14, fb15, fb16;

	assign fb1 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd80 & hcount <=
11'd150);
  assign fb2 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd150 & hcount <= 11'd220);
  assign fb3 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd220 & hcount <= 11'd290);
  assign fb4 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd290 & hcount <= 11'd360);
  assign fb5 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd360 & hcount <= 11'd430);
  assign fb6 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd430 & hcount <= 11'd500);
  assign fb7 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd500 & hcount <= 11'd570);
  assign fb8 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd570 & hcount <= 11'd640);
  assign fb9 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd640 & hcount <= 11'd710);
  assign fb10 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd710 & hcount <= 11'd780);
  assign fb11 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd780 & hcount <= 11'd850);
  assign fb12 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd850 & hcount <= 11'd920);
  assign fb13 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd920 & hcount <= 11'd990);
  assign fb14 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd990 & hcount <= 11'd1060);
  assign fb15 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd1060 & hcount <= 11'd1130);
  assign fb16 = (vcount >= 10'd320 & vcount < 10'd340) & (hcount > 11'd1130 & hcount <= 11'd1200);

	logic screen_button1, screen_button2, screen_button3, screen_button4, screen_button5;

	assign screen_button1 = (vcount >= 10'd350 & vcount <= 10'd415) & (hcount[10:1] >= 10'd71
& hcount[10:1] <= 10'd169);
	assign screen_button2 = (vcount >= 10'd350 & vcount <= 10'd415) & (hcount[10:1] >= 10'd171
& hcount[10:1] <= 10'd269);
	assign screen_button3 = (vcount >= 10'd350 & vcount <= 10'd415) & (hcount[10:1] >= 10'd271
& hcount[10:1] <= 10'd369);
	assign screen_button4 = (vcount >= 10'd350 & vcount <= 10'd415) & (hcount[10:1] >= 10'd371
& hcount[10:1] <= 10'd469);
	assign screen_button5 = (vcount >= 10'd350 & vcount <= 10'd415) & (hcount[10:1] >= 10'd471
& hcount[10:1] <= 10'd569);

	//Display
	logic line1, line2;
	logic key1, key2, key3, key4, key5, key6, key7, key8, key9, key10, key11, key12, key13, key14,
key15, key16;

  assign line1 = (vcount >= 10'd340) & (hcount >= 11'd0 & hcount <= 11'd80);
	assign line2 = (vcount >= 10'd340) & (hcount > 11'd1200 & hcount <= 11'd1280);
  assign key1 = (vcount >= 10'd340) & (hcount > 11'd80 & hcount <= 11'd150);
  assign key2 = (vcount >= 10'd340) & (hcount > 11'd150 & hcount <= 11'd220);
  assign key3 = (vcount >= 10'd340) & (hcount > 11'd220 & hcount <= 11'd290);
  assign key4 = (vcount >= 10'd340) & (hcount > 11'd290 & hcount <= 11'd360);
  assign key5 = (vcount >= 10'd340) & (hcount > 11'd360 & hcount <= 11'd430);
  assign key6 = (vcount >= 10'd340) & (hcount > 11'd430 & hcount <= 11'd500);
  assign key7 = (vcount >= 10'd340) & (hcount > 11'd500 & hcount <= 11'd570);
```

```
   assign key8 = (vcount >= 10'd340) & (hcount > 11'd570 & hcount <= 11'd640);
   assign key9 = (vcount >= 10'd340) & (hcount > 11'd640 & hcount <= 11'd710);
   assign key10 = (vcount >= 10'd340) & (hcount > 11'd710 & hcount <= 11'd780);
   assign key11 = (vcount >= 10'd340) & (hcount > 11'd780 & hcount <= 11'd850);
   assign key12 = (vcount >= 10'd340) & (hcount > 11'd850 & hcount <= 11'd920);
   assign key13 = (vcount >= 10'd340) & (hcount > 11'd920 & hcount <= 11'd990);
   assign key14 = (vcount >= 10'd340) & (hcount > 11'd990 & hcount <= 11'd1060);
   assign key15 = (vcount >= 10'd340) & (hcount > 11'd1060 & hcount <= 11'd1130);
   assign key16 = (vcount >= 10'd340) & (hcount > 11'd1130 & hcount <= 11'd1200);

	logic autoplay, freemode; //gamemode//;
	assign autoplay = (vcount > 10'd210 & vcount <= 10'd224) & (hcount[10:1] > 11'd165 &
hcount[10:1] <= 11'd500);
	assign freemode = (vcount > 10'd210 & vcount <= 10'd224) & (hcount[10:1] > 11'd217 &
hcount[10:1] <= 11'd423);
	//assign gamemode = (vcount > 10'd210 & vcount <= 10'd224) & (hcount[10:1] > 11'd222 &
hcount[10:1] <= 11'd418);


	logic explaintext;
	logic w_next1, w_color1, w_calibration1, w_confirm, w_no_color, w_calibration2, w_recognize,
w_keys;
	logic w_next2, w_free, w_mode1, w_auto_play, w_mode2, w_exit, w_back_to1;
	logic w_screen, w_calibration3, w_back_to2, w_color3, w_calibration4;



	assign w_next1 = (vcount > 10'd367 & vcount <= 10'd395) & (hcount[10:1] > 11'd275 &
hcount[10:1] <= 11'd363);

	assign w_color1 = (vcount > 10'd366 & vcount <= 10'd380) & (hcount[10:1] > 11'd96 &
hcount[10:1] <= 11'd147);
	assign w_calibration1 = (vcount > 10'd385 & vcount <= 10'd399) & (hcount[10:1] > 11'd72 &
hcount[10:1] <= 11'd169);
	assign w_no_color = (vcount > 10'd366 & vcount <= 10'd380) & (hcount[10:1] > 11'd180 &
hcount[10:1] <= 11'd269);
	assign w_calibration2 = (vcount > 10'd385 & vcount <= 10'd399) & (hcount[10:1] > 11'd172 &
hcount[10:1] <= 11'd269);
	assign w_confirm = (vcount > 10'd375 & vcount <= 10'd389) & (hcount[10:1] > 11'd285 &
hcount[10:1] <= 11'd358);
	assign w_recognize = (vcount > 10'd366 & vcount <= 10'd380) & (hcount[10:1] > 11'd372 &
hcount[10:1] <= 11'd467);
	assign w_keys = (vcount > 10'd385 & vcount <= 10'd399) & (hcount[10:1] > 11'd399 &
hcount[10:1] <= 11'd442);
	assign w_next2 = (vcount > 10'd375 & vcount <= 10'd389) & (hcount[10:1] > 11'd499 &
hcount[10:1] <= 11'd543);
```

```
	assign w_free = (vcount > 10'd366 & vcount <= 10'd380) & (hcount[10:1] > 11'd99 &
hcount[10:1] <= 11'd143);
	assign w_mode1 = (vcount > 10'd385 & vcount <= 10'd399) & (hcount[10:1] > 11'd98 &
hcount[10:1] <= 11'd146);
	assign w_auto_play = (vcount > 10'd366 & vcount <= 10'd380) & (hcount[10:1] > 11'd179 &
hcount[10:1] <= 11'd261);
	assign w_mode2 = (vcount > 10'd385 & vcount <= 10'd399) & (hcount[10:1] > 11'd198 &
hcount[10:1] <= 11'd246);
	//assign w_game = (vcount > 10'd366 & vcount <= 10'd380) & (hcount[10:1] > 11'd297 &
hcount[10:1] <= 11'd344);
	//assign w_mode3 = (vcount > 10'd385 & vcount <= 10'd399) & (hcount[10:1] > 11'd298 &
hcount[10:1] <= 11'd346);
	assign w_exit = (vcount > 10'd375 & vcount <= 10'd389) & (hcount[10:1] > 11'd300 &
hcount[10:1] <= 11'd337);
	assign w_back_to1 = (vcount > 10'd360 & vcount <= 10'd374) & (hcount[10:1] > 11'd386 &
hcount[10:1] <= 11'd456);
	assign w_screen = (vcount > 10'd375 & vcount <= 10'd389) & (hcount[10:1] > 11'd387 &
hcount[10:1] <= 11'd455);
	assign w_calibration3 = (vcount > 10'd390 & vcount <= 10'd404) & (hcount[10:1] > 11'd372 &
hcount[10:1] <= 11'd469);
   assign w_back_to2 = (vcount > 10'd360 & vcount <= 10'd374) & (hcount[10:1] > 11'd486 &
hcount[10:1] <= 11'd556);
	assign w_color2 = (vcount > 10'd375 & vcount <= 10'd389) & (hcount[10:1] > 11'd496 &
hcount[10:1] <= 11'd547);
	assign w_calibration4 = (vcount > 10'd390 & vcount <= 10'd404) & (hcount[10:1] > 11'd472 &
hcount[10:1] <= 11'd569);

	logic b_screen_calibration, b_color_calibration, b_adjust1, b_adjust2;

	assign b_screen_calibration = (vcount >= 10'd50 & vcount < 10'd78) & (hcount[10:1] > 11'd145
& hcount[10:1] <= 11'd495);
	assign b_color_calibration = (vcount >= 10'd200 & vcount < 10'd228) & (hcount[10:1] >
11'd159 & hcount[10:1] <= 11'd481);
	assign b_adjust1 = (vcount >= 10'd140 & vcount < 10'd154) & (hcount[10:1] > 11'd153 &
hcount[10:1] <= 11'd488);
	assign b_adjust2 = (vcount >= 10'd160 & vcount < 10'd174) & (hcount[10:1] > 11'd136 &
hcount[10:1] <= 11'd504);

	logic backbutton;
	assign backbutton = (vcount >= 10'd330 & vcount < 10'd340) & (hcount[10:1] >= 10'd40 &
hcount[10:1] <= 10'd600);

	logic note_display;

	assign note_display = (vcount >= 10'd180 & vcount < 10'd240) & (hcount[10:1] > 10'd290 &
hcount[10:1] <= 10'd350);
```

```
	logic dao,re,mi,fa,so,la,xi;

	assign dao = (vcount >= 10'd196 & vcount < 10'd224) & (hcount[10:1] >= 10'd296 &
hcount[10:1] < 10'd342);
	assign re = (vcount >= 10'd196 & vcount < 10'd224) & (hcount[10:1] >= 10'd298 &
hcount[10:1] < 10'd342);
	assign mi = (vcount >= 10'd196 & vcount < 10'd224) & (hcount[10:1] >= 10'd302 &
hcount[10:1] < 10'd338);
	assign fa = (vcount >= 10'd196 & vcount < 10'd224) & (hcount[10:1] >= 10'd298 &
hcount[10:1] < 10'd342);
	assign so = (vcount >= 10'd196 & vcount < 10'd224) & (hcount[10:1] >= 10'd300 &
hcount[10:1] < 10'd342);
	assign la = (vcount >= 10'd196 & vcount < 10'd224) & (hcount[10:1] >= 10'd300 &
hcount[10:1] < 10'd338);
	assign xi = (vcount >= 10'd196 & vcount < 10'd224) & (hcount[10:1] >= 10'd304 &
hcount[10:1] < 10'd336);


	always_comb begin
		{VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00}; // red

		//Screen Calibration mode
		if ((mode_select == 0) | (mode_select == 1) | (mode_select == 2)) begin
			if (feedback) begin
				if ((vcount - 199) == (row_num - 114) | (vcount - 199) == (row_num - 98)
| (vcount - 199) == (row_num - 130)) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};//green
				else if (pixel[feedback_index][feedback_xx] == 1) {VGA_R, VGA_G,
VGA_B} = {8'hff, 8'hff, 8'h00};//yellow
				else {VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'h00};//red
			end else if (feedbackframe) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff,
8'h00};//green

			else if (b_screen_calibration) begin
				if (bit_screen_calibration[vcount[9:1] - 5'b11001][hcount[10:2] -
7'b1000111] == 1) {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff, 8'h00};//yellow
				else {VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'h00};//red
			end else if (b_adjust1) begin
				if (bit_adjust1[vcount - 140][hcount[10:1] - 154] == 1) {VGA_R,
VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};//green
				else {VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'h00};//red
			end else if (b_adjust2) begin
				if (bit_adjust2[vcount - 160][hcount[10:1] - 137] == 1) {VGA_R,
VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};//green
				else {VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'h00};//red

			end else if (sprite1on) {VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'h00};//red
			else if ((mode_select == 1) & screen_button3) begin
```

```
                                        if (w_next1) begin
                                                  if (bit_next[vcount[9:1] - 8'b10111000][hcount[10:2] -
8'b10001010] == 1) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hff};//blue
                                                  else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark
blue

                                        end else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                              end else if ((mode_select == 2) & screen_button3)  begin

                                        if (w_next1) begin
                                                  if (bit_next[vcount[9:1] - 8'b10111000][hcount[10:2] -
8'b10001010] == 1) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hff};//blue
                                                  else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark
blue

                                        end else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};//green
                              end else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hff};//blue


                    //Keynoard Calibration mode
                    end else if (mode_select == 3) begin
                              if (real_line_area) {VGA_R, VGA_G, VGA_B} = {real_VGA_R, real_VGA_G,
real_VGA_B};
                              else if (seg_line_area) begin
                                        if (pixel_RGB[hcount[10:1] - 10'd146] == 2'b01) {VGA_R, VGA_G,
VGA_B} = {8'hff, 8'h00, 8'h00};
                                        else if (pixel_RGB[hcount[10:1] - 10'd146] == 2'b10) {VGA_R, VGA_G,
VGA_B} = {8'h00, 8'hff, 8'h00};
                                        else if (pixel_RGB[hcount[10:1] - 10'd146] == 2'b11) {VGA_R, VGA_G,
VGA_B} = {8'h00, 8'h00, 8'hff};
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                              end else if (denoise_line_area) begin
                                        if (fix_pixel_RGB_denise[hcount[10:1] - 10'd146] == 2'b01) {VGA_R,
VGA_G, VGA_B} = {8'hff, 8'h00, 8'h00};
                                        else if (fix_pixel_RGB_denise[hcount[10:1] - 10'd146] == 2'b10)
{VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff, 8'h0};
                                        else if (fix_pixel_RGB_denise[hcount[10:1] - 10'd146] == 2'b11)
{VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hff};
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                              end else if (fix_keys_line_area) {VGA_R, VGA_G, VGA_B} =
{fix_keys_position[hcount[10:1] - 10'd146], 3'b111, fix_keys_position[hcount[10:1] - 10'd146], 3'b111,
fix_keys_position[hcount[10:1] - 10'd146], 3'b111};
                              else if (inChar) begin
                                        if ( |(curSegs & segment) ) {VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00,
8'h00}; // Red
```

```
					else if ( |segment ) {VGA_R, VGA_G, VGA_B} = {8'h30, 8'h30, 8'h30};
// Dark Gray
				end else if (lineup | whitearea) {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff,
8'hff};//white
				else if (blackarea) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};//black
				else if (line1|line2) {VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'h00};
				else if (key2 | key4 | key6 | key8 | key10 | key12 | key14 | key16) {VGA_R,
VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};
				else if (key1 | key3 | key5 | key7 | key9 | key11 | key13 | key15) {VGA_R,
VGA_G, VGA_B} = {8'h00, 8'h00, 8'hff};

				else if (b_color_calibration) begin
					if (bit_color_calibration[vcount[9:1] - 7'b1100100][hcount[10:2] -
7'b1010000] == 1) {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff, 8'h00};//yellow
					else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};//black

				end else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};//black

			//Keyboard Calibration select mode
			end else if (mode_select == 4 | mode_select == 5 | mode_select == 6 | mode_select == 7 |
mode_select == 8 | mode_select == 9) begin
				if (real_line_area) {VGA_R, VGA_G, VGA_B} = {real_VGA_R, real_VGA_G,
real_VGA_B};
				else if (seg_line_area) begin
					if (pixel_RGB[hcount[10:1] - 10'd146] == 2'b01) {VGA_R, VGA_G,
VGA_B} = {8'hff, 8'h00, 8'h00};
					else if (pixel_RGB[hcount[10:1] - 10'd146] == 2'b10) {VGA_R, VGA_G,
VGA_B} = {8'h00, 8'hff, 8'h00};
					else if (pixel_RGB[hcount[10:1] - 10'd146] == 2'b11) {VGA_R, VGA_G,
VGA_B} = {8'h00, 8'h00, 8'hff};
					else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
				end else if (denoise_line_area) begin
					if (fix_pixel_RGB_denise[hcount[10:1] - 10'd146] == 2'b01) {VGA_R,
VGA_G, VGA_B} = {8'hff, 8'h00, 8'h00};
					else if (fix_pixel_RGB_denise[hcount[10:1] - 10'd146] == 2'b10)
{VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff, 8'h0};
					else if (fix_pixel_RGB_denise[hcount[10:1] - 10'd146] == 2'b11)
{VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hff};
					else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
				end else if (fix_keys_line_area) {VGA_R, VGA_G, VGA_B} =
{fix_keys_position[hcount[10:1] - 10'd146], 3'b111, fix_keys_position[hcount[10:1] - 10'd146], 3'b111,
fix_keys_position[hcount[10:1] - 10'd146], 3'b111};
				else if (inChar) begin
					if ( |(curSegs & segment) ) {VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00,
8'h00}; // Red
					else if ( |segment ) {VGA_R, VGA_G, VGA_B} = {8'h30, 8'h30, 8'h30};
// Dark Gray
```

```
				end else if (w_color1) begin
					if (bit_color[vcount - 367][hcount[10:1] - 97] == 1) {VGA_R, VGA_G, 
VGA_B} = {8'h00, 8'hff, 8'h00};//green
					else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
				end else if (w_calibration1) begin
					if (bit_calibration[vcount - 386][hcount[10:1] - 73] == 1) {VGA_R, 
VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};//green
					else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
				end else if (w_no_color) begin
					if (bit_no_color[vcount - 367][hcount[10:1] - 181] == 1) {VGA_R, 
VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};//green
					else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
				end else if (w_calibration2) begin
					if (bit_calibration[vcount - 386][hcount[10:1] - 173] == 1) {VGA_R, 
VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};//green
					else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
				end else if (w_confirm) begin
					if (bit_confirm[vcount - 376][hcount[10:1] - 286] == 1) {VGA_R, 
VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};//green
					else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
				end else if (w_recognize) begin
					if (bit_recognize[vcount - 367][hcount[10:1] - 373] == 1) {VGA_R, 
VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};//green
					else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
				end else if (w_keys) begin
					if (bit_keys[vcount - 386][hcount[10:1] - 400] == 1) {VGA_R, VGA_G, 
VGA_B} = {8'h00, 8'hff, 8'h00};//green
					else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
				end else if (w_next2) begin
					if (bit_next[vcount - 376][hcount[10:1] - 500] == 1) {VGA_R, VGA_G, 
VGA_B} = {8'h00, 8'hff, 8'h00};//green
					else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue

				end else if (screen_button1) begin
					if (mode_select == 5) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff, 
8'h00};//green
					else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
				end else if (screen_button2) begin
					if (mode_select == 6) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff, 
8'h00};//green
					else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
				end else if (screen_button3) begin
					if (mode_select == 7) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff, 
8'h00};//green
					else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
				end else if (screen_button4) begin
```

```
                                        if (mode_select == 8) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff,
8'h00};//green
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                                end else if (screen_button5) begin
                                        if (mode_select == 9) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff,
8'h00};//green
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                                end else if (lineup | whitearea) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00,
8'hff};//blue
                                else if (blackarea) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hff};//blue

                                else if (line1|line2) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hff};//blue
                                else if (key2 | key4 | key6 | key8 | key10 | key12 | key14 | key16) {VGA_R,
VGA_G, VGA_B} = {8'h00, 8'h00, 8'hff};
                                else if (key1 | key3 | key5 | key7 | key9 | key11 | key13 | key15) {VGA_R,
VGA_G, VGA_B} = {8'h00, 8'h00, 8'hff};

                                else if (b_color_calibration) begin
                                        if (bit_color_calibration[vcount[9:1] - 7'b1100100][hcount[10:2] -
7'b1010000] == 1) {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff, 8'h00};//yellow
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};//black

                                end else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};//black

                        //Welcome page
                        end else if (mode_select == 10 | mode_select == 11 | mode_select == 12 | mode_select
== 13 | mode_select == 14 | mode_select == 15) begin
                                if (real_line_area1) {VGA_R, VGA_G, VGA_B} = {real_VGA_R, real_VGA_G,
real_VGA_B};
                                else if (seg_line_area1) begin
                                        if (pixel_RGB[hcount[10:1] - 10'd146] == 2'b01) {VGA_R, VGA_G,
VGA_B} = {8'hff, 8'h00, 8'h00};
                                        else if (pixel_RGB[hcount[10:1] - 10'd146] == 2'b10) {VGA_R, VGA_G,
VGA_B} = {8'h00, 8'hff, 8'h00};
                                        else if (pixel_RGB[hcount[10:1] - 10'd146] == 2'b11) {VGA_R, VGA_G,
VGA_B} = {8'h00, 8'h00, 8'hff};
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                          //end

                                end else if (w_free) begin
                                        if (bit_free[vcount - 367][hcount[10:1] - 100] == 1) {VGA_R, VGA_G,
VGA_B} = {8'h00, 8'hff, 8'h00};//green
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                                end else if (w_mode1) begin
                                        if (bit_mode[vcount - 386][hcount[10:1] - 99] == 1) {VGA_R, VGA_G,
VGA_B} = {8'h00, 8'hff, 8'h00};//green
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                                end else if (w_auto_play) begin
                                        if (bit_auto_play[vcount - 367][hcount[10:1] - 180] == 1) {VGA_R,
VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};//green
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                                end else if (w_mode2) begin
                                        if (bit_mode[vcount - 386][hcount[10:1] - 199] == 1) {VGA_R, VGA_G,
VGA_B} = {8'h00, 8'hff, 8'h00};//green
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                                end else if (w_exit) begin
                                        if (bit_exit[vcount - 376][hcount[10:1] - 301] == 1) {VGA_R, VGA_G,
VGA_B} = {8'h00, 8'hff, 8'h00};//green
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                                end else if (w_back_to1) begin
                                        if (bit_back_to[vcount - 360][hcount[10:1] - 387] == 1) {VGA_R,
VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};//green
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                                end else if (w_screen) begin
                                        if (bit_screen[vcount - 376][hcount[10:1] - 388] == 1) {VGA_R, VGA_G,
VGA_B} = {8'h00, 8'hff, 8'h00};//green
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                                end else if (w_calibration3) begin
                                        if (bit_calibration[vcount - 391][hcount[10:1] - 373] == 1) {VGA_R,
VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};//green
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                                end else if (w_back_to2) begin
                                        if (bit_back_to[vcount - 360][hcount[10:1] - 487] == 1) {VGA_R,
VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};//green
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                                end else if (w_color2) begin
                                        if (bit_color[vcount - 376][hcount[10:1] - 497] == 1) {VGA_R, VGA_G,
VGA_B} = {8'h00, 8'hff, 8'h00};//green
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                                end else if (w_calibration4) begin
                                        if (bit_calibration[vcount - 391][hcount[10:1] - 473] == 1) {VGA_R,
VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};//green
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                                end else if (autoplay && mode_select == 12) begin

                                                if (bit_autoplay[vcount - 211][hcount[10:1] - 166] == 1) {VGA_R,
VGA_G, VGA_B} = {8'hff, 8'h00, 8'hff};//pink
                                                else {VGA_R, VGA_G, VGA_B} = {8'hf1, 8'hf1,
8'hbb};//background

                                end else if (freemode && mode_select == 11) begin

                                                if (bit_freemode[vcount - 211][hcount[10:1] - 218] == 1)
{VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'hff};//pink
```

```
                                        else {VGA_R, VGA_G, VGA_B} = {8'hf1, 8'hf1,
8'hbb};//background

                        end else if (screen_button1) begin
                                if (mode_select == 11) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff,
8'h00};//green
                                else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                        end else if (screen_button2) begin
                                if (mode_select == 12) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff,
8'h00};//green
                                else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                        end else if (screen_button3) begin
                                if (mode_select == 13) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff,
8'h00};//green
                                else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                        end else if (screen_button4) begin
                                if (mode_select == 14) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff,
8'h00};//green
                                else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                        end else if (screen_button5) begin
                                if (mode_select == 15) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff,
8'h00};//green
                                else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h8b};//dark blue
                        end else if (sprite2on) {VGA_R, VGA_G, VGA_B} = {in_R2,in_G2,in_B2};
                        else if (sprite3on) {VGA_R, VGA_G, VGA_B} = {in_R3,in_G3,in_B3};
                        else if (keyboardarea) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hff};//blue
                        else {VGA_R, VGA_G, VGA_B} = {8'hf1, 8'hf1, 8'hbb};

                //Freedom mode / Song player mode / Game mode
                end else if ((mode_select == 16) | (mode_select == 17) | (mode_select == 18) |
(mode_select == 19)) begin
                        if (real_line_area1) {VGA_R, VGA_G, VGA_B} = {real_VGA_R, real_VGA_G,
real_VGA_B};
                     else if (seg_line_area1) begin
                                if (pixel_RGB[hcount[10:1] - 10'd146] == 2'b01) {VGA_R, VGA_G,
VGA_B} = {8'hff, 8'h0, 8'h0};
                                else if (pixel_RGB[hcount[10:1] - 10'd146] == 2'b10) {VGA_R, VGA_G,
VGA_B} = {8'h0, 8'hff, 8'h0};
                                else if (pixel_RGB[hcount[10:1] - 10'd146] == 2'b11) {VGA_R, VGA_G,
VGA_B} = {8'h0, 8'h0, 8'hff};
                                else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                        end else if (line1|line2) {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff, 8'h00};
                        else if
(keychar1|keychar3|keychar5|keychar7|keychar9|keychar11|keychar13|keychar15) begin
                                if ( |(keycurSegs & keysegment) ) begin
                                        if (keychar1|keychar3|keychar5|keychar7) {VGA_R, VGA_G,
VGA_B} = {8'hff, 8'hff, 8'h00}; // yellow
```

```
                                        else if (keychar9|keychar11|keychar13) {VGA_R, VGA_G, 
VGA_B} = {8'hff, 8'h8c, 8'h00}; // yellow_2
                                        else {VGA_R, VGA_G, VGA_B} = {8'hff, 8'h69, 8'hb4}; // 
yellow_3
              end else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h99, 8'hff};
                        end else if 
(keychar2|keychar4|keychar6|keychar8|keychar10|keychar12|keychar14|keychar16) begin
                                if ( |(keycurSegs & keysegment) ) begin
                                        if (keychar2|keychar4|keychar6) {VGA_R, VGA_G, VGA_B} = 
{8'hff, 8'hff, 8'h00}; // yellow
                                        else if (keychar8|keychar10|keychar12|keychar14) {VGA_R, 
VGA_G, VGA_B} = {8'hff, 8'h8c, 8'h00}; // yellow_2
                                        else {VGA_R, VGA_G, VGA_B} = {8'hff, 8'h69, 8'hb4}; // 
yellow_3
              end else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hcc, 8'hff};
      end else if (key1) begin
                                if (press[1] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h99, 8'hff};
                          else begin
                                        if (vcount >= 343 & vcount <= 477 & hcount > 83 & hcount <= 
147) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                          end
                        end else if (key2) begin
                          if (press[2] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hcc, 8'hff};
                          else begin
                                        if (vcount >= 343 & vcount <= 477 & hcount > 153 & hcount <= 
217) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                                end
                        end else if (key3) begin
                          if (press[3] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h99, 8'hff};
                          else begin
                                        if (vcount >= 343 & vcount <= 477 & hcount > 223 & hcount <= 
287){VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                                end
                        end else if (key4) begin
                          if (press[4] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hcc, 8'hff};
                          else begin
                                        if (vcount >= 343 & vcount <= 477 & hcount > 293 & hcount <= 
357) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                                end
                        end else if (key5) begin
                          if (press[5] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h99, 8'hff};
                          else begin
```

```
                                        if (vcount >= 343 & vcount <= 477 & hcount > 363 & hcount <=
427) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                                end
                        end else if (key6) begin
                          if (press[6] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hcc, 8'hff};
                          else begin
                                        if (vcount >= 343 & vcount <= 477 & hcount > 433 & hcount <=
497) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                                end
                        end else if (key7) begin
                          if (press[7] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h99, 8'hff};
                          else begin
                                        if (vcount >= 343 & vcount <= 477 & hcount > 503 & hcount <=
567) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                                end
                        end else if (key8) begin
                          if (press[8] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hcc, 8'hff};
                          else begin
                                        if (vcount >= 343 & vcount <= 477 & hcount > 573 & hcount <=
637) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                                end
                        end else if (key9) begin
                          if (press[9] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h99, 8'hff};
                          else begin
                                        if (vcount >= 343 & vcount <= 477 & hcount > 643 & hcount <=
707) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                  else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                                end
                        end else if (key10) begin
                                if (press[10] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hcc, 8'hff};
                          else begin
                                        if (vcount >= 343 & vcount <= 477 & hcount > 713 & hcount <=
777) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                                end
                        end else if (key11) begin
                          if (press[11] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h99, 8'hff};
                          else begin
                                        if (vcount >= 343 & vcount <= 477 & hcount > 783 & hcount <=
847) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                        else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                                end
                        end else if (key12) begin
```

```
                                      if (press[12] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hcc, 8'hff};
                                      else begin
                                                              if (vcount >= 343 & vcount <= 477 & hcount > 853 & hcount <=
917) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                                              else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                                                  end
                                    end else if (key13) begin
                                      if (press[13] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h99, 8'hff};
                                      else begin
                                                              if (vcount >= 343 & vcount <= 477 & hcount > 923 & hcount <=
987) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                                              else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                                                  end
                                    end else if (key14) begin
                                      if (press[14] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hcc, 8'hff};
                                      else begin
                                                              if (vcount >= 343 & vcount <= 477 & hcount > 993 & hcount <=
1057) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                                              else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                                                  end
                                    end else if (key15) begin
                                      if (press[15] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h99, 8'hff};
                                      else begin
                                                              if (vcount >= 343 & vcount <= 477 & hcount > 1063 & hcount <=
1127) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                                              else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                                                  end
                                    end else if (key16) begin
                                      if (press[16] == 0) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'hcc, 8'hff};
                                      else begin
                                                              if (vcount >= 343 & vcount <= 477 & hcount > 1133 & hcount <=
1197) {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'hee};
                                                              else {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};
                                                  end
                                    end else if (backbutton) begin
                                                  if ((mode_select == 17) | (mode_select == 19)) {VGA_R, VGA_G,
VGA_B} = {8'h00, 8'hff, 8'h00};
                                                  else {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff, 8'h00};

                                    end else if (note_display) begin
                                                  {VGA_R, VGA_G, VGA_B} = {in_R1,in_G1,in_B1};
                                                  if (mode_select == 18) begin
                                                              if (press[1] == 1 | press[8] == 1 | press[15] == 1) begin
                                                                          if (dao) begin
                                                                                      if (bit_dao[vcount[9:1] - 7'b1100010][hcount[10:2]
- 8'b10010100] == 1){VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'hff};
```

```
                                                    else {VGA_R, VGA_G, VGA_B} =
{in_R1,in_G1,in_B1}; // background
                                    end
                                        end else if (press[2] == 1 | press[9] == 1 | press[16] == 1) begin
                                            if (re) begin
                                                    if (bit_re[vcount[9:1] - 7'b1100010][hcount[10:2] -
8'b10010101] == 1){VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'hff};
                                                    else {VGA_R, VGA_G, VGA_B} =
{in_R1,in_G1,in_B1}; // background
                                         end
                                        end else if (press[3] == 1 | press[10] == 1) begin
                                            if (mi) begin
                                                    if (bit_mi[vcount[9:1] - 7'b1100010][hcount[10:2] -
8'b10010111] == 1){VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'hff};
                                                    else {VGA_R, VGA_G, VGA_B} =
{in_R1,in_G1,in_B1}; // background
                                         end
                                        end else if (press[4] == 1 | press[11] == 1) begin
                                            if (fa) begin
                                                    if (bit_fa[vcount[9:1] - 7'b1100010][hcount[10:2] -
8'b10010101] == 1){VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'hff};
                                                    else {VGA_R, VGA_G, VGA_B} =
{in_R1,in_G1,in_B1}; // background
                                            end
                                        end else if (press[5] == 1 | press[12] == 1) begin
                                            if (so) begin
                                                    if (bit_so[vcount[9:1] - 7'b1100010][hcount[10:2] -
8'b10010110] == 1){VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'hff};
                                                    else {VGA_R, VGA_G, VGA_B} =
{in_R1,in_G1,in_B1}; // background
                                      end
                                        end else if (press[6] == 1 | press[13] == 1) begin
                                            if (la) begin
                                                    if (bit_la[vcount[9:1] - 7'b1100010][hcount[10:2] -
8'b10010110] == 1){VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'hff};
                                                    else {VGA_R, VGA_G, VGA_B} =
{in_R1,in_G1,in_B1}; // background
                                            end
                                        end else if (press[7] == 1 | press[14] == 1) begin
                                            if (xi) begin
                                                    if (bit_xi[vcount[9:1] - 7'b1100010][hcount[10:2] -
8'b10011000] == 1){VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00, 8'hff};
                                                    else {VGA_R, VGA_G, VGA_B} =
{in_R1,in_G1,in_B1}; // background
                                      end
        end else {VGA_R, VGA_G, VGA_B} = {in_R1,in_G1,in_B1};
                                  end
```

```
                /*end else if (inChar) begin
                        if ( |(curSegs & segment) ) {VGA_R, VGA_G, VGA_B} = {8'hff, 8'h00,
8'h00}; // Red
                        else if ( |segment ) {VGA_R, VGA_G, VGA_B} = {8'h30, 8'h30, 8'h30};
// Dark Gray
                end else if (star_area1) begin
                        //if ((in_R5 == 0) & (in_G5 == 0) & (in_B5 == 0)) {VGA_R, VGA_G,
VGA_B} = {in_R1,in_G1,in_B1};
                        //else
                        if (star_row_current != 0) {VGA_R, VGA_G, VGA_B} =
{in_R4,in_G4,in_B4};
                        else {VGA_R, VGA_G, VGA_B} = {in_R1,in_G1,in_B1};
                */
          end else if (sprite1on) {VGA_R, VGA_G, VGA_B} = {in_R1,in_G1,in_B1};

        //Exit page
        end else if ((mode_select == 20)) begin
                if (sprite4on) begin
                        if (byechange == 0) {VGA_R, VGA_G, VGA_B} =
{in_R4,in_G4,in_B4};
                        else {VGA_R, VGA_G, VGA_B} = {in_R5,in_G5,in_B5};
                end else {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff, 8'hff};
        end
    end
endmodule // VGA_LED_Emulator
```

RGB CONTROLLER

```
module rgb_controller(
input logic clk,
input logic  [9:0]    h,
input logic  [9:0]    c,
input logic  [23:0]   M_bg1,
input logic  [23:0]   M_bg2,
input logic  [23:0]   M_bg3,
input logic  [23:0]   M_bg4,
input logic  [23:0]   M_bg5,
output logic [7:0]        R1,
output logic [7:0]        G1,
output logic [7:0]        B1,
output logic [7:0]        R2,
output logic [7:0]        G2,
output logic [7:0]        B2,
output logic [7:0]        R3,
output logic [7:0]        G3,
output logic [7:0]        B3,
output logic [7:0]        R4,
output logic [7:0]          G4,
output logic [7:0]          B4,
output logic [7:0]          R5,
output logic [7:0]          G5,
output logic [7:0]          B5,
output logic  [15:0]        addr_bg1,
output logic  [15:0]        addr_bg2,
output logic  [15:0]        addr_bg3,
output logic  [15:0]        addr_bg4,
output logic  [15:0]        addr_bg5
);
assign addr_bg1 = c * 128 + h % 128;
assign addr_bg2 = (c - 100) * 600 + h - 20;
assign addr_bg3 = (c - 240) * 128 + h % 128;
assign addr_bg4 = (c - 240) * 90 + h - 275;
assign addr_bg5 = (c - 240) * 90 + h - 275;

assign R1 = M_bg1[23:16];
assign R2 = M_bg2[23:16];
assign R3 = M_bg3[23:16];
assign R4 = M_bg4[23:16];
assign R5 = M_bg5[23:16];

assign G1 = M_bg1[15:8];
assign G2 = M_bg2[15:8];
assign G3 = M_bg3[15:8];
assign G4 = M_bg4[15:8];
assign G5 = M_bg5[15:8];

assign B1 = M_bg1[7:0];
assign B2 = M_bg2[7:0];
assign B3 = M_bg3[7:0];
assign B4 = M_bg4[7:0];
assign B5 = M_bg5[7:0];

endmodule
```

KEYCHAR DISPLAY

```
module KEYCHAR_DISP(input logic [2:0]  offset,
                    input logic [2:0]  digit,
                    input logic [9:0]  vcount,
                    input logic [10:0] hcount,

                            output logic [7:0] keycurSegs, keysegment);

  logic [2:0]               charx; // Coordinate within the 8x16 char
  logic [3:0]               chary;
  assign charx = hcount[3:1] - offset;
  assign chary = vcount[3:0];

  logic horizBar, leftCol, rightCol, topCol, botCol; // Parts of the disp.

  assign horizBar = !(charx[2:1] == 2'b11);  // When in any horizontal bar
  assign leftCol  = (charx == 3'd0);         // When in left column
  assign rightCol = (charx == 3'd5);         // When in right column
  assign topCol   = !chary[3] & !(chary[2:0] == 3'd7); // Top columns
  assign botCol   = (chary >= 4'd6) & (chary <= 4'd12); // Bottom columns

  assign keysegment[0] = horizBar & (chary == 4'd 0);
  assign keysegment[1] = rightCol & topCol;
  assign keysegment[2] = rightCol & botCol;
  assign keysegment[3] = horizBar & (chary == 4'd 12);
  assign keysegment[4] = leftCol & botCol;
  assign keysegment[5] = leftCol & topCol;
  assign keysegment[6] = horizBar & (chary == 4'd 6);
  assign keysegment[7] = (charx == 3'd6) & (chary == 4'd14);

  logic [7:0]  hex;
        assign hex = digit == 4'd0 ? 8'h3f :
                                          digit == 4'd1 ? 8'h06 :
                                          digit == 4'd2 ? 8'h5b :
                                          digit == 4'd3 ? 8'h4f :
                                          digit == 4'd4 ? 8'h66 :
                                          digit == 4'd5 ? 8'h6d :
                                          digit == 4'd6 ? 8'h7d :
                                          digit == 4'd7 ? 8'h07 :
                                          digit == 4'd8 ? 8'h7f :
                                          8'h6f;

  assign keycurSegs = hex;

endmodule
```

HEX CONVERTER

```
module HEX_CONVERT(input logic [8:0]  digit_1, digit_2,
            input logic [9:0]  vcount,
                                              input logic [10:0] hcount,

            output logic [7:0] curSegs, segment);

  logic [3:0] digit1, digit2, digit3, digit4, digit5, digit6;
  assign digit1 = digit_1 / 100;
  assign digit2 = (digit_1 - digit1 * 100) / 10;
  assign digit3 = digit_1 - digit1 * 100 - digit2 * 10;
          assign digit4 = digit_2 / 100;
  assign digit5 = (digit_2 - digit4 * 100) / 10;
  assign digit6 = digit_2 - digit4 * 100 - digit5 * 10;

          logic [2:0]                              charx; // Coordinate within the 8x16 char
  logic [3:0]                              chary;

  assign charx = hcount[3:1];
  assign chary = vcount[3:0];

  logic horizBar, leftCol, rightCol, topCol, botCol; // Parts of the disp.

  assign horizBar = !(charx[2:1] == 2'b11);  // When in any horizontal bar
  assign leftCol  = (charx == 3'd0);         // When in left column
  assign rightCol = (charx == 3'd5);         // When in right column
  assign topCol   = !chary[3] & !(chary[2:0] == 3'd7); // Top columns
  assign botCol   = (chary >= 4'd6) & (chary <= 4'd12); // Bottom columns

  assign segment[0] = horizBar & (chary == 4'd 0);
  assign segment[1] = rightCol & topCol;
  assign segment[2] = rightCol & botCol;
  assign segment[3] = horizBar & (chary == 4'd 12);
  assign segment[4] = leftCol & botCol;
  assign segment[5] = leftCol & topCol;
  assign segment[6] = horizBar & (chary == 4'd 6);
  assign segment[7] = (charx == 3'd6) & (chary == 4'd14);

  logic [2:0] column; // Being displayed
  assign column = hcount[6:4];

  logic [7:0]  hex1, hex2, hex3, hex4, hex5, hex6;
          assign hex1 = digit1 == 4'd0 ? 8'h3f :
                                                      digit1 == 4'd1 ? 8'h06 :
                                                      digit1 == 4'd2 ? 8'h5b :
                                                      digit1 == 4'd3 ? 8'h4f :
                                                      digit1 == 4'd4 ? 8'h66 :
                                                      digit1 == 4'd5 ? 8'h6d :
                                                      digit1 == 4'd6 ? 8'h7d :
                                                      digit1 == 4'd7 ? 8'h07 :
                                                      digit1 == 4'd8 ? 8'h7f :
                                                      8'h6f;

          assign hex2 = digit2 == 4'd0 ? 8'h3f :
                                                      digit2 == 4'd1 ? 8'h06 :
                                                      digit2 == 4'd2 ? 8'h5b :
```

```
                                          digit2 == 4'd3 ? 8'h4f :
                                          digit2 == 4'd4 ? 8'h66 :
                                          digit2 == 4'd5 ? 8'h6d :
                                          digit2 == 4'd6 ? 8'h7d :
                                          digit2 == 4'd7 ? 8'h07 :
                                          digit2 == 4'd8 ? 8'h7f :
                                          8'h6f;

        assign hex3 = digit3 == 4'd0 ? 8'h3f :
                                          digit3 == 4'd1 ? 8'h06 :
                                          digit3 == 4'd2 ? 8'h5b :
                                          digit3 == 4'd3 ? 8'h4f :
                                          digit3 == 4'd4 ? 8'h66 :
                                          digit3 == 4'd5 ? 8'h6d :
                                          digit3 == 4'd6 ? 8'h7d :
                                          digit3 == 4'd7 ? 8'h07 :
                                          digit3 == 4'd8 ? 8'h7f :
                                          8'h6f;

        assign hex4 = digit4 == 4'd0 ? 8'h3f :
                                          digit4 == 4'd1 ? 8'h06 :
                                          digit4 == 4'd2 ? 8'h5b :
                                          digit4 == 4'd3 ? 8'h4f :
                                          digit4 == 4'd4 ? 8'h66 :
                                          digit4 == 4'd5 ? 8'h6d :
                                          digit4 == 4'd6 ? 8'h7d :
                                          digit4 == 4'd7 ? 8'h07 :
                                          digit4 == 4'd8 ? 8'h7f :
                                          8'h6f;

        assign hex5 = digit5 == 4'd0 ? 8'h3f :
                                          digit5 == 4'd1 ? 8'h06 :
                                          digit5 == 4'd2 ? 8'h5b :
                                          digit5 == 4'd3 ? 8'h4f :
                                          digit5 == 4'd4 ? 8'h66 :
                                          digit5 == 4'd5 ? 8'h6d :
                                          digit5 == 4'd6 ? 8'h7d :
                                          digit5 == 4'd7 ? 8'h07 :
                                          digit5 == 4'd8 ? 8'h7f :
                                          8'h6f;

        assign hex6 = digit6 == 4'd0 ? 8'h3f :
                                          digit6 == 4'd1 ? 8'h06 :
                                          digit6 == 4'd2 ? 8'h5b :
                                          digit6 == 4'd3 ? 8'h4f :
                                          digit6 == 4'd4 ? 8'h66 :
                                          digit6 == 4'd5 ? 8'h6d :
```

```
                                       digit6 == 4'd6 ? 8'h7d :
                                       digit6 == 4'd7 ? 8'h07 :
                                       digit6 == 4'd8 ? 8'h7f :
                                       8'h6f;

   assign curSegs = column == 3'd1 ? hex1 :
                          column == 3'd2 ? hex2 :
                          column == 3'd3 ? hex3 :
                          column == 3'd4 ? 8'hb9 :
                          column == 3'd5 ? hex4 :
                          column == 3'd6 ? hex5 :
                          column == 3'd7 ? hex6 :
                          8'hf7;

endmodule
```

Piano Control

```
/*
 * Avalon memory-mapped peripheral for the VGA LED Emulator
 *
 * Stephen A. Edwards
 * Columbia University
 */

module PIANO_CONTROL(input logic         clk,
                                          input logic          reset,
                                          input logic [7:0]   writedata,
                                          input logic          write,
                                          input                    chipselect,
                                          input logic [10:0]  address,
                                          input logic [3:0]   SW,
                                          input logic [3:0]   KEY,


                                          output logic [7:0]  VGA_R, VGA_G, VGA_B,
                                          output logic         VGA_CLK, VGA_HS,
VGA_VS, VGA_BLANK_n,
                                          output logic         VGA_SYNC_n,

//AUD/////////////////////////////////////////////////////////////////////////////////////
                    inout AUD_ADCLRCK,
                                          input AUD_ADCDAT,
                                          inout AUD_DACLRCK,
                                          output AUD_DACDAT,
                                                            output AUD_XCK,
                                                            inout AUD_BCLK,
                                                            output AUD_I2C_SCLK,
                                                            inout AUD_I2C_SDAT,
                                                            output AUD_MUTE
                                                            );

        //pixel: 0~1055
        //base line: 1056~1057
        //mode select: 1058
        //audio demo: 1059
        //game control: 1060
        //audio player: 1061~1062
  logic [7:0] pixel [0:1057];
        logic [7:0] mode_select;
        //mode  0: Screen Calibration running
        //mode  1: Screen Calibration done
        //mode  2: Screen Calibration button3 pressed to Color Calibration mode
        //mode  3: Color Calibration running
        //mode  4: Color Calibration function select
        //mode  5: Color Calibration button1 pressed to redo Color Calibration
        //mode  6: Color Calibration button2 pressed to display Original pixel
        //mode  7: Color Calibration button3 pressed to denoise pixel
        //mode  8: Color Calibration button4 pressed to recognize keys
        //mode  9: Color Calibration button5 pressed to Welcome page
        //mode 10: Welcome page
        //mode 11: Welcome page button1 pressed to Freedom mode
        //mode 12: Welcome page button2 pressed to Song player mode
        //mode 13: Welcome page button3 pressed to Game mode
        //mode 14: Welcome page button4 pressed to Screen Calibration mode
        //mode 15: Welcome page button5 pressed to Color Calibration mode
        //mode 16: Freedom mode
        //mode 17: Freedom mode all button pressed to back to Welcome page
        //mode 18: Song player mode
        //mode 19: Song player mode all button pressed to back to Welcome page
        //mode 20: Exit page
        logic [7:0] audio_demo;
        logic [7:0] drop_position;
        logic [7:0] speed;

        initial begin
                mode_select = 0;
        end

        logic [1:0] pixel_RGB [0:351];
        logic [4:0] fix_keys_position [0:351];
        logic [1:0] fix_pixel_RGB [0:351];
        logic [1:0] fix_pixel_RGB_denise [0:351];

        logic [4:0] key_num;
        logic [16:0] press;
        logic [16:0] press1;
        logic [16:0] press2;
        logic [7:0] R1, G1, B1;
        logic [7:0] R2, G2, B2;
        logic [7:0] R3, G3, B3;
        logic [7:0] R4, G4, B4;
        logic [7:0] R5, G5, B5;
        logic [9:0] star_offset;
        logic [5:0] star_row_current;
        integer i;
  logic [7:0] row_num_1, row_num_2;
        logic [8:0] row_num;
        logic [15:0]  addr1, addr2, addr3, addr4, addr5;
  logic [23:0] pixel1, pixel2, pixel3, pixel4, pixel5;
  logic [9:0] hcount, vcount;
        logic       clk_vga;

        assign row_num = row_num_1 + row_num_2;

        rom newrom1(.clock(clk), .address(addr1), .q(pixel1));
        ROM1 newrom2(.clock(clk), .address(addr2), .q(pixel2));
        rom2 newrom3(.clock(clk), .address(addr3), .q(pixel3));
        ROM_bye1 newrom4(.clock(clk), .address(addr4), .q(pixel4));
        ROM_bye2 newrom5(.clock(clk), .address(addr5), .q(pixel5));
        rgb_controller controller_1
(.clk(clk), .h(hcount), .c(vcount),.addr_bg1(addr1), .M_bg1(pixel1), .addr_bg2(addr2), .M_bg2(pixel2), .
addr_bg3(addr3), .M_bg3(pixel3), .addr_bg4(addr4), .M_bg4(pixel4), .addr_bg5(addr5), .M_bg5(pixel5
), .*);

  PIANO_DISPLAY
display(.clk50(clk), .in_R1(R1), .in_G1(G1), .in_B1(B1), .in_R2(R2), .in_G2(G2), .in_B2(B2),.in_R3(R
3), .in_G3(G3), .in_B3(B3), .in_R4(R4), .in_G4(G4), .in_B4(B4), .in_R5(R5), .in_G5(G5), .in_B5(B5),
 .h(hcount), .v(vcount), .*);
        SEGMENT seg(.clk50(clk), .*);
        RECOGNIZE_KEY recog(.clk50(clk), .*);
        DENOISE deno(.clk50(clk), .*);
        MULTI_KEY multikey (.clk50(clk), .press(press), .press1(press1), .press2(press2));


        logic [16:0]  press_real;
        FINGER_DETECTION finger(.clk50(clk), .press(press_real), .*);
        PRESS pre(.clk50(clk), .*);
  always_ff @(posedge clk)
    if (reset)
                     begin
                           for (i = 0; i < 1056; i = i + 1)
                                 begin
                                   pixel[i] <= 8'd255;
                                 end
                     end
            else if (chipselect && write)
              begin
                     if (address == 11'd1056)
                           begin
                             row_num_1 <= writedata;
                       end
                           else if (address == 11'd1057)
                             begin
                               row_num_2 <= writedata;
                       end
                           else if (address == 11'd1058)
                             begin
                               mode_select <= writedata;
                       end
                           else if (address == 11'd1059)
                             begin
                               audio_demo <= writedata;
                       end
                           else
                             begin
                               pixel[address] <= writedata;
                       end
              end

          always_ff @(posedge clk) begin
                 if (mode_select == 3) fix_pixel_RGB <= pixel_RGB;
          end

//AUD///////////////////////////////////////////////////////////////////////////////////////
  wire AUD_reset = !KEY[2];
          wire main_clk;
          wire audio_clk;
          wire clk27;
          wire [1:0] sample_end;
          wire [1:0] sample_req;
          wire [15:0] audio_output;
          wire [15:0] audio_input;
          wire [15:0] audio_output1;
          wire [15:0] audio_output2;
```

```
clock_pll pll (
                    .refclk (clk),
                    .rst (AUD_reset),
                    .outclk_0 (main_clk),
                    .outclk_1 (audio_clk),
                    .outclk_2 (clk27)
);

i2c_av_config av_config (
                    .clk (main_clk),
                    .reset (AUD_reset),
                    .i2c_sclk (AUD_I2C_SCLK),
                    .i2c_sdat (AUD_I2C_SDAT),
);

assign AUD_XCK = audio_clk;
assign AUD_MUTE = 1;

assign audio_output = (audio_output1 + audio_output2)/2;

audio_codec ac (
                              .clk (audio_clk),
                              .reset (AUD_reset),
                              .sample_end (sample_end),
                              .sample_req (sample_req),
                              .audio_output (audio_output),
                              .audio_input (audio_input),
                              .channel_sel (2'b10),
                              .AUD_ADCLRCK (AUD_ADCLRCK),
                              .AUD_ADCDAT (AUD_ADCDAT),
                              .AUD_DACLRCK (AUD_DACLRCK),
                              .AUD_DACDAT (AUD_DACDAT),
                              .AUD_BCLK (AUD_BCLK)
);
karplus_note kar1 (
                              .clock50 (clk27),
                              .audiolrclk (AUD_DACLRCK),
                              .reset (AUD_reset),
                              .audio_output (audio_output1),
                              .audio_input (audio_input),
                              .control (SW),
                              .press (press1)
);
karplus_note kar2 (
                              .clock50 (clk27),
                              .audiolrclk (AUD_DACLRCK),
                                        .reset (AUD_reset),
                                        .audio_output (audio_output2),
                                        .audio_input (audio_input),
                                        .control (SW),
                                        .press (press2)
          );
  //AUD////////////////////////////////////////////////////////////////////////////////


endmodule
```

Denoise

```
module DENOISE(input logic         clk50,
          input logic [7:0]   mode_select,
         input logic [1:0]   fix_pixel_RGB [0:351],

         output logic [1:0]  fix_pixel_RGB_denise [0:351]);

         integer i;

         initial
    begin
           i = 1;
                  fix_pixel_RGB_denise[0] = fix_pixel_RGB[0];
                  fix_pixel_RGB_denise[351] = fix_pixel_RGB[351];
         end

        always_ff @(posedge clk50)
    begin
                 if (mode_select == 7)
                  begin
                        i <= i + 1;
                        if (i == 350)
                                    i <= 1;

                        if ((fix_pixel_RGB[i] != fix_pixel_RGB[i - 1]) & (fix_pixel_RGB[i] !=
fix_pixel_RGB[i + 1]))
                              fix_pixel_RGB_denise[i] <= fix_pixel_RGB[i - 1];
                        else
                              fix_pixel_RGB_denise[i] <= fix_pixel_RGB[i];
                       end
        end

endmodule
```

Finger Detection

```
module FINGER_DETECTION(input logic                     clk50,
                        input logic [4:0]       fix_keys_position [0:351],
                        input logic [1:0]               pixel_RGB [0:351],

                        output logic [16:0]     press
                                                        );
logic [4:0] position;
logic [4:0] red_num [0:16];
logic [4:0] red_num_stable [0:16];
integer i;
integer j;

always_ff @(posedge clk50)
begin
        i <= i + 1;
        if (i == 351) begin
                red_num_stable <= red_num;
                i <= 0;
                red_num[0] <= 0;
                red_num[1] <= 0;
                red_num[2] <= 0;
                red_num[3] <= 0;
                red_num[4] <= 0;
                red_num[5] <= 0;
                red_num[6] <= 0;
                red_num[7] <= 0;
                red_num[8] <= 0;
                red_num[9] <= 0;
                red_num[10] <= 0;
                red_num[11] <= 0;
                red_num[12] <= 0;
                red_num[13] <= 0;
                red_num[14] <= 0;
                red_num[15] <= 0;
                red_num[16] <= 0;
        end
        if (pixel_RGB[i] == 2'b01 && fix_keys_position[i] != 5'd17) begin
                position <= fix_keys_position[i];
                red_num[position] <= red_num[position] + 1;
        end

        for (j = 1; j < 17; j = j + 1)
        begin
                if (red_num_stable[j] > 7) begin
                        press[j] <= 1;
                end else begin
```

```
                        press[j] <= 0;
                end
        end
end

endmodule
```

Recognize keys

```
module RECOGNIZE_KEY(input logic        clk50,
                     input logic [7:0]  mode_select,
                     input logic [1:0]  fix_pixel_RGB_denise [0:351],

                     output logic [4:0] fix_keys_position [0:351],
                     output logic [4:0] key_num);

        integer i;
        initial
   begin
           i = 0;
           key_num = 5'd17;
         end

        always_ff @(posedge clk50)
   begin
                if (mode_select == 8)
                  begin
                        i <= i + 1;
                        fix_keys_position[i] <= key_num;
                        if (i == 351)
                                i <= 0;

                        if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 17))
                                key_num <= 5'd1;
                        else if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 1))
                                key_num <= 5'd1;
                        else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 1))
                                key_num <= 5'd2;
                        else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 2))
                                key_num <= 5'd2;
                        else if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 2))
                                key_num <= 5'd3;
                        else if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 3))
                                key_num <= 5'd3;
                        else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 3))
                                key_num <= 5'd4;
                        else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 4))
```

```
                    key_num <= 5'd4;
else if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 4))
                    key_num <= 5'd5;
else if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 5))
                    key_num <= 5'd5;
else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 5))
                    key_num <= 5'd6;
else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 6))
                    key_num <= 5'd6;
else if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 6))
                    key_num <= 5'd7;
else if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 7))
                    key_num <= 5'd7;
else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 7))
                    key_num <= 5'd8;
else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 8))
                    key_num <= 5'd8;
else if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 8))
                    key_num <= 5'd9;
else if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 9))
                    key_num <= 5'd9;
else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 9))
                    key_num <= 5'd10;
else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 10))
                    key_num <= 5'd10;
else if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 10))
                    key_num <= 5'd11;
else if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 11))
                    key_num <= 5'd11;
else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 11))
                    key_num <= 5'd12;
else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 12))
                    key_num <= 5'd12;
else if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 12))
                    key_num <= 5'd13;
else if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 13))
                    key_num <= 5'd13;
else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 13))
                    key_num <= 5'd14;
else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 14))
                    key_num <= 5'd14;
else if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 14))
                    key_num <= 5'd15;
else if ((fix_pixel_RGB_denise[i] == 2'b11) & (key_num == 15))
                    key_num <= 5'd15;
else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 15))
                    key_num <= 5'd16;
                    else if ((fix_pixel_RGB_denise[i] == 2'b10) & (key_num == 16))
                              key_num <= 5'd16;
                    else if ((fix_pixel_RGB_denise[i] == 2'b01) & (key_num == 16))
                              key_num <= 5'd17;
                   end
      end

endmodule
```

Segmentation

```
module SEGMENT(input logic        clk50,
               input logic                reset,
               input logic [7:0]   pixel [0:1057],

               output logic [1:0]  pixel_RGB [0:351]);

      integer i;

      always_comb begin
       for (i = 0; i < 352; i = i + 1)
         begin
               if (((pixel[i * 3 + 1] > 130) & (pixel[i * 3 + 0] > 130)) | ((pixel[i * 3 + 0] > pixel[i * 3 +
1]) & (pixel[i * 3 + 0] > pixel[i * 3 + 2])))
                 begin
                      pixel_RGB[i] = 2'b01;  //red
                    end
               else if ((pixel[i * 3 + 2] > pixel[i * 3 + 0]) & (pixel[i * 3 + 2] > pixel[i * 3 + 1]))
                 begin
                      pixel_RGB[i] = 2'b11;  //blue
                    end
               else if ((pixel[i * 3 + 1] > pixel[i * 3 + 0]) & (pixel[i * 3 + 1] > pixel[i * 3 + 2]))
                 begin
                      pixel_RGB[i] = 2'b10;  //green
                    end
               else
                 begin
                      pixel_RGB[i] = 2'b00;  //others, red
                    end
              end
      end

endmodule
```