# Virtual Conducting

6.111: Introductory Digital Systems Laboratory

Andy Lin and Brandon Yoshimoto

December 11, 2006

**Abstract**

The purpose of this project was to design and implement an interactive music player which allows the user to control the sound of a composition through hand movements. The idea is to emulate the experience of a conductor directing the flow of a musical performance. The design uses a camera to detect hand movements of the user which are then analyzed to adjust musical qualities of tempo, dynamics, and articulation in response to these motions.

The volume of high frequency audio content is controlled by the user's right hand, while the volume of low frequency content is controlled by the user's left hand to allow control over the balance of the playback. The design includes a screen which displays a visualization of the hand movements, including colored squares following the path of the conductor's hands. The screen also displays current tempo, volume, and acceleration magnitudes to provide feedback for the user as he conducts.

## Contents

## List of Figures

## List of Tables

# 1. Overview

The Virtual Conducting system is an interactive music player which allows the user to control the sound of a composition through hand movements. The user stands in front of a camera with two bright blue LEDs, one in each hand. As the user moves his hands, he divides the music into different beats which are matched with the audio to allow control over tempo. Additionally, volume of the high frequency audio is controlled by the size of the right hand's movement, while the volume of low frequencies are controlled by the left hand. The system is divided into two main components: video and audio, as seen in Figure 1.

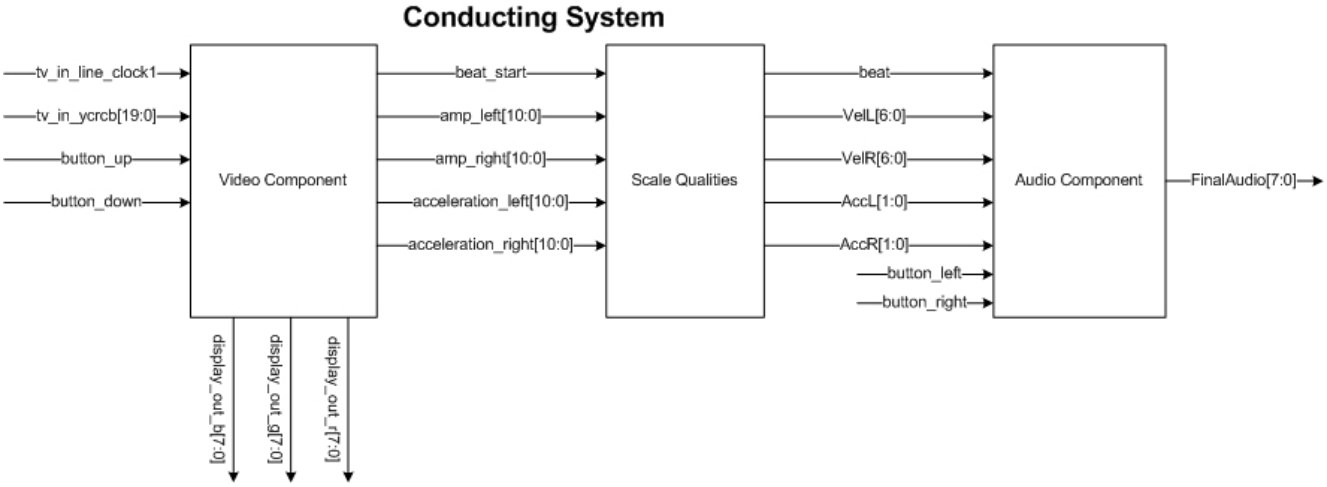


Figure 1: Organization of the Virtual Conducting System

For video, a camera is used to detect the position of the two bright blue LEDs held by the user. The position of each hand in the 1024x768 VGA coordinate system is determined for each frame of the video display. These coordinates are analyzed over time to determine when a user has started and ended each beat. Beats are determined by the user's right hand only. A beat starts when the current right hand position moves out of a bounded area surrounding the coordinates registered upon a beat end. A beat end is registered if the user's hand remains within certain bounds on the screen for a certain amount of time. This condition essentially examines the speed of the user's motion and detects the end of a beat when the user's motion is too slow. Once the user's motion is divided in to beats, qualities of amplitude, period, and acceleration of each beat are calculated for output to the audio component. While beat period is calculated based on just the right hand, amplitude and acceleration calculations are done for both the left and right hand to control the volume of low and high frequencies respectively.

Figure 2: The working Virtual Conducting System

The monitor is used to display the current position of the user's hands, coordinates upon a beat start and end, as well as a motion analyzer display which contains bars that change width in proportion to the magnitude of each motion quality. Figure 3 is a sample image of the visualization. A more detailed description of each part of the display follows in the Module Description section.

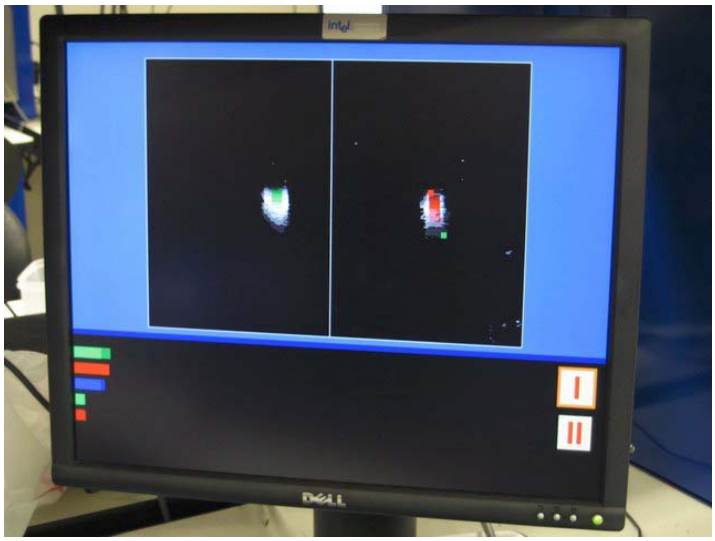


Figure 3: A screen capture of the visualization

The audio component uses qualities of the user's motion generated in the video component to adjust the playback of the audio. If a fast beat period is specified by the user through the Video Component, there will be a fast playback; conversely, if a slow

beat period is specified by the user, there will be a slow playback. Moreover, the gesture amplitude of both hands will determine the loudness and articulation of the audio playback. Also, the left hand will control the volume and articulation of the bass, while the right hand will control the treble. Between the video and audio components is a module which translates the values from the video component into valid scaled input values for the audio component.

The audio source is from the flash ROM. The natural beat period of this music can be programmed using the buttons on the FPGA's. In order to program the beat period, button3 must be depressed, while the 8-bit switches are set to their intended positions. In order to program the initial offset, button2 and button3 must be depressed, while the 8-bit switches are set to their intended positions. To reset the beat period and offset to their default values, button1 is pressed. To reset the audio back to the first sample in the flash ROM, button0 is pressed. The output from the Audio Component is through an AC '97 interface; the audio is 8-bit 24 KHz audio.

# 2. Module Description and Implementation

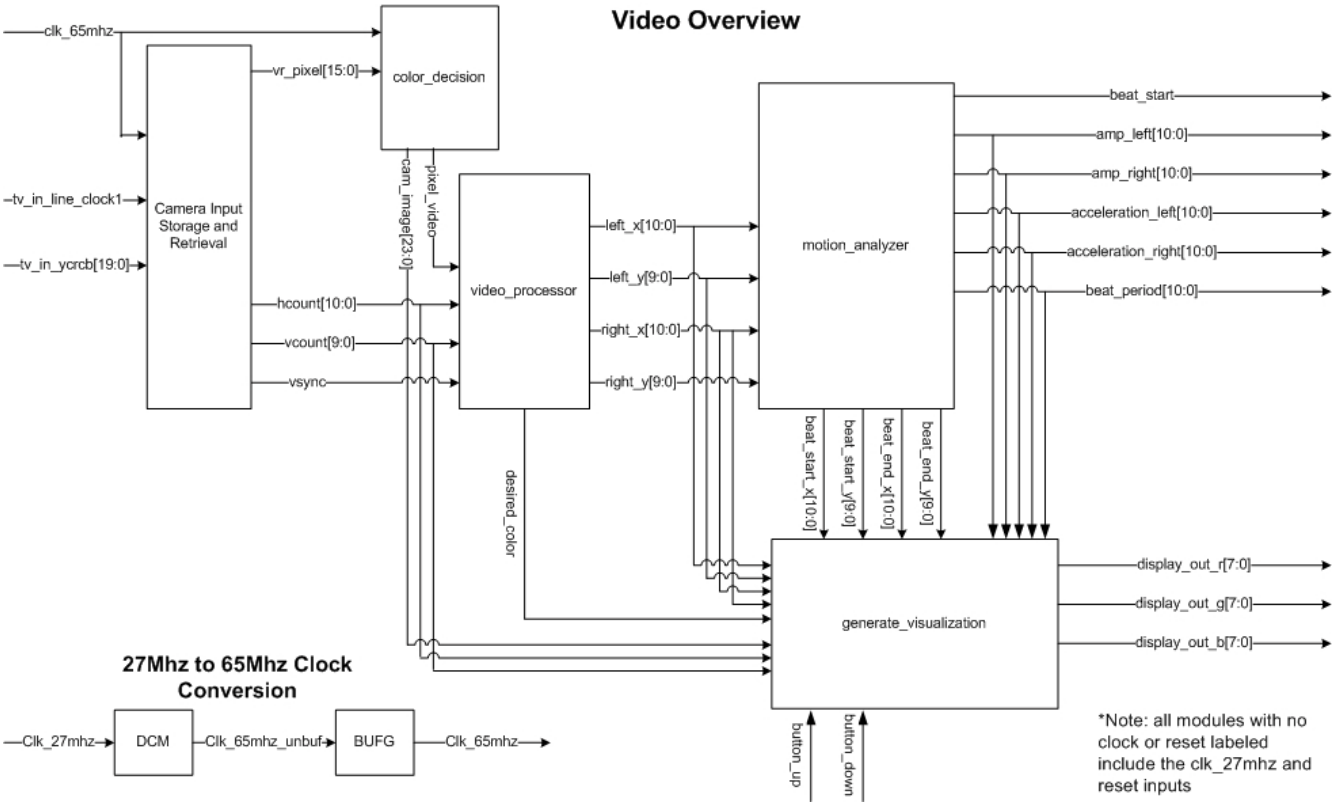


Figure 4: Block diagram overview of Video Component

## 2.1 Video Component Overview (Brandon)

The Video Component of the project consists of five main parts: the camera input storage and retrieval, color_decision, video_processor, motion_analyzer, and generate_visualization modules, as arranged in Figure 4. On a general level, this portion of the project first analyzes data from the camera input to determine the user's hand positions. Once this information is found in the video_processor, the motion_analyzer examines the movement of the hands over time to determine the start and end of a beat. It also determines qualities of the motion for sending to the Audio Component of the project. Finally, the generate_visualization module takes in information from the other modules to create the video for display on the monitor. Each module is described in further detail below.

## 2.2 Camera Input Storage and Retrieval *(Brandon)*

The Camera Input Storage and Retrieval block encompasses the modules which are used for storing the incoming stream of camera data in the ZBT and reading out the contents for use in video processing and visualization. The modules in this component were taken from the sample code on the 6.111 website and modified for this project. The block diagram is outlined in Figure 5.

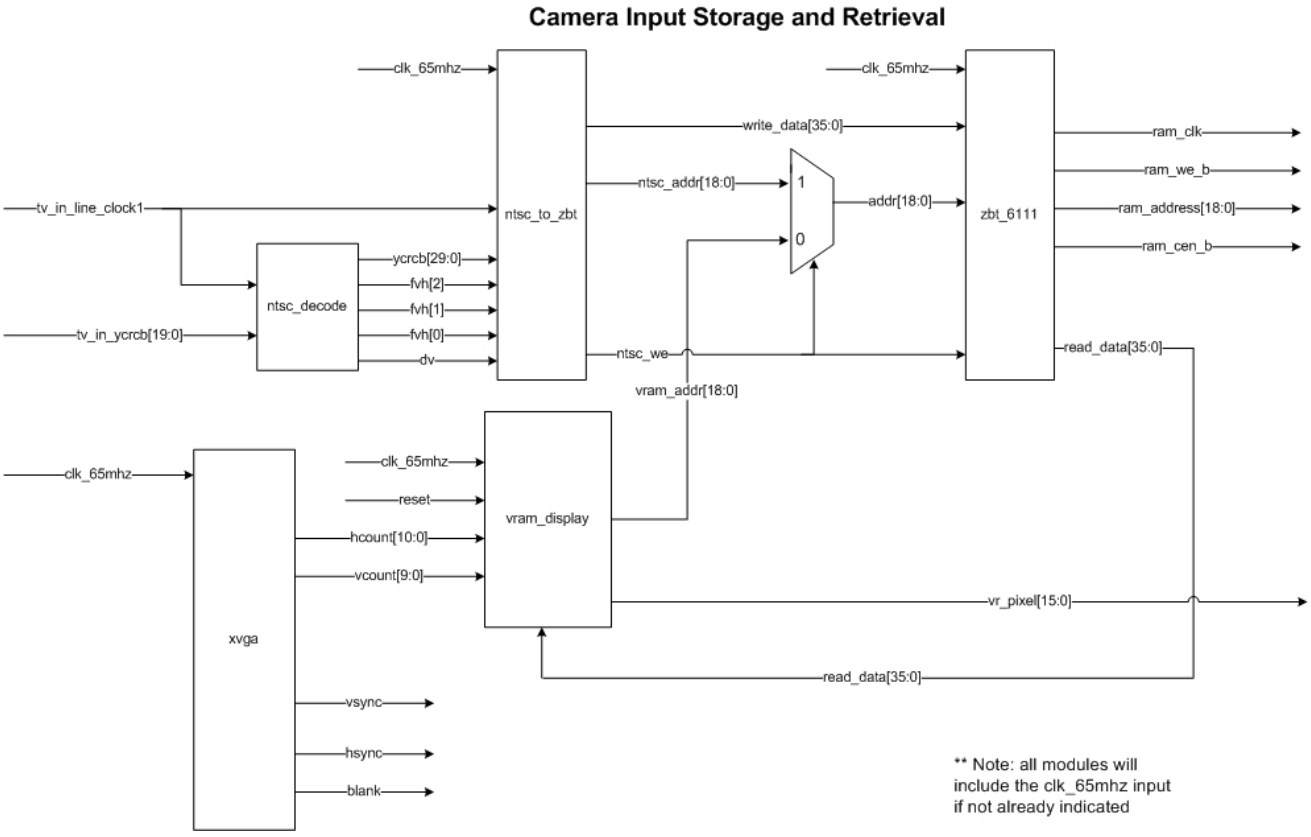


Figure 5: Camera Input Storage and Retrieval block diagram

Firstly, the ntsc_decode module uses the *tv_in_ycrcb[19:0]* signal to generate the *ycrcb[29:0]* signal. The *ycrcb[29:0]* signal contains a full 30-bit representation of the camera input steam in YCrCb format. The ntsc_to_zbt takes this stream of incoming data and stores 16 bits of information for each incoming pixel in the ZBT in the following format: {4'b0, highest 6 Y bits, highest 5 Cr bits, highest 5 Cb bits}. The sample code was modified from four 8-bit pixels per location to handle the storage of two 16-bit blocks of pixel data per location instead. This amount of information is sufficient for color detection and for display of fairly accurate color video. Addresses for storage are generated such that each encodes a pixel's x and y position, allowing for easy lookup of any particular pixel for display on the monitor. Reading and writing from the ZBT is handled by the *ntsc_we* output from the ntsc_to_zbt module.

The xvga module generates the necessary *hcount[10:0]*, *vcount[9:0]*, *vsync*, *hsync*, and *blank* signals for use in the display of 1024x768 video. These signals are used in the vram_display module which reads raw data from the ZBT and parses it into a stream of 16-bit pixel data in YCrCb format suitable for use in video display. Since two pixels are stored per location, the module holds on to data from the same location for two clock cycles and separates the two pixels' color information to produce the *vr_pixel[15:0]* output. Another necessary modification of the sample code was to flip the camera image along the y-axis to display the video as if looking into a mirror. This is done in the accessing stage of the pixel data by negating the hcount bits used in

constructing the read address. The flipped image allows for better visual feedback for the user in controlling movement.

All modules used in this storage and retrieval operate on a 65 MHz clock except the ntsc_decode module, which operates on the clock from the camera, denoted as tv_in_line_clock1. The ntsc_to_zbt module handles the interaction between the camera clock and the 65 Mhz clock, ensuring that the correct information from the camera is stored in the ZBT.

### 2.3 Color Decision Module (color_decision.v) *(Brandon)*

The color_decision module determines whether or not a pixel from the camera is of the desired blue color. It also outputs a 24-bit RGB representation of the current pixel.

To determine if a pixel is of the desired blue color, the module does threshold tests on the stored Cb and Cr data. If *Cb[4:0]* is greater than or equal to 18 and *Cr[4:0]* is less than or equal to 16, the *color_found* signal will go high. Otherwise, the signal will be low, indicating the blue color is not detected. This threshold was tested to be the best for filtering out dark blues and optimizing detection of the bright blue lights held by the user. The *color_found* signal is tied to the output *pixel_video* for use in the video_processing module to display detected blue areas on the monitor.

Additionally, the color_decision module uses the YCrCb2RGB module provided by Xilinx to convert from the YCrCb color space to the RGB color space. Since the VGA display requires RGB content, this component was necessary for displaying color video. This information is encoded in the *cam_image* output as {8 bits of R, 8 bits of G, 8 bits of B}.

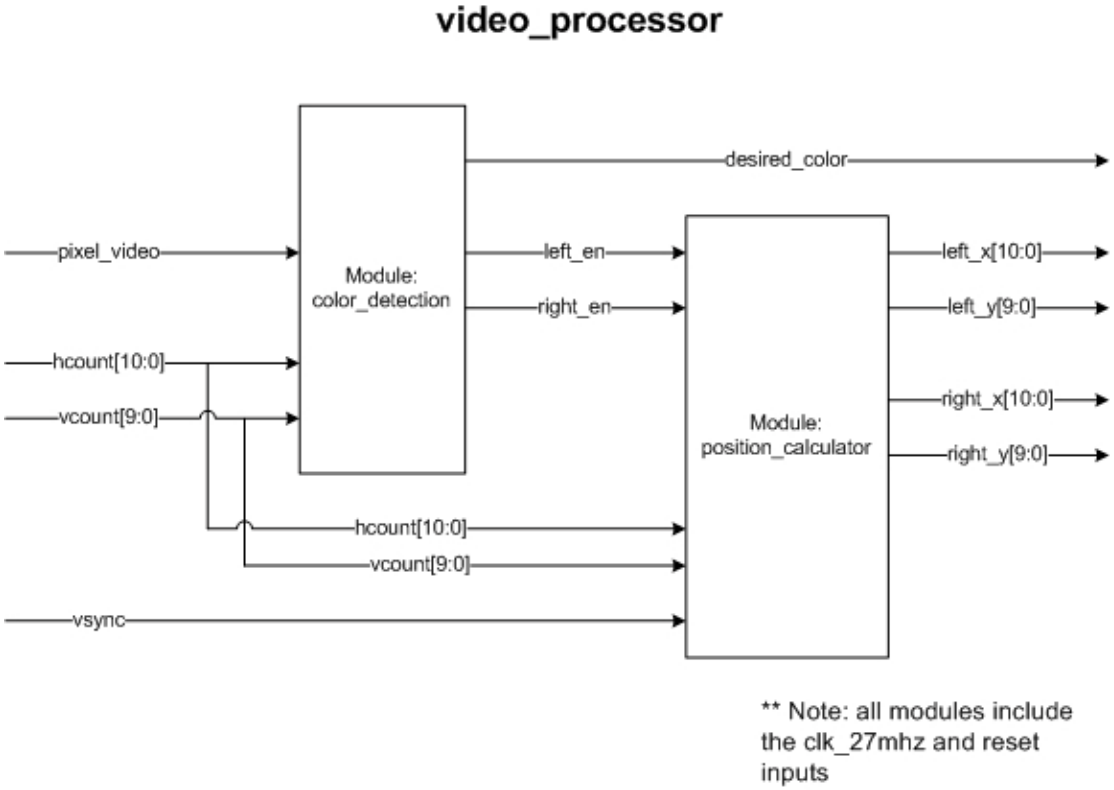


Figure 6: Video Processor block diagram

### 2.4 Video Processor Module (video_processor.v) *(Brandon)*

The video_processor module takes in the *pixel_video* input from the color_decision module and outputs the average coordinates of each hand as 11-bit x and 10-bit y coordinates. The outputs are defined as *left_x[10:0]* for the x-position of the left hand and *left_y[9:0]* for the y-position of the left hand. This same notation is used for the right hand. Calculation of these positions is divided into two modules: color_detection and position_calculator. The organization of this module is described in Figure 6.

#### 2.4.1 Color Detection Module (color_detection.v) *(Brandon)*

The color_detection module determines if a pixel should be used in the calculation of a hand's position. This module has two main functions: one is to decide which half of the screen the detected pixel is in, and the other is to reduce the noise of random pixels detected by the color_decision module that should not be included as part of the hand.

The error reduction function is implemented using shift registers to compare the *pixel_video* values across three consecutive samples. The temporary wire *desired_color_temp* will be high only if *pixel_video* is high for two consecutive pixels. This method detects the user's blue lights quite well while reducing the amount of noisy pixels which could affect average position calculations.

Additionally, *left_side* and *right_side* signals are used to determine if the current pixel is inside the left or right half of the region of display, divided along its center. The display region includes only the camera display window as defined in Figure 12. Combining these two tests, the *left_en* output is high only if both *left_side* and *desired_color_temp* are high, corresponding to a pixel which withstands the error correction test and is in the left hand plane of the screen. The *right_en* output is generated in the same way, but uses the *right_side* signal. The *desired_color* output is tied to the *desired_color_temp* signal for use in displaying the detected pixels in the visualization module.

#### 2.4.2 Position Calculator (position_calculator.v) *(Brandon)*

The position_calculator module uses the *left_en* and *right_en* signals from the color_detection module to calculate of the weighted average hand positions. On a basic level, the module uses four weighted_average modules to compute the average x and y positions for each hand, as seen in Figure 7. However, the position_calculator first applies some tests to the averages calculated by the weighted_average modules before updating the positions for output.

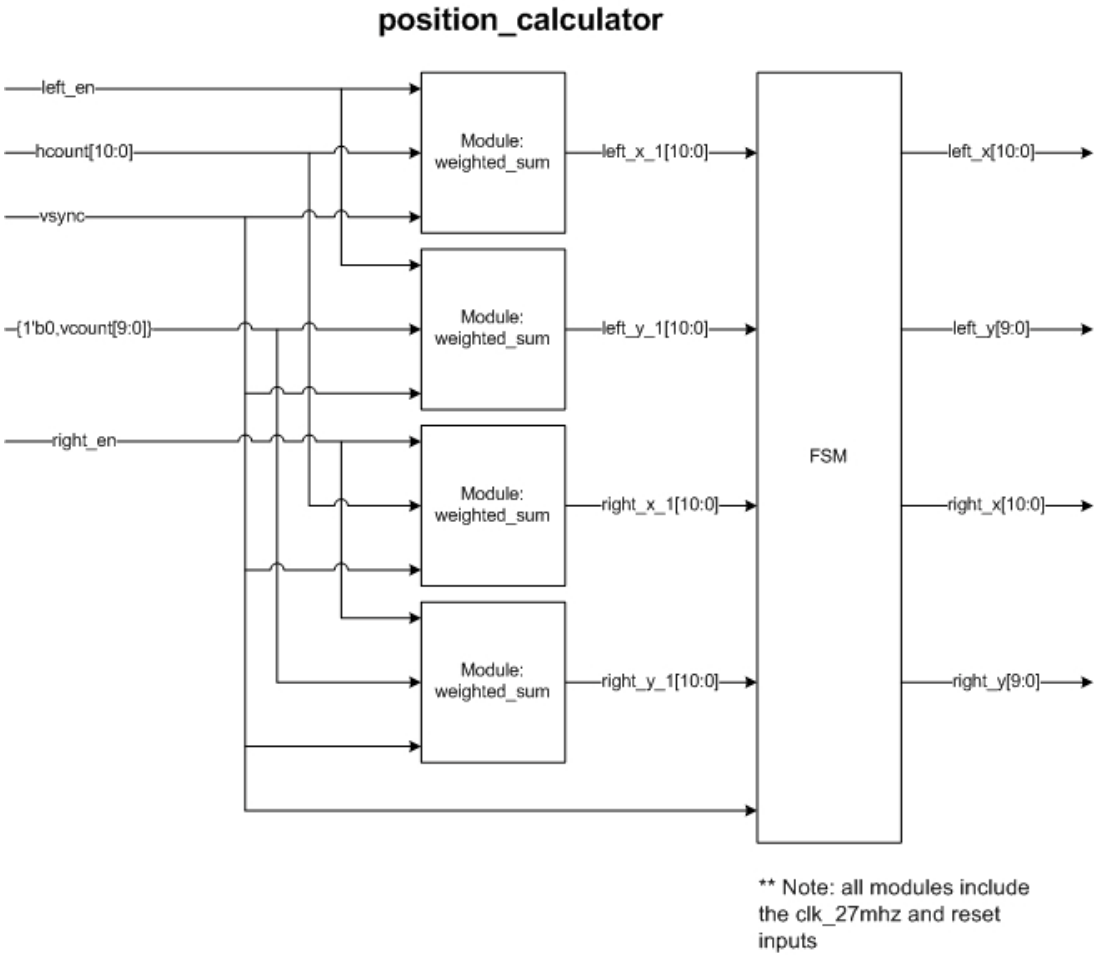


Figure 7: Position Calculator block diagram

The *left_in_left_side*, and *right_in_right_side* signals are used to prevent the left and right hand positions from going out of their respective halves of the camera display. The *left_in_left_side* signal is high when the calculated weighted average coordinates for the left hand are within the appropriate bounds for the left half. The *right_in_right_side* signal is similarly high when the right hand is within the appropriate bounds for the right side.

A second test checks if the new coordinates are sufficiently close to the currently stored coordinates. This test prevents large jumps in the coordinate positions to provide smoother motion. If the new coordinates are within a 150 x 150 pixel square centered at the current left hand coordinates, *no_left_jump* will be high. The same applies for *no_right_jump* on the right hand.

The third test detects if the current pixels are outside of the camera video frame. The coordinates upon starting the system are all initialized to 0, so both *outside_frame_left* and *outside_frame_right* will start off high.

Using these three test signals, the coordinates for the registers holding the output coordinates for the left hand will only update if the new coordinates are in the left hand plane and there is either no large jump in position or the current coordinates are outside of the frame. This is summarized as the condition: (left_in_left_side && (no_left_jump ||

outside_frame_left)). The same applies to the equivalent right hand signals for updating the right hand coordinates.

The potential coordinate update occurs only when a new frame starts at the rising edge of the *vsync* signal. Figure 8 shows the two-state FSM used for coordinating the update of coordinates.

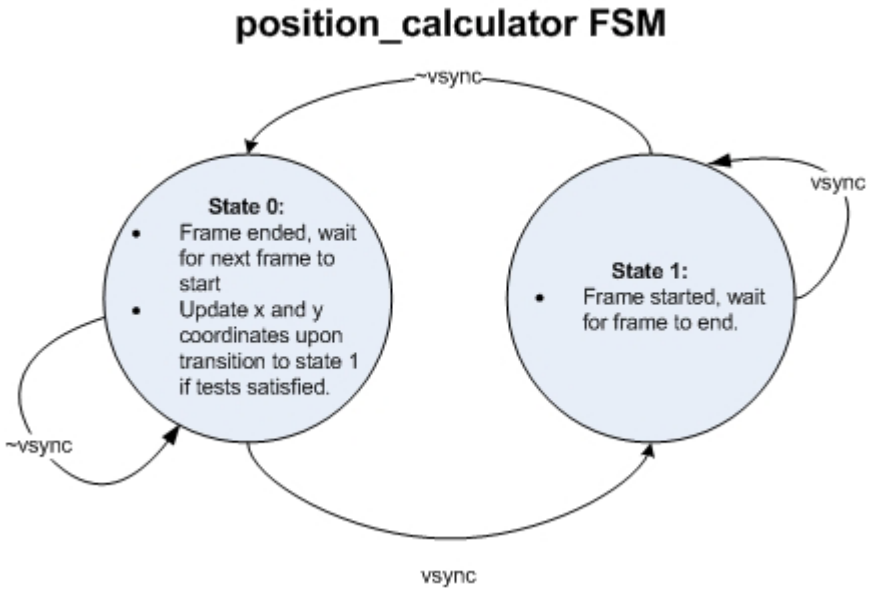


Figure 8: FSM for the Position Calculator module

### 2.4.3 Weighted Average Calculator (weighted_sum.v) *(Brandon)*

The weighted_average module takes signals *enable* and *count[10:0]* as inputs to calculate the average of the count values received when *enable* is high. The *sum[27:0]* registers are used to hold a running sum of the count values, only adding new value if *enable* is high. This running sum calculation begins as soon as *vsync* goes high. *Pixel_count[17:0]* keeps track of how many times *enable* goes high.

To calculate the average, *sum/pixel_count* is calculated using the Xilinx Pipelined Divider v3.0. The lower 11 bits of the divider result are tied to the output *avg*.

Since divider module requires 28 clock cycles to compute, the module is enabled as soon as the current *vcount* is beyond the lower border of the camera image, defined as when *vcount* > BOTTOM_BORDER. The clear is done through changing the *sclr* input to the divider. This timing provides more than enough clocks at 27 MHz to complete the divide calculation before a new frame starts.

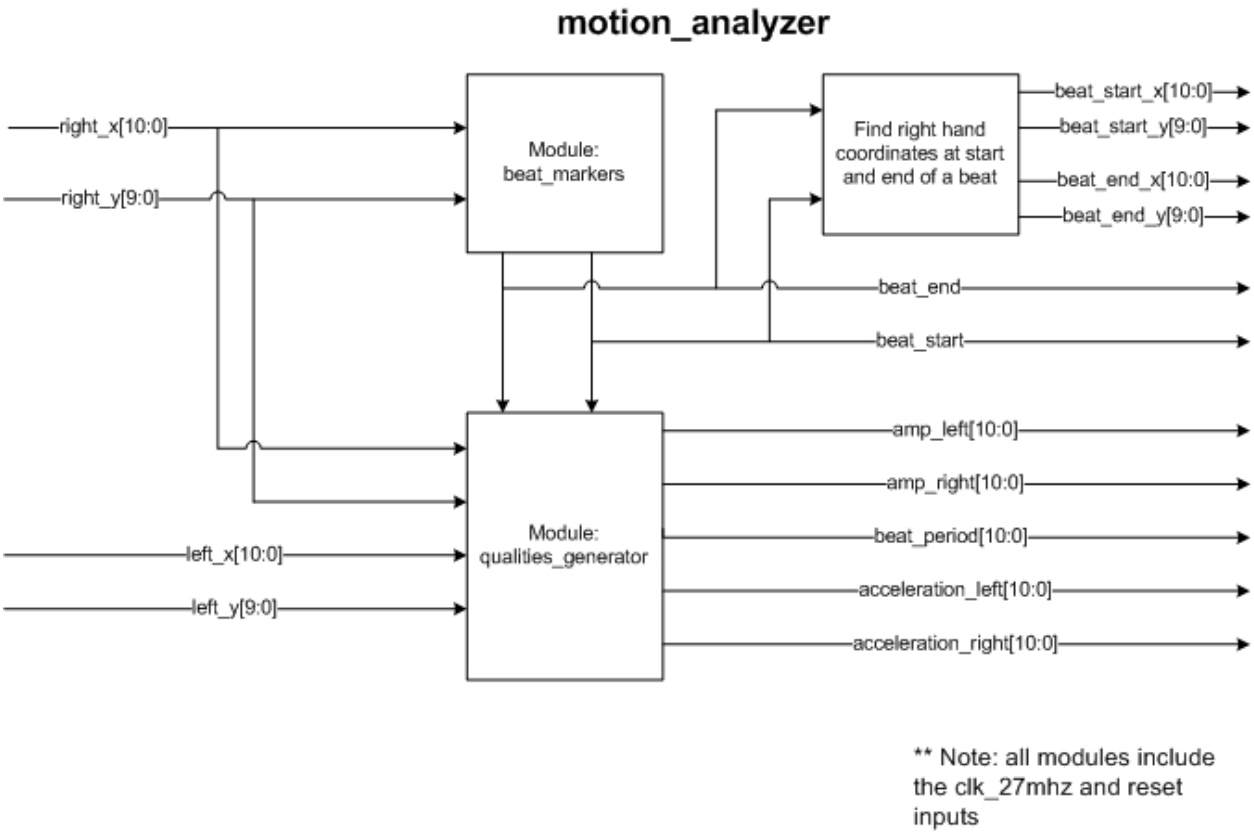


Figure 9: Motion Analyzer block diagram

### 2.5 Motion Analyzer Module (motion_analyzer.v) *(Brandon)*

The motion_analyzer module generates the qualities of beat amplitude, period, and acceleration by analyzing the movement of each hand's x and y hand coordinates over time. The computations are divided into two submodules: the beat_markers and qualities_generator modules, as seen in Figure 9.

The beat_markers module decides when a beat starts and end. The qualities_generator uses this beat marker information to determine the amplitude, period, and acceleration calculations for each beat. More detailed descriptions of these modules are in their respective sections that follow.

The other outputs of the motion_analyzer module are the x and y coordinates of each hand at the start and end of a beat. These coordinates are used for displaying the beat start and end points in the generate_visualization module. The beat start coordinates are stored in registers that are updated whenever *beat_start* goes high. Similarly, the end coordinates are held in registers updated when *beat_end* goes high.

#### 2.5.1 Beat Marker Generator (beat_markers.v) *(Brandon)*

The beat_markers module determines the start and end of a beat based on the movement of the right hand over time. Since beats are only formed by the motion of the right hand, *right_x[10:0]* and *right_y[9:0]* are the only inputs to this module. There are two outputs: *beat_start*, and *beat_end. Beat_start* will go high for one clock cycle upon detection of the start of a beat, while *beat_end* will go high for one clock cycle at the end of a beat.

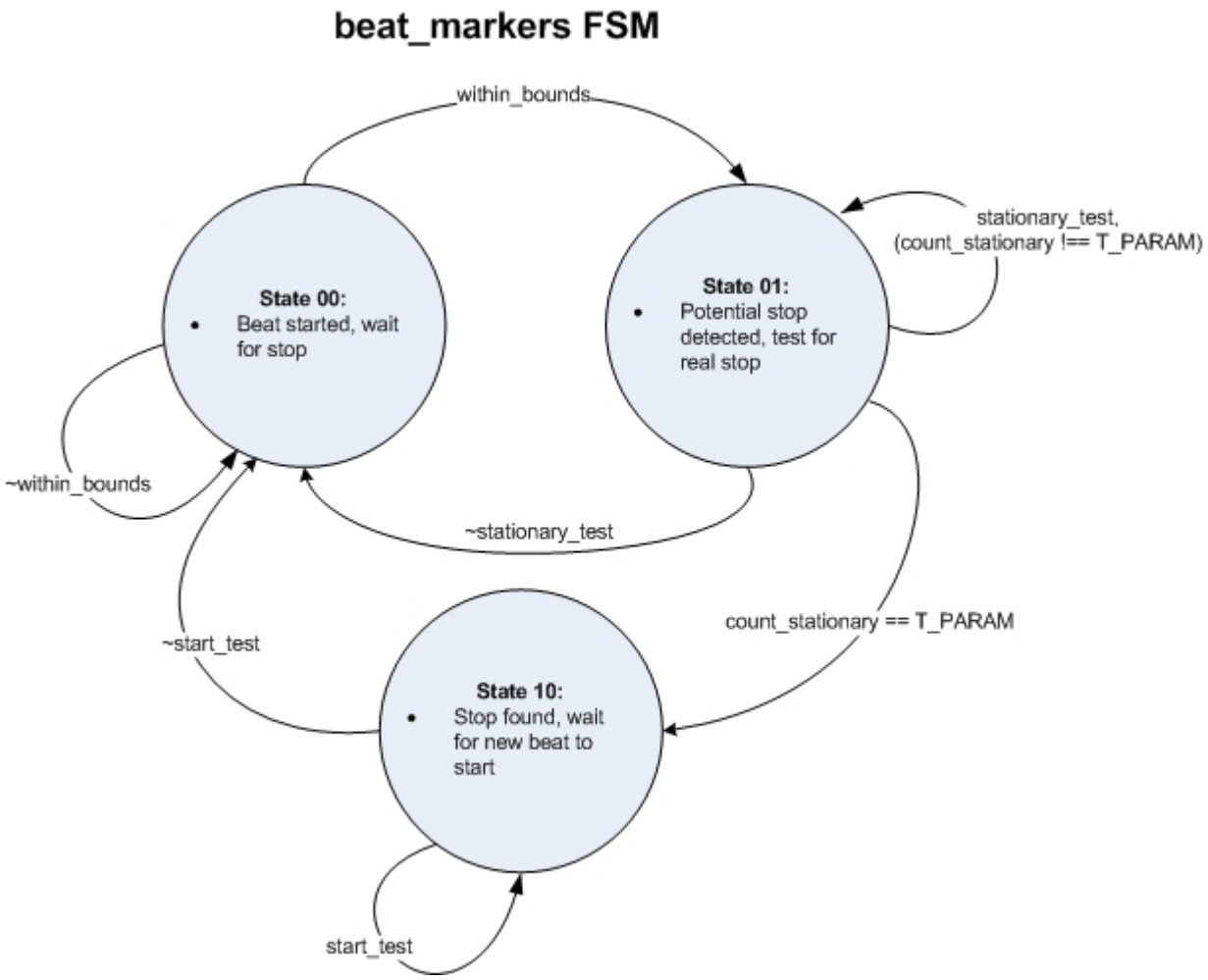

Figure 10: FSM for generating beat markers

Management of the start and end states of a beat is handled by a three state FSM diagrammed in Figure 10. State 00 is when a beat already started and the system is waiting for a stop signal. The stop signal is called *within_bounds*. This signal goes high when the current pixel coordinates are within a centered box of 2*TOLERANCE_X and 2*TOLERANCE_Y of the previous pixel. Assertion high indicates a potential beat stop and changes the FSM to state 01.

In state 01, the module tests if the potential beat stop should be registered as an actual stop. Upon transition to this state, the FSM stored the x and y coordinates of the right hand in registers for use in the next test called *stationary_test*. *Stationary_test* goes high if the current frame's coordinates remain for several clocks within a centered box of 2*TOLERANCE_END_X and 2*TOLERANCE_END_Y of the coordinates stored upon transition. The length in time that this wait test must remain active is a parameter which affects sensitivity towards how long the hand must remain within appropriate bounds to be registered as a beat end. In this implementation, *stationary_test* must be valid for 1,048,575 clock cycles at 27 MHz to move to state 10 and detect a beat end. This corresponds to a wait of about 0.039 seconds. If the coordinates go outside of this bound within that time frame, the FSM will return to state 00 to wait for the next potential

beat end. If the test holds, the *beat_end* output will go high for one clock cycle upon transition to state 10.

In state 10, a stop has already been detected, so the FSM will remain in this state until a new beat is detected. A new beat is detected if the *start_test* signal goes low. *Start_test* goes low when the current coordinates are outside an area bounded by the centered box of 2*TOLERANCE_START_X by 2*TOLERANCE_START_Y around the stored coordinates upon leaving state 01. The coordinates leaving this bound means the hand has moved enough to be considered the start of a new beat. Upon *start_test*, the FSM will transition to state 00 and the *beat_start* output will go high for one clock cycle.

The parameters in this module were tested and set for the user to have best control over generating beats while conducting. The values used in the final implementation are listed in Table 1.

| Parameter: | Value: |
|---|---|
| TOLERANCE_X | 2 |
| TOLERANCE_Y | 2 |
| TOLERANCE_END_X | 5 |
| TOLERANCE_END_Y | 5 |
| TOLERANCE_START_X | 25 |
| TOLERANCE_START_Y | 25 |

Table 1: Parameter values used in beat_markers module for best operation

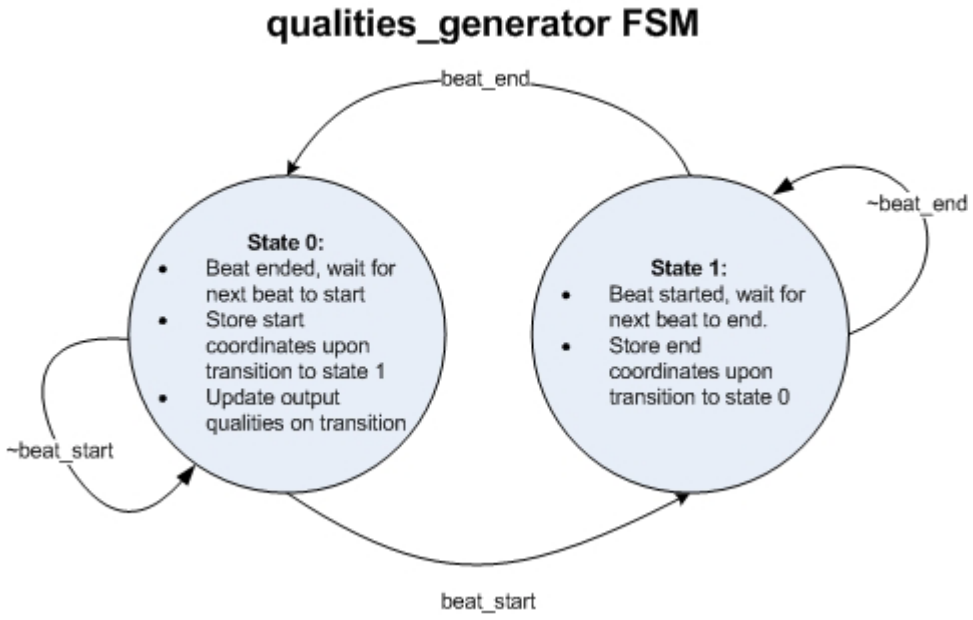


Figure 11: FSM for handling the update of beat motion qualities

### 2.5.2 Qualities Generator Module (qualities_generator.v) *(Brandon)*

The qualities_generator module uses the *beat_start* and *beat_end* signals determined by the beat_markers module to calculate amplitude, time duration, and acceleration values of the motion every beat. A two-state FSM is used in this module to coordinate the update of these qualities with the video display as seen in Figure 11.

In state 0, a beat has ended and the system is waiting for a new beat to start, corresponding to when *beat_start* goes high. Upon receiving a beat start signal, FSM transitions to state 1 and stores the start coordinates of each hand in registers. In state 1, the system is now waiting for a beat to end. When *beat_end* goes high, the FSM transitions back to state 0 and stores the coordinates upon transition into registers. These registers are used to calculate the distance in pixels between the start and end of a beat. This calculation is done for each hand. The left hand distance is used for updating the *amp_left[10:0]* output, while the right hand distance is used for updating the *amp_right[10:0]* output. Distance calculation is done in the find_distance module described in the section that follows.

For beat period and acceleration calculation, a signal called *count[18:0]* is used to divide the clock into roughly 0.02 second divisions, or 2^19 clock cycles at 27 Mhz. Every time this time period passes, the *time_count[10:0]* registers will increment by one to count the time spent in state 1 waiting for a beat to end.

For acceleration calculation, shift registers are used to compute the difference of distances between three successive coordinate samples. These three coordinate points are roughly 0.02 seconds apart from each other and are stored only when the *time_count[10:0]* signal is less than or equal to 6. This means that only coordinates at the beginning of a new beat are used for calculations of a single beat's acceleration. The stored values in the shift registers are used to find the difference in distances for each hand using a total of four find_distance modules (two for each hand). The results are rough acceleration approximations for each hand.

Each of these calculated qualities are only updated for output upon the start of a new beat, as detected in the FSM when *beat_start* goes high while in state 0.

### 2.5.3 Find Distance Module (find_distance.v) *(Brandon)*

The find_distance module takes in two coordinates and outputs the distance in pixels between the two points. This calculation is done by first calculating the difference in x coordinates and y coordinates between the two points, ensuring that the difference is positive. Next, each difference is multiplied by itself to get the squared values. The two results are added together to form the *sum_of_squares[19:0]* wire. The Xilinx square root Core module is used to calculate the square root of this value to produce the *distance[10:0]* output.

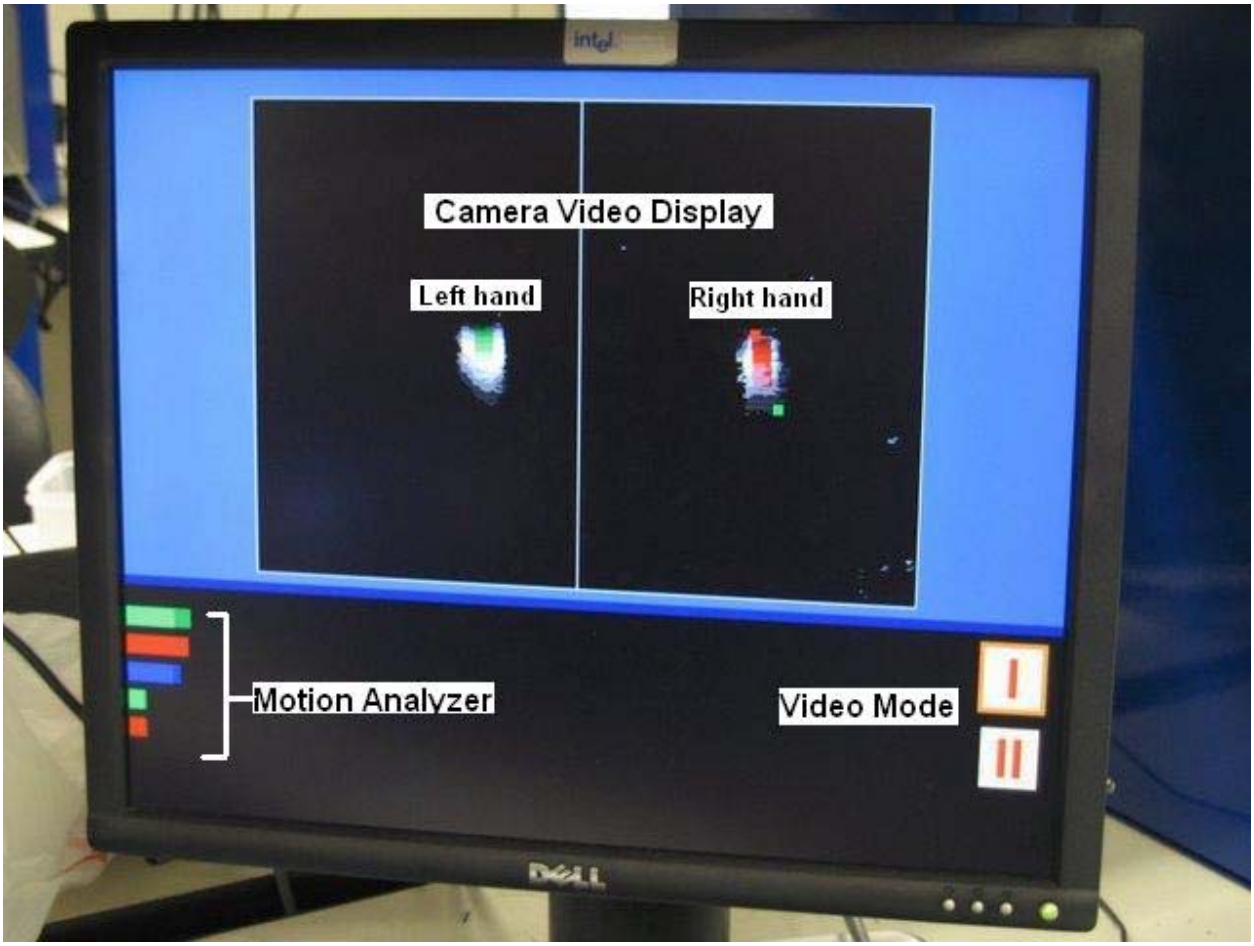


Figure 12: Description of Screen Components

### 2.6 Generate Visualization Module (generate_visualization.v) *(Brandon)*

The generate_visualization module takes in number of signals from the color_decision, video_processor, and motion_analyzer modules to output three signals of R, G, and B data for display on the monitor. Figure 12 is a sample display generated by this module.

The *cam_image[23:0]* signal from the color_decision module is used to display the raw camera input video in RGB format. The *desired_color* input from the video_processor module is used to display a pixel as white if high and black if low. Switching between these two display modes for the camera image area is handled by an FSM which keeps track of the current display mode. Mode 0 is the default that corresponds to the black and white *desired_color* display. Mode 1 is entered after pressing the down button on the lab kit and corresponds to color display of the camera input.

Additionally, the *left_x[10:0]*, *left_y[9:0]*, *right_x[10:0]*, and *right_y[9:0]* signals are used as inputs to centered_block modules to display squares at each hand's current coordinates. The centered_block module is a sprite which generates a color block on screen with its center at the input coordinates. For example, the block following the left hand will have input coordinates at *left_x[10:0]* and *left_y[9:0]*, with width and height of

20 pixels. The right hand has similar structure but with *right_x[10:0]* and *right_y[9:0]* as inputs. For color, the left hand block is green, while the right hand block is red to differentiate between the two. To add to the visualization, two sprites of darker hue for each hand follow the path of the hand blocks at positions delayed by around 0.2 second intervals. This is accomplished using the *count[18:0]* registers to count clock cycles at 27 Mhz such that each time *count* == 0 the current hand coordinates are stored in registers. These blocks are displayed on screen to add a slight trailing effect to the hand motion.

Another set of sprites is used to display the coordinates at which a beat begins and ends. Since beats are dictated by just the right hand, only two sprites are needed: one to remain at the start coordinates, and the other to remain at the end coordinates. These blocks are smaller in size, with a height and width of 10 pixels. The block at the start of a beat is green, while the block at the end of a beat is red.

Since all the above display signals are within the same area on screen, logic is used to determine which pixel data should be displayed over others. This layering is important to prevent odd colors from arising when adding together two different pixel streams. The order of precedence as described by the logic is as follows, from the uppermost to lower layer: hand position blocks, hand trailing blocks, beat start and beat end blocks, then finally the *cam_image[23:0]* or *desired_color* display depending on the current mode.

Other features of the display include the 1-pixel wide white border around the camera video, as well as the white line dividing the two halves of the screen. Around the thin white border is a blue border to fill the space of the upper half of the screen. These colored areas are generated using logic to divide up the screen into parts using the *hcount[10:0]* and *vcount[9:0]* inputs.

Below the camera display are motion analyzer bars which indicate relative amplitude, beat period, and acceleration values from the motion_analyzer over time. There are two of each the amplitude and acceleration values (one for each hand) and one beat period value, for a total of five bars on the screen.

The magnitude of each motion quality scales with the width of its respective bar on screen. These sprites were generated using the analyzer_bar module which takes the bar width and upper left hand corner coordinates as parameters. The positioning of the five bars are, in order from top to bottom of the screen: left hand amplitude, right hand amplitude, beat period, left hand acceleration, and right hand acceleration. All qualities for the right hand are colored red to match the block following the right hand, while qualities for the left hand are colored green to match the block following the left hand. This display allows the user to clearly see how his motions are affecting the playback of the music.

The button display on the bottom right of the screen is used to keep track of what display to show in the camera video frame. The upper button with the “I” label is the default mode for display of detected blue areas. The lower button with the “II” label is the mode for viewing a dimmed version of the camera input video.

### 2.7 Signal Tamer *(Andy)*

The Signal Tamer module is placed between the Video and Audio Processing parts of the system. The basic function of this module is to take the raw inputs of

velocity and acceleration, and to scale them and/or bit-shift them to make them compatible and within the range needed for the Audio Processing Module. Moreover, the module also “tames” the signals, meaning it low-pass filters the acceleration signals and allows for gradual volume transitions.

The low-pass filter for the acceleration signals is merely an average of the acceleration values of the last two beats. To achieve a gradual volume transition between two beats, interpolation is required. An audio beat is divided into 32 sections, each of which has a different volume value. Division 0 will have the volume of the previous beat and Division’s 15-31 will have the volume of the new beat. There will be a linear transition between divisions 0 and 15. Figure 13 illustrates this point. See Verilog code in the Appendix for details.

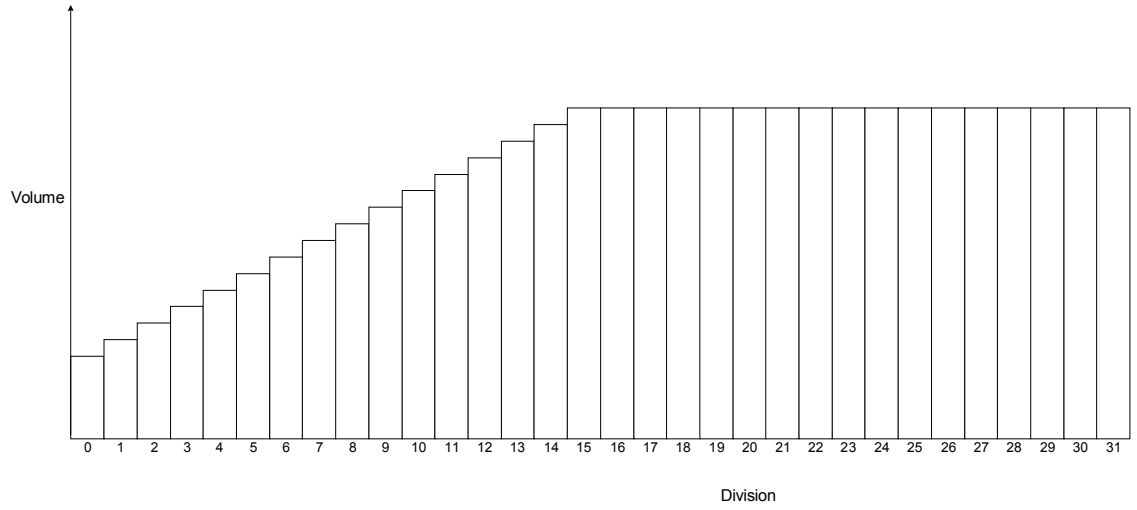


Figure 13: The volume fading feature of the signal tamer steps the volume between the previous and new volumes by using linear interpolation of the first 16 divisions of the new beat.

## 2.8 Audio Processing *(Andy)*

The Audio Processing Unit of the Virtual Conducting Project interprets the signals, *acceleration_left_fixed[1:0], acceleration_right_fixed[1:0], VelL_fixed[6:0]*, and *VelR_fixed[6:0]* to produce audio, stored in the flash ROM, to correspond to these characteristics. The Audio Processing Component takes audio stored from the flash ROM, and outputs it one beat at a time. The audio is also modulated in tempo, volume, and articulation.

The data flow begins at the ROM FSM. Interacting with the ZBT FSM and Beat Generator, the ROM FSM reads audio data from the flash ROM at 4 clock cycles per address ( 6.75 MHz). The ROM FSM will read data and pass it onto the ZBT FSM one beat at a time with help from Beat Generator. The data read in from the ROM FSM will be written into the ZBT SRAM at 4 clock cycles per address.

This is performed by reading in data from flash ROM and storing it into the ZBT RAM until a *musicbeat* signal is declared, demarking the end of a beat. After the *musicbeat* signal is declared, the ZBT FSM will allow the same beat of music stored in

the RAM to be read out at 24 KHz to the AC ’97 interface. Since the writing in to ZBT RAM is done at 6.75 MHz, the pause in between beats is barely noticeable by human ear.

The ZBT RAM allows for the writing and reading of different addresses in the ZBT RAM. During the writing cycle, it gives access only to the ROM FSM to write data into the RAM. After the write cycle, the read cycle begins, and the ZBT RAM with help from the Tempo Modulator reads data from the ZBT RAM.

The Tempo Modulator takes in the original beat period (from the music), and the user specified beat period and helps change the audio output speed. The main output from the Tempo Modulator will be addrmod[15:0] which will interact with the ZBT FSM to change the address accessed from the ZBT SRAM at the right times to change the tempo. If the user specified tempo is greater than the original tempo, addresses are added, if the specified tempo is slower, then addresses are subtracted.

The output from the ZBT FSM will be 24 KHz 8-bit PCM data. This audio data is fed to low pass and high pass filters. The outputs from the filters are the inputs for the Volume and Articulation Modulator.

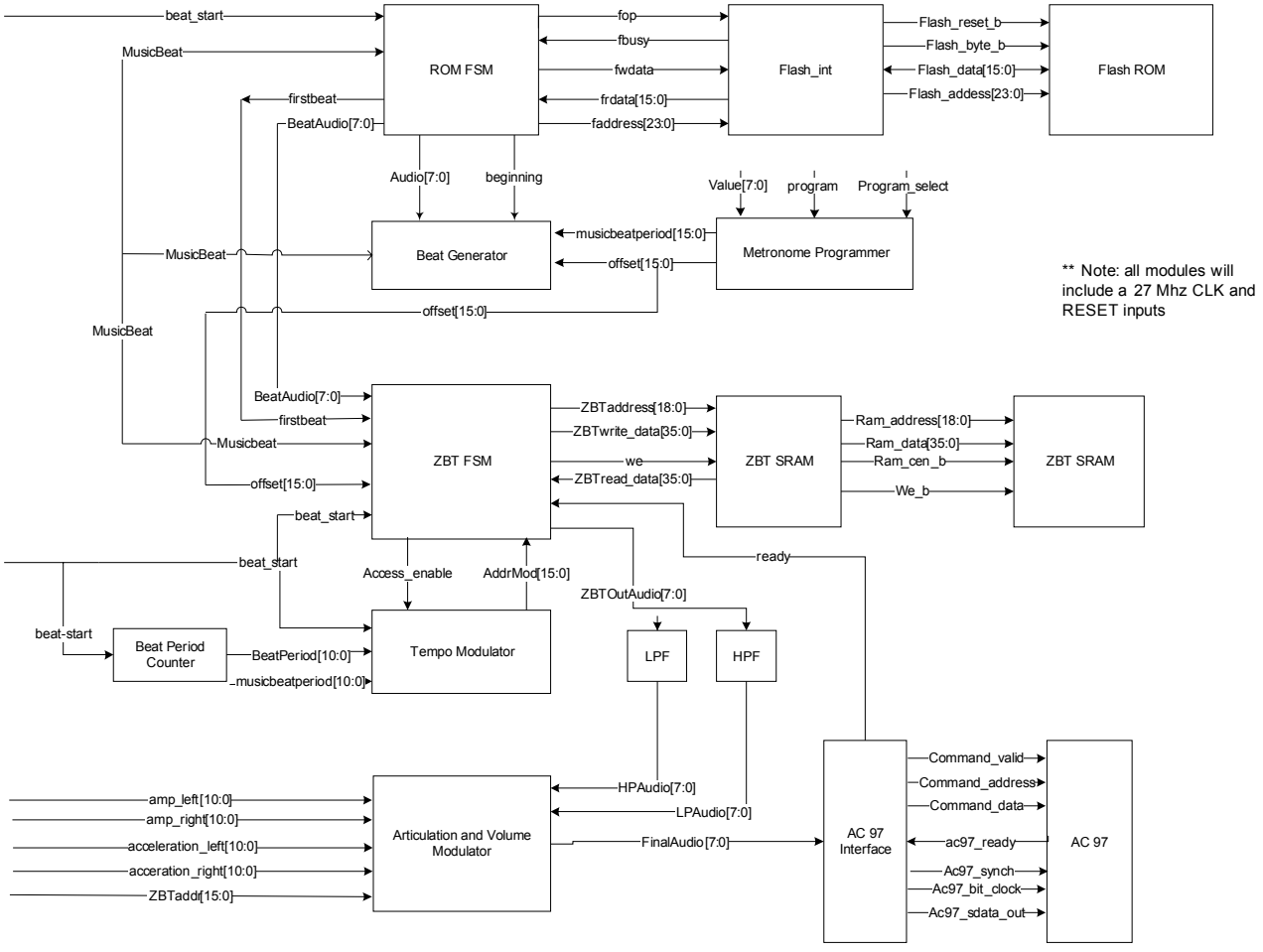


Figure 14: The block diagram for the entire audio system. User inputs are *Value[7:0]*, *program*, and *program_select* in order to reprogram *musicbeatperiod[15:0]* and *offset[15:0]*

The Volume and Articulation Modulator simply multiplies the existing audio by a preset function. If the articulation is weak (the acceleration is weak), the function to be multiplied is simply a constant throughout the beat. However, if the articulation is strong, the function is low initially, quickly climbs to a maximum, and slowly decays.
The output from the Volume and Articulation Modulator is finally fed into the AC '97 interface for audio output. See Figure 14 for the overall Audio Component block diagram.

## 2.9 ROM FSM *(Andy)*

The ROM FSM reads from the flash ROM and interacts with the ZBT FSM, and beat generator to allow output of one musical beat at one time. Initially starting at zero, an address counter increments in order to read from ROM sequentially. In this implementation, it takes four clock cycles to read from ROM. The first clock cycle declares a write operation. It was chosen to wait three more clock cycles to insure that the FPGA has correctly read one word of data from the flash RAM.

The ROM FSM has 4 states. The first state declares a read operation; the next two states are dedicated to allow for a delay before actually reading data. The last state reads the data from the flash, increments the address to read from, and returns the state to *state 0*. If the address to read from exceeds the number of addresses stored in ROM, the address will loop to 0. This cycle will not occur unless if *readcontinue* is true. *Readcontinue* turns high when *beat* is declared, and remains high until *musicbeat* is declared. Since *beat* represents when the user wants another beat to be played, and *musicbeat* declares the end of this beat, this allows audio data to be read out one beat at a time. See Figure 15 for details.

Moreover, the ROM FSM is responsible for letting the rest of the system know when the first beat occurs. The first beat contains a different beat period, and a different note extension technique, and is a signal other modules must use. Refer to the Appendix for details.

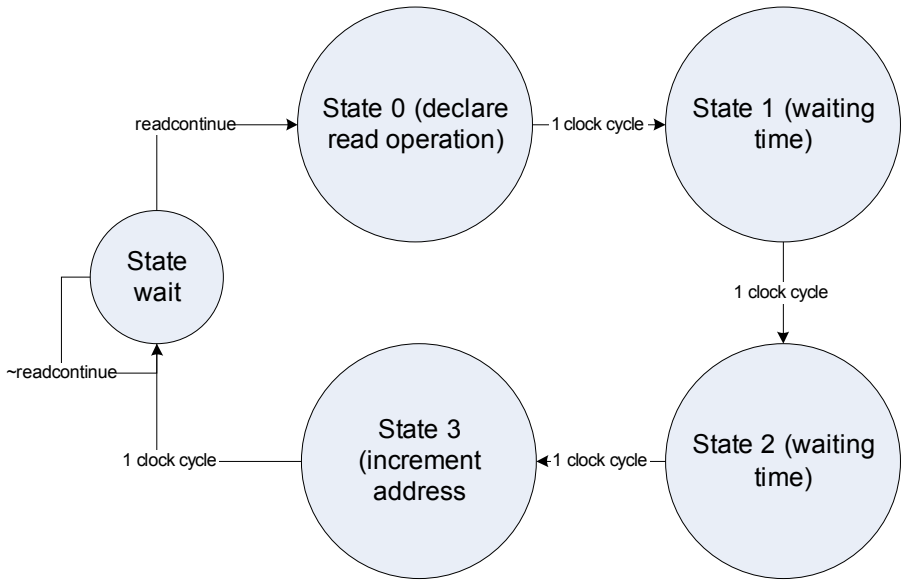

Figure 15: The ROM FSM has 4 real states, and one "wait state." The FSM is used to read data from the ROM at 6.75 MHz (4 clock cycles per address). *Readcontinue* turns high when *beat* is declared, and remains high until *musicbeat* is declared.

### 2.10 Beat Generator *(Andy)*

The Beat Generator module generates *musicbeat* signals that indicate the division between beats. This module uses a set beat period – given by the user- in order to generate this signal. The *sample_count[15:0]* register counts the number of times that the ROM FSM accesses memory. The number of times a different address in flash memory is accessed will be the unit of measure for the Beat Generator. When *sample_count[15:0]* reaches the beat period of the audio data, *musicbeat* will go high and *sample_count[15:0]* will reset to zero.

The Beat Generator interacts with the Metronome Programmer, which stores the value of *musicbeatperiod[15:0]* and *offset[15:0]. Musicbeatperiod[15:0]* specifies the regular beat period in flash ROM address accesses, and *offset[15:0]* specifies the offset to initially assign *sample_count[15:0].* The use of Metronome Programmer allows *musicbeatperiod[15:0]* and *offset[15:0]* to be reprogrammed by the user if desired.

### 2.11 Metronome Programmer *(Andy)*

Metronome Programmer is a small reprogrammable ROM that stores the constant values for *musicbeatperiod[15:0]* and *offset[15:0]. Musicbeatperiod[15:0]* represents the time period between musical beats in the audio that is loaded in the ROM. This time period will vary from song to song. By keeping *program* high, selecting which value to program with *program_select*, and setting the FPGA buttons to the intended value, *musicbeatperiod[15:0]* and *offset[15:0]* can be reprogrammed. Note that since the FPGA only has 8 buttons, the input will be an 8-bit number, but the needed value is a 16-bit number. The input is multiplied by 100 to solve this problem.

### 2.12 ZBT FSM *(Andy)*

The ZBT FSM interacts with the ROM FSM, the ZBT SRAM, and the Tempo Modulator. Inputs for the ZBT FSM are *musicbeat*, *first*, *beataudio[15:0], offset[15:0]*, *beat_start, addr_mod[15:0], ZBTreaddata[35:0],* and *ready*. Outputs for this module are *access_enable*, *ZBTaddress[18:0]*, *ZBTwrite_data[35:0]*, we, and *ZBTOutAudio[7:0]*. The ZBT FSM interacts closely with the ROM FSM; when the ROM FSM is reading out data, the ZBT FSM fetches the data from *beataudio[15:0]* and writes it into the ZBT SRAM simultaneously. Like the ROM FSM, the ZBT FSM writes a sample once every 4 clock cycles. When the ROM FSM is not reading out data, the ZBT FSM reads out the data that was just stored at 24 KHz. See Figure 16 for the state transition diagram.

The transition between writing and reading from RAM is dictated by the *continue* signal. When *continue* is high, the ZBT FSM will write data into the RAM. When *continue* is low, the ZBT FSM will read data and play it at 24 KHz. *Continue* is high when a user beat is specified. *Continue goes* low when *musicbeat* goes high, specifying the end of the beat stored in the ROM.

When the ZBT FSM has reached the end of the beat, and a new beat has not been specified yet, it will replay back sections of the end of the beat to allow for note

extension. Note extension allows the music playback to appear fluent and without any gaps in playback. After exceeding the *musicbeatperiod[15:0]* length, the FSM will reverse playback until it reaches address *musicbeatperiod[15:0*] – 2000. Upon reaching this point, it will reverse playback again, until reaching *musicbeatperiod[15:0]* – 500, in which case it reverse direction again. This loop continues indefinitely until the next beat is specified by the user. See Figure 17 for a diagram of this loop. If *beat* is specified before the end of the beat in the flash ROM, the existing beat is truncated, and the next beat is read from the flash ROM, written into the ZBT SRAM, and then read out from the ZBT SRAM.

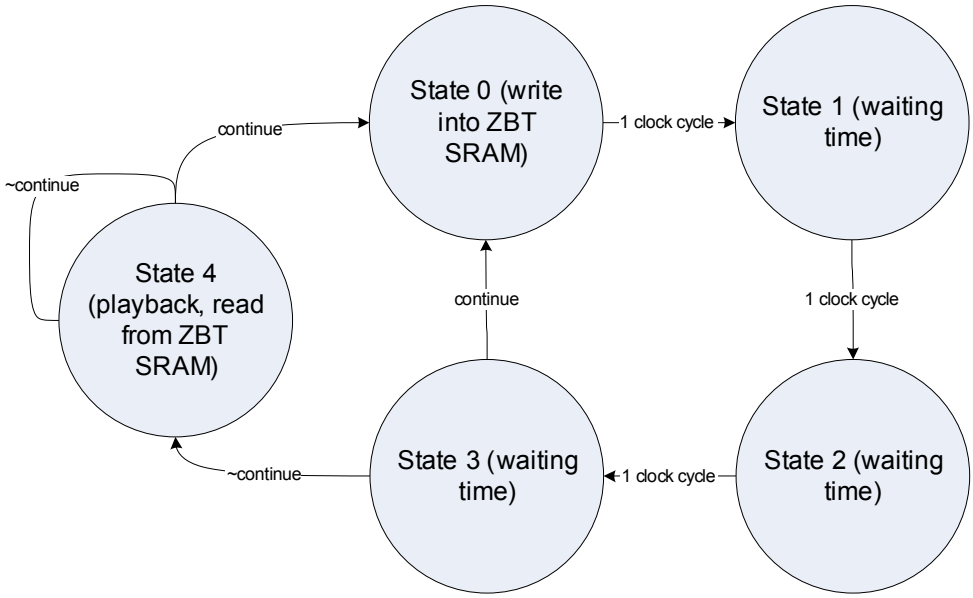


Figure 16: The ZBT FSM. The ZBT FSM alternates between writing into the ZBT SRAM, and reading out from it. This alternation is controlled by the *continue* signal.


Figure 17: Note extend works by reversing the playback once the address reaches *musicbeatperiod*[15:0]. Playback will reverse again after reaching *musicbeatperiod*[15:0] – 3000. After reaching *musicbeatperiod*[15:0] – 500, playback will reverse again. Thus, playback is contained within *musicbeatperiod*[15:0] – 3000 and *musicbeatperiod*[15:0] – 500.

## 2.13 Beat Period Counter *(Andy)*

The Beat Period Counter is a very simple module which merely counts the time period between user specified beats. The module consists of a sample access counter which increments every 100 accesses of a sample. The counter resets every time *beat* is high, and immediately outputs the resulting beat period. This module is particularly important for the Tempo Modulator Module.

## 2.14 Tempo Modulator *(Andy)*

The theory behind the Tempo Modulator is simple. Audio is divided into segments of 800 segments; these segments are then added or subtracted to change the tempo of the music output. See Figure 18 for an illustration of this concept. The Tempo Modulator is divided into two sub-modules: the Division Converter and the Division Counter.

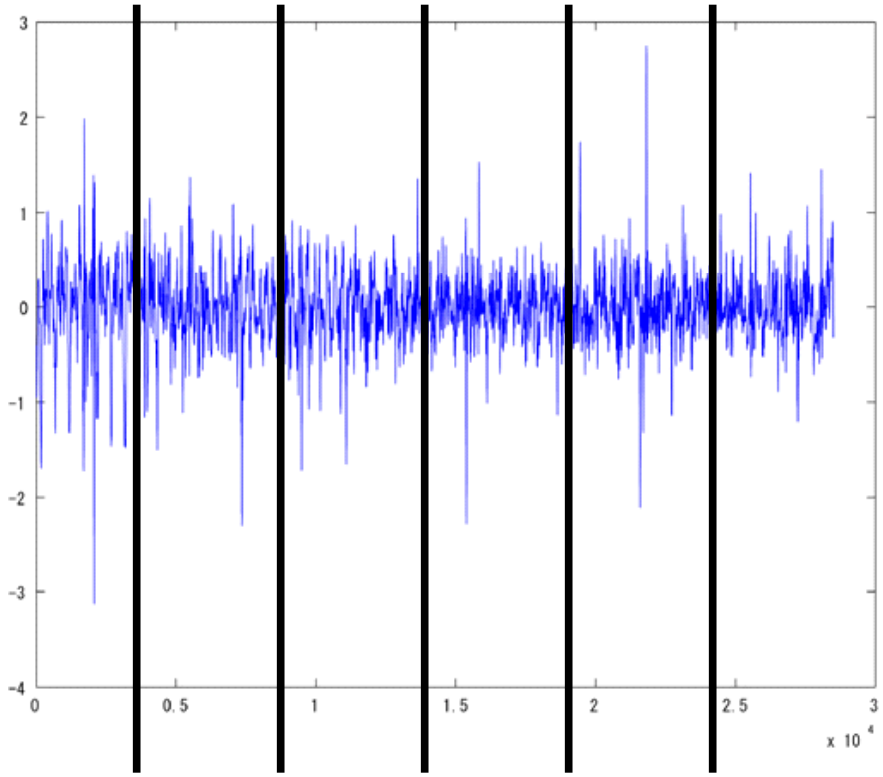


Figure 18: This illustrates the basic concept behind the tempo modulator. Audio data is grouped into segments of 800 samples and are added or subtracted in order create a slower or faster playback.

The Division Converter converts a ratio between the original and the intended beat periods into signals that are easier to use – *interval[2:0], skip [2:0],* and *add*. These signals instruct the Division Counter to add or subtract *skip[2:0]* amount of divisions every *interval[2:0]* of divisions. These signals are fed into the Division Counter to produce an *addrmod[15:0]* signal when *interval[2:0]* amounts of divisions has been

read. *Addrmod[15:0]* instructs the ZBT FSM to change the address by *addrmod[15:0]* in order to add or subtract samples, and thus change the speed of playback.

The concept behind Division Converter is simple, but the implementation is a harder than suspected. Division Converter is an FSM. In state 0, the initial state, initial values are assigned to registers *NewBeatPeriod[10:0]* and *NewOriginalBeatPeriod[10:0]*. In state 1, these values are shifted right until the values of both *NewBeatPeriod[10:0]* and *NewOriginalBeatPeriod[10:0]* are less than 7 (which means both values are 3 bits or less). When the values are 7 or less, then the state machine goes to state 2. State 2 determines interval and skip on a case by case basis. Refer to Figure 19 for a state transition diagram.

There are 3 basic cases, when *BeatPeriod[10:0]* is greater than *OriignalBeatPeriod[10:0],* when they are equal, and when *BeatPeriod[10:0]* is less than *OriginalBeatPeriod[10:0].* In the first case, interval is set to equal *NewOriginalBeatPeriod[2:0], interval[2:0]* is set to 1, and add is set to equal to 0; in the 2nd case, all three signals are set to 0 – this reproduces the original tempo; in the 3rd case, interval is set to *NewBeatPeriod[2:0]*, *skip[2:0]* is set to *NewOriginalBeatPeriod[2:0]*. The result of this is that the playback will be at a proportional speed to the ratio between *BeatPeriod[10:0]* and *OriginalBeatPeriod[10:0]*. Refer to Appendix and Figure 20 for details on special cases of these three cases. Note that this calculation is merely an estimate of how fast the playback should be. If there is a large difference between intended playback speed and actual playback speed, the difference is not a problem because of note extension and beat truncation, which keeps the playback smooth.

The second part of this module, Division Counter takes the outputs from Division Converter and converts the inputs into the *addrmod[15:0]* signal. Division Converter interacts with the ZBT FSM to count the number of divisions accessed. When *interval[2:0]* divisions has been accessed, *addrmod[15:0]* will indicate the number of addressed to add. Note that *addrmod[15:0]* is signed, so when addressed must be subtracted to slow down the playback, *addrmod[15:0]* will be negative. Refer to Figure 21 for the block diagram for Tempo Modulator.

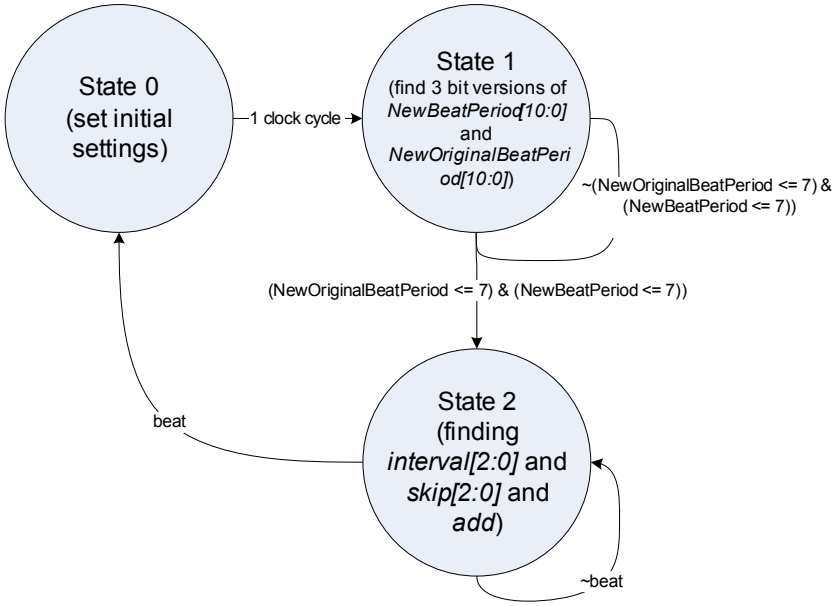


Figure 19: The state transition diagram for Division Converter. Division Converter converts *BeatPeriod[10:0]* and *MusicBeatPeriod[15:0]/100* into *interval[2:0]*, *skip[2:0]* and *add*.


Figure 20: These are the different cases for the Division Converter for *BeatPeriod[10:0]* and *OriginalBeatPeriod[10:0]*. Note that the Division Converter will make the maximum speed 8X and the minimum speed ½ X.

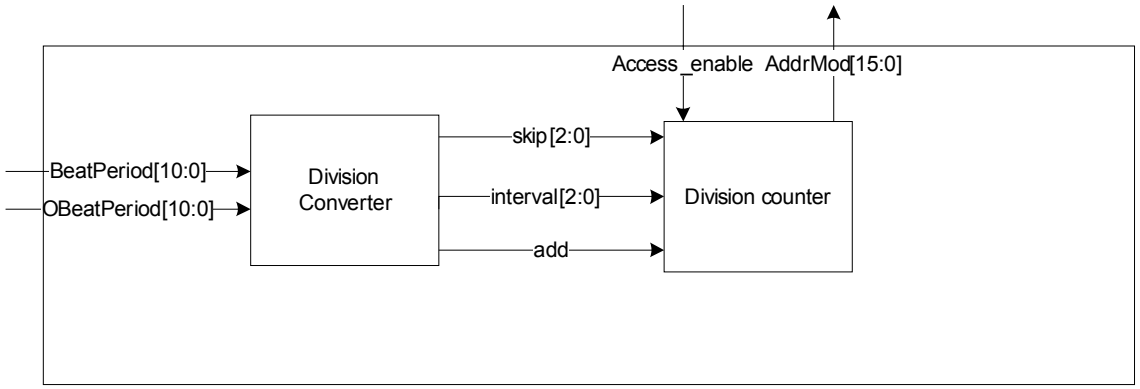


Figure 21: Block diagram for the Tempo Modulator. *Access_enable* and *AddrMod[15:0]* interact with ZBT FSM above (refer to Figure 14 for details).

### 2.15 Volume and Articulation Modulator *(Andy)*

The Volume and Articulation Modulator simply multiplies one beat of audio by a function to achieve the results. In theory, multiplying each beat by an envelope function (coefficient function) will replicate articulation effects such as staccato. This module is split into two units, one for the treble, and one for the bass. Each Volume and Articulation Unit divides each existing beat of audio into 16 divisions. Each one of the 16 divisions is multiplied by a coefficient (a function of which is stored in a small ROM). There are 4 settings of articulation; when *Acc[1:0]* is 0, this corresponds to the smoothest playback and coefficients which are constant at 255; when *Acc[1:0]* is 3, this corresponds to the choppiest playback and coefficients which change the most. See Figure 22 for an illustration of this concept.

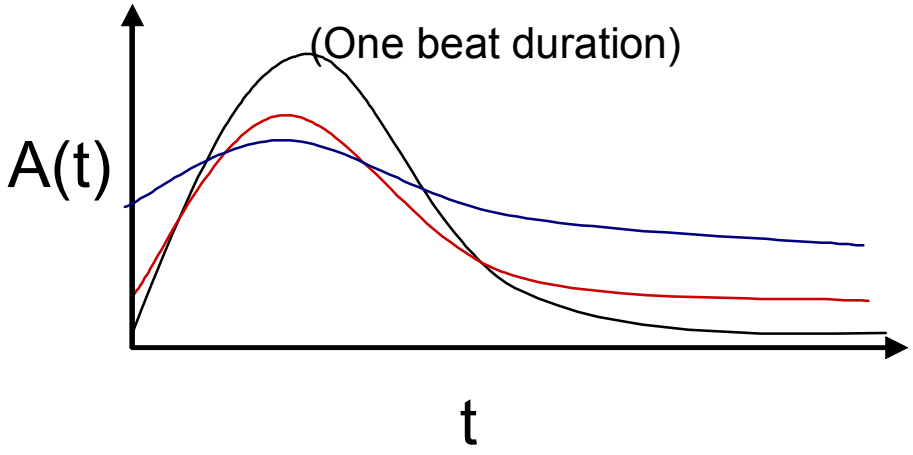


Figure 22: This figure illustrates the concept of articulation modulation. A smooth playback corresponds to an envelope such as the blue one, a medium articulation playback corresponds to an envelope like the red one, and a staccato playback corresponds to something like the black envelope.

For volume modulation, each sample of audio is simply multiplied by an appropriate constant. Note that if the audio is scaled down too much to correspond to a

soft sound, the resolution will deteriorate (imagine shifting audio to the right until there is only 1 bit). This problem is solved by having the volume data scale not only the raw audio signal, but the AC'97 volume level as well. The Volume and Articulation Modulator uses the greater of the two (left and right) inputs to determine the AC'97 volume level. See Figure 23 for the block diagram for the Volume and Articulation Modulator. The results of the audio data from the two units (left and right) corresponding to bass and treble are then added together to produce the final result.

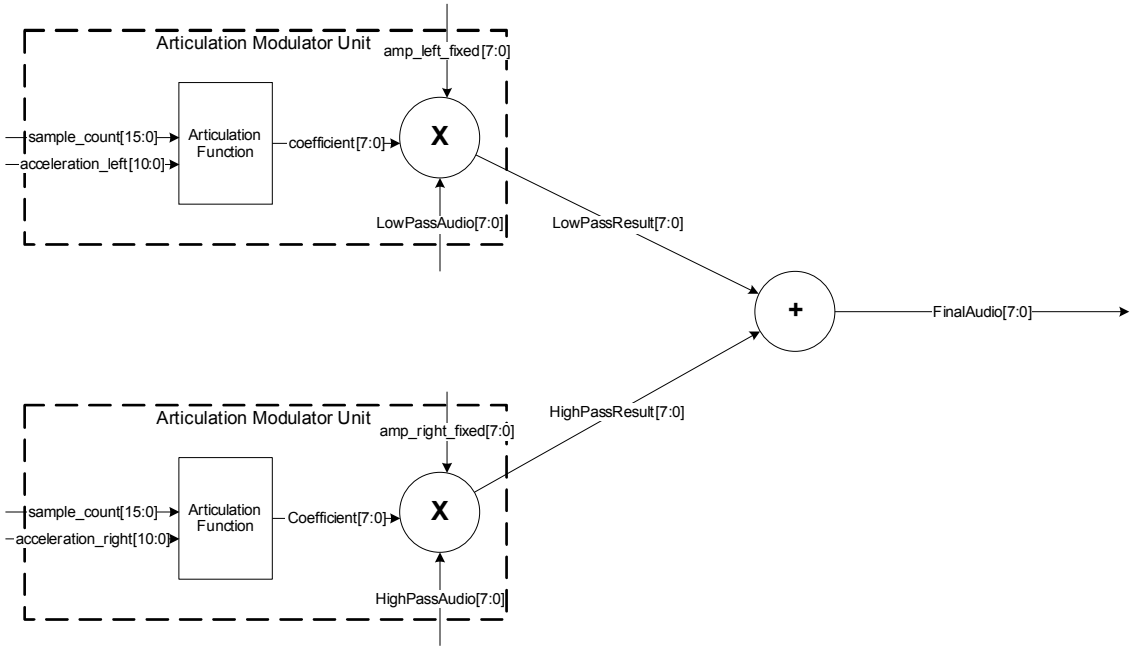


Figure 23: This block diagram represents the Volume and Articulation Modulator.

## 2.16 HP/LP Filters *(Andy)*

The high-pass and low-pass filters used to divide the output of the Tempo Modulator were generated from the Xilinx Coregen Finite Impulse Response Module. They take as input the original audio, and a processing enable, and outputs the resulting audio and a ready signal. Two 31-segment, 16-bit finite impulse response filters were used for the project. The coefficients for the filters are the impulse response for a simple Hamming Window that has a cut-off at π/32. A Hamming Window compared to a Rectangular Window trades off a sharp cut-off for a greater and non-oscillatory attenuation of stop-band frequencies. A discrete time frequency of π/32 corresponds to a continuous time cut-off of 750 Hz (π/32) / (2π) * 48000). See Appendix for the coefficients used for the two filters. See Figure 24 and Figure 25 for the Fourier Transform of the two filters.

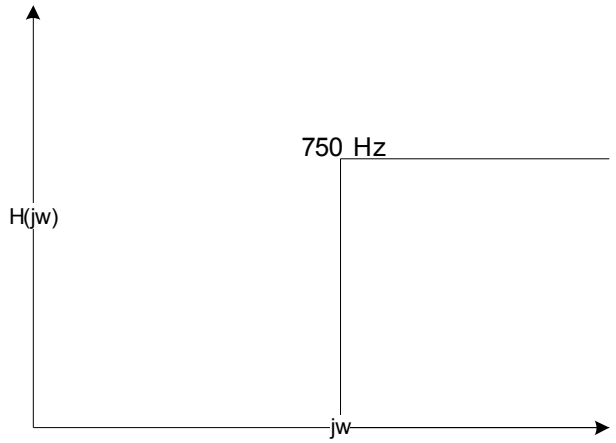

Figure 24: Fourier Transform of high-pass filter with a cut-off frequency of 750 Hz.

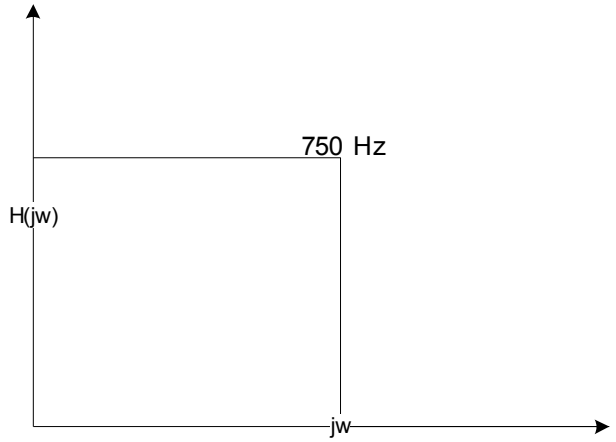

Figure 25: Fourier Transform of low-pass filter with a cut-off frequency of 750 Hz.

### 2.17 ROM Writer *(Andy)*

In order to write data onto the ROM, a ROM writer had to be built. The ROM Writer basically unlocked the flash ROM, erased the necessary blocks, and wrote the appropriate data onto the ROM at the correct addresses. The FSM needed to build this was derived from the Flash ROM test example provided on the 6.111 website. Instead of writing dummy data onto the ROM and testing if it was correct, then erasing it again, the system was modified to write data from a BRAM, and to not erase the data. The data from the BRAMS from .coe files. These files were obtained using Matlab to extract and scale the audio .wav files, and the Xilinx Coregen Memory Tool was used to create the .coe files. A total of 24 BRAMS were created, each with a depth of 65536, and a width of 8.

# 3. Testing and Debugging

## 3.1 Video Component *(Brandon)*

For the modules involving video, testing and debugging was mostly done on the monitor. Since the display was such an important part of testing, retrieving the camera input video was the first important task. The sample code on the 6.111 website was used for interacting with the ZBT, with certain modifications for this project. The provided code stored only 8 bits of information per pixel, while this implementation required 16 bits. Determining how many bits to store per pixel required testing in itself, involving changing the number of and proportion between stored Y, Cr, and Cb data to view each modification's affect on display quality. The sample code initially stored only 8 bits of Y information per pixel, while this project required both Cr and Cb bits for color detection and display. As the modules of the sample code were already organized nicely, much of the debugging process involved changing parts of the ntsc_to_zbt and vram_display modules to reflect the new storage scheme of 16 bits per pixel.

Modification of these two modules required thoroughly understanding the sample code. For a couple of days, changing parts of code and rebuilding repeatedly led to no good results. The video always had some problem with it, no matter what combination of changes were tried. It was taking too long to figure out, and there were many other modules to build, so it was important to move on. Yet, since video was such an important part of debugging the other modules, using the sample code was necessary. Extending the storage to 16 bits per pixel was a problem, but changing the composition of the initial 8 bits in the sample code wasn't a problem. Thus, a storage scheme of 5 bits of Y and 3 bits of Cb information per pixel was temporarily chosen for use while testing the other video modules.

The next module to build was the color_decision module. This module was tested by connecting it between the camera retrieval blocks and the VGA display. This method of debugging proved to be much simpler than generating simulations in Modelsim. The output of *pixel_video* was used to display white pixels when the desired color was detected, and black otherwise on the monitor. This display allowed for testing the use of different thresholds on the magnitude of Cb for adjusting sensitivity to the blue content in a camera image. Using switches, the minimum Cb value was tuned to fit the output with the best detection of bright blue lights. Having the detected areas displayed on screen provided a lot of useful feedback for adjusting such color threshold parameters.

Once this was working, the next module was the video_processor module, consisting of its two submodules of color_detection and position_calcuator. Testing was first done on the color_detection module to ensure the correct *left_en* and *right_en* outputs would be sent to the position_calculator. In this case, the first round of tests were done in Modelsim. After adjusting inputs, simulations were run until the correct *left_en* and *right_en* outputs were displayed. As this module follows from the inputs of the color_decision module, it was first added to the project with direct connections, rather than through the enclosing video_processor module. Error correction of comparing consecutive *pixel_video* bits was tested at this stage. Again, the results were evident visually on the screen. Connecting the module to a switch allowed for comparison between the cases with and without error correction. Through this process,

it was found that comparing two consecutive pixels was enough to reduce noise considerably, while the area of the user's lights still remained large enough to be useful in position calculations.

Next, the position_calculator was tested to output the result of an average calculation. First, this was tried in Modelsim, but the method of working directly with the Labkit proved to be the most effective debugging tool in the end. For one, use of the Xilinx Pipelined Divider Core was necessary in the actual implementation, so it was important to test with this particular module included. To test if the divider worked correctly, an instance of it was first instantiated at the top-level conducting file in Xilinx and connected to constant inputs. The result was displayed on the 8 LEDs of the Labkit. Since the divided results for chosen constant inputs were known beforehand, matching expected results with the LEDs provided evidence to the correct operation of the divider.

For testing the correct result of the position_calculator modules as a whole, sprites at output coordinates for the left and right hands were instantiated in the top-level file of Xilinx to track the hand motions. Initially, the outputs produced unexpected results. The results would only sometimes match the expected location of the weighted average positions. It turned out to be the timing of the divide calculations which led to this problem. The divide calculation takes a total of 28 clock cycles to complete. Code was written to time a synchronous clear input to the divider to make sure new data would always be received at output. However, this timing allotment was done assuming a 65 Mhz clock was used as soon as vsync went low for output results to be ready the next time vsync asserted high. This was a mistake in that the operation of the divider was actually on the 27 Mhz clock, so the divide results were not calculated in enough time for capturing the correct values.

Correcting this problem still led to puzzling results, however. The problem this time was that the sclr input to divide was disabled by default in Xilinx. As a result, none of the clear signals were even hooked up to the module. As the module was never synced with the video display, this led to odd results in the cases when divide results were misaligned in time. Fixing these problems led to correct display of each hand's position.

Once both of these submodules of the video_processor module were completed and tested on the top-level, they were moved into the video_processor module for better organization.

The motion_analyzer module also consists of two submodules which were each tested sequentially on the top-level file by viewing the monitor. To test the beat_markers module, a square block is placed on screen at the coordinates when a beat start was registered, as well as at the coordinates when a beat end was registered. The placement of blocks allowed for easy adjustment of the sensitivity of the system to beat detection. Using switches to adjust parameters described in the beat_markers section, testing was done to find the best combination of values. No Modelsim simulation could provide a sense of the user’s actual motion, so it was important to test this module using the display.

The qualities_generator module was also tested using the display. Firstly, the find_distance module was tested with constant inputs for correct operation. Displaying the results on the LEDs allowed for debugging of the module. Next, the

qualities_genetator module was tested by displaying the amplitude, beat frequency, and acceleration outputs as the width of bars displayed on the bottom half of the monitor. Since amplitude calculations were based on the pixel distance between a beat start and end, it was easiest to see the module working by these bars. The monitor provided instant feedback for guaranteeing the calculation was correct. The original idea was to use the logic analyzer for testing, but this proved unnecessary and even harder to do since the tester cannot remember every set of motions he made to compare for correct results on the analyzer. Rather, visually confirming correct calculations while conducting provides the best feedback.

Once all these calculations were complete, the sprites for testing were all grouped into the generate_visualization module for better organization. When moving large amounts of signals to other modules, problems with forgotten or mismatched signals arose. These issues were eventually resolved, but took time to correct and debug.

Finally, after completing most of the video portion, the camera input and storage portion was fixed. After careful examination of the code structure, it became much clearer as to how to modify the code to support 16 bits per pixel of YCrCb data storage. As this greater amount of color information became available, it was possible to generate color video in RGB format for display on the screen. Additionally, new color threshold tests were done using the new Cr and Cb data to find even better thresholds for detecting the blue LEDs.

In general, it was particularly time-consuming to debug modifications to the camera input and display modules. Much time could've been saved in taking the extra time to clearly undertand the operation of the existing code. Making changes with only half an understanding of the code led to countless compiles which could have been avoided.

Tuning the system to be responsive enough to the user's motions yet not too sensitive was also a long process. The orginal design registered a beat end as soon as the user's motion slowed. This led to problems, however, with conducting at a slow tempo. If the user's hand moved slowly, multiple beats would be registered in a row, leading to a string of unwanted beats in a row.

To remedy this problem, the implemented FSM was used to separate a potential stop from an actual stop. With this system, the user could move very slowly yet still produce beat end markers within reasonable time. The many adjustable parameters in this model were a bit overwhelming at first, but provided the freedom needed to tune the system for our particular purpose.

### 3.2 Audio Component *(Andy)*

Testing and Debugging of the Audio section always maintained an audio input and output so that an audio signal could be heard at all times. Before embarking on tackling the design head-on, it was decided that small prototypes should be developed first. It did not take long to develop a prototype for the high-pass and low-pass filters, as well as a simple tempo-modulator which was only able to play back the audio twice as fast or twice as slow. However, the audio data used was directly read from a BRAM, for simplicity sake.

In order to build the actual system, some sort of audio needed to be loaded onto ROM. This task had proved to be harder than expected, since there was no direct example of audio being read from the flash ROM available on the website. However, there were examples of data being written into the flash ROM – and this example was modified to write data from the BRAM onto the flash ROM. In order to read from the flash ROM, Lab 4 was modified to include part of the flash ROM testing code available on the website.

After finally obtaining some audio output from the ROM, the ROM FSM and ZBT FSM were implemented. This had proved to be more difficult than expected, once again. The majority of time was spent trying to obtain some sort of output. Since the incorrectness of a system is hard to judge without any audible output, the use of the logic analyzer proved quite helpful. The logic analyzer helped display the data that was read from the flash ROM, as well as the data to be written into the ZBT SRAM. Using the logic analyzer to display the states of the two finite state machines also proved quite helpful. At first, the data was being read from the ROM incorrectly because of the incorrect declaration of the ROM reading commands and insufficient delay between reads. These two problems were quickly corrected.

The next step to the completing of the audio section of the project was to add the Beat Generator. For reasons to keep the system simple, the Beat Generator was originally incorporated in the ROM FSM. However, to make the implementation clean, the Beat Generator was eventually moved to its own module. The use of ModelSim was used to determine if Beat Generator operated correctly. See Figure 26 for a screenshot of ModelSim.

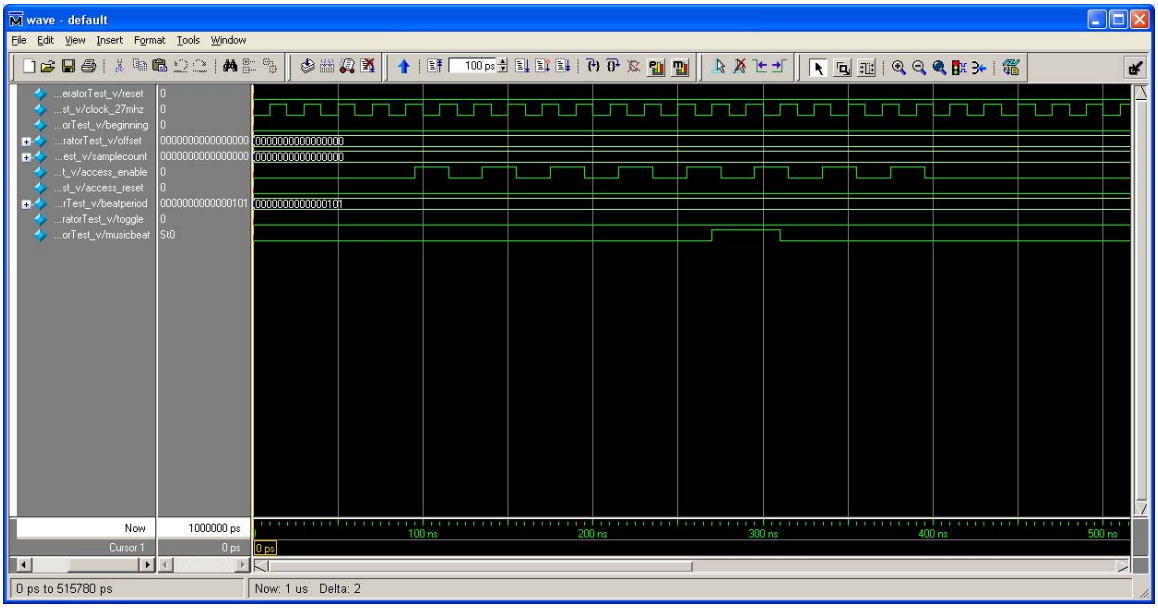


Figure 26: ModelSim simulation of Beat Generator Module. This is one of many ModelSim simulations performed.

Afterwards, the Tempo Modulator was implemented. Since the Tempo Modulator involved a tricky FSM, it was useful to debug the module by itself in ModelSim beforehand. ModelSim proved to be a real time-saver because instead of

compiling the entire project, the module was simply fed into ModelSim, shortening the debug cycle time drastically. The use of a logic analyzer was again useful, in determining if the system was working in actuality, not just in theory.

The completion of the Tempo Modulator was a major milestone in for the audio section of the project. The high-pass and low-pass filters were then added to the chain of audio processing. Checking the correctness of these filters was done by comparing logic analyzer results with Matlab results. See Figure 27 for a Matlab screenshot of the theoretical results of the low-pass filter and Figure 28 for the simulation for the high-pass filter. These Matlab results closely matched actual low-pass and high-pass data, as displayed on the logic analyzer.

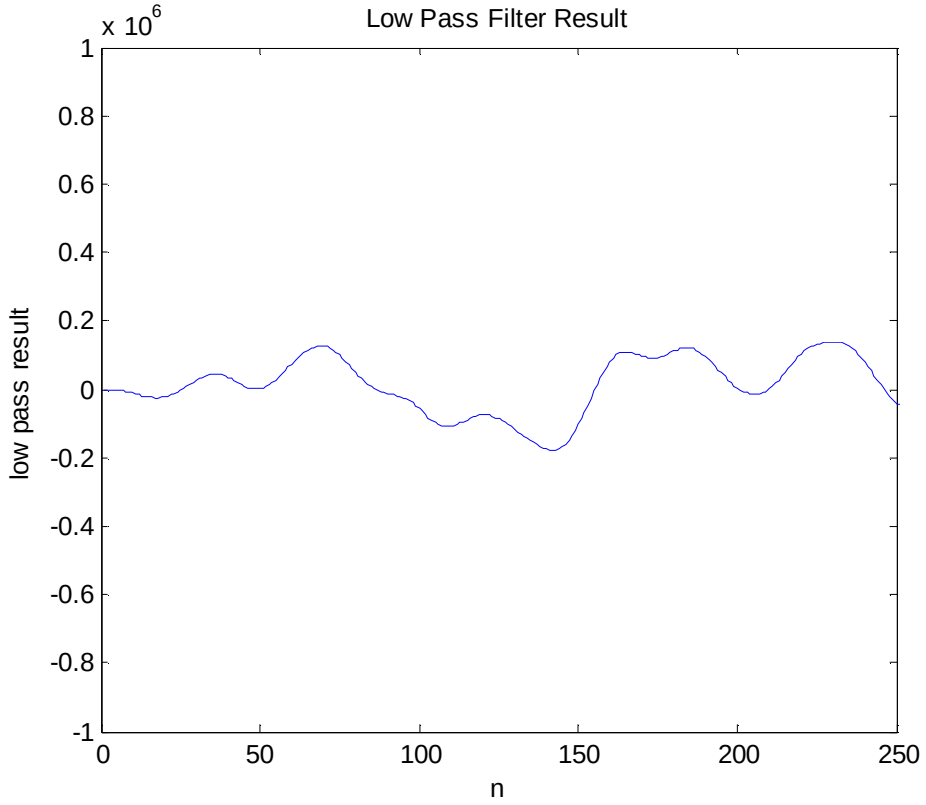


Figure 27: Matlab simulation of low-pass filter on a sample of audio.

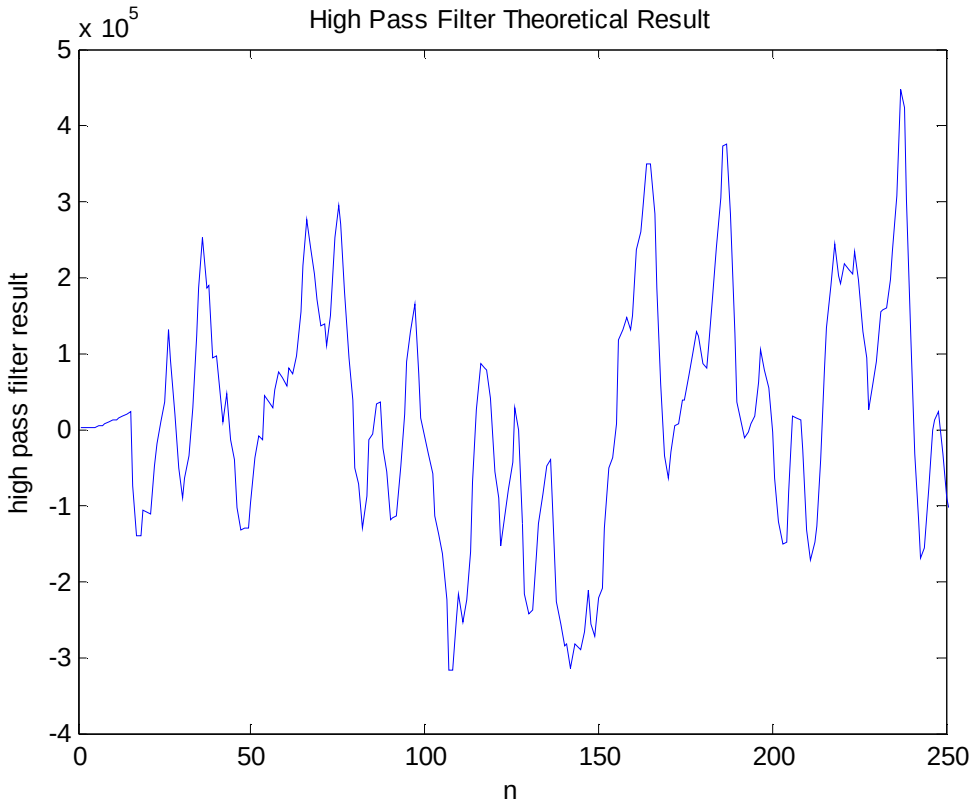


Figure 28: Matlab simulation of high-pass filter on a sample of audio.

The next module was the Volume and Articulation Modulator. This module was completed rather quickly mostly because it was simpler than the rest. The scaling of the audio as a whole required major tinkering to make sure that the audio occupied the correct bits of the output, and made use of the 8-bit resolution to the fullest. However, it was apparent that there was a rather grave problem: the articulation modulation performed as intended from theory, however, did not make the music sound like it had different articulation. Instead, it just made the beginning parts of beats sound louder than others.

This problem was derived from the fact that the melody changed notes multiple times during one beat; the articulation modulation implemented only accounted for 1 note-change per beat. Thus, every note-change needed to be identified to solve the problem. A large amount of research was then performed to try to correct this problem.

However, it was later discovered that real articulation modulation was quite difficult. Many algorithms using the Fast Fourier Transform existed to identify note changes – but those only worked for songs that had no accompaniment (only one instrument played at once). However, the song we decided to use, *Bolero,* consisted of many instruments playing at once. Therefore, a significant change in the FFT spectrum did not necessarily translate into a change in a note in the melody. Thus, the idea of correct articulation modulation was abandoned, and the existing articulation modulation was put into place for completeness.

### 3.3 Overall System

When the audio section was connected, it was discovered that the scaling of audio signal in the volume modulation was problematic: when the audio signal is reduced to too great of an extent, the resolution deteriorates and the sound quality becomes abysmal. To solve this problem, the Volume Modulator not only scaled the signal – it also scaled the volume of the AC ’97 interface. Therefore, to create a soft audio playback, the audio signal was scaled down – but not to the extent that playback quality deteriorated, and the AC ’97 volume was turned down as well. This resulted in a much larger gamut of dynamic values possible from the system.

Another problem which occurred: the playback sounded rather choppy because particularly when there was a large difference between volume levels of neighboring beats. At first, an averaging technique was put into place to set the current volume to be the volume of the last 3 beats. However, the response time of volume changes suffered, but in exchange for a smoother playback.

To also allow for faster response time and even smoother playback, another technique was used: interpolation. One beat was divided into 32 segments. The first 16 segments consisted of a linear change between the old volume and the new volume, while the last 16 segments maintained the new volume. The result was an amazingly improved playback, and a more natural-sounding system.

# 4. Conclusion

The goal of the Virtual Conducting System was to emulate the experience of a conductor directing an orchestra. A real conductor usually waves his/her hands to adjust the tempo, the dynamics, and the feelings of the orchestra. Moreover, the conductor can choose to adjust the balance of the orchestra (how loud each instrument plays).

Our Virtual Conducting System achieves these objects in many ways. A gesture given by the user, by blue LEDs in both hands is interpreted into musical qualities: the beat period and the amplitude and acceleration of the gesture are specified by the Video Component. The Audio Component takes these signals and reproduces audio that is played back a different tempo, loudness, and articulation depending on what the signals provided are. Even the smoothness of volume transitions is implemented using the Signal Tamer module.

However, there are many ways to improve our system. First of all, the usability of our system is susceptible to background noise and other light sources. Moreover, the presence of the color blue may cause our system to behave undesirably. The optimum conditions would have been operating the system in a dark room – this would be similar to a real conductor conducting an orchestra during a performance, but we'd prefer to conduct in normal conditions. If available, a different motion tracking medium could had been experimented with – for example, maybe the use of a laser-tracking system could potentially produce better results.

The audio playback also displayed some nondesirable behavior: for example, the audio seemed particularly noisy for a digital system. This was primarily due to the way the scaled audio was created: by removing and adding groups of samples, high frequency noise was added to the system. We tried to eliminate this problem by simply pointing the balance on the speakers towards the bass; however, a more robust solution existed: using a low-pass filter. However, in order for this technique to work, we would have unavoidably lost some high frequency audio since the high frequency noise was in the same frequency range as some parts of the audio. A way around this could had been to turn on the low-pass filter only when the division additions or subtractions occurred, thus only turning on the low-pass filter when needed, and minimizing the undesirable effect of the low-pass filter.

As discussed before, if time were available, a different approach to the articulation modulation could have been attempted, though good results do not seem so easy to obtain. Also, if more time were available, an easier way to load audio onto the flash ROM could had been implemented. Early on, Andy experimented with loading audio directly from an AC '97 input to a flash ROM, but did not obtain usable results. Such a technique could have been researched further. Moreover, other techniques such as uploading data through the RS232 interface, or using a USB or even Ethernet connection interface could have been attempted. Such an accomplishment would make the system more complete, flexible and universal.

Overall, the implementation of the Virtual Conduction System was tedious, yet enjoyable, and an excellent experience. From the project, it was valuable to understand how important it was to plan ahead, and have a "masterplan" before the system implementation was underway. It was also very important to understand that

prototyping and experimentation is greatly necessary before creating the “masterplan.” Working with a partner was a good way to experience real-world situations.

Moreover, the Virtual Conducting Project taught the ways to debug a very complex system. Usually, when a component did not work, it was useful to reduce the problem to something simpler, and to get the simpler system to work first. Implementation of the Virtual Conducting Project required important attention to details, without losing sight of the big picture.

In many ways, the Virtual Conducting System was a huge success: all planned features were implemented and worked in practice. User gestures using 2 blue LEDs due indeed result in different audio that varies in tempo, volume, balance, and articulation. The potentials of the idea of a Virtual Conducting System are endless. For example, more emotion can be added to the audio playback. Also, the system could potentially detect different instruments and allow the user to control specific instruments instead of only the bass and the treble. A supplementary display could potentially display a “virtual orchestra” playing the song the user is conducting. No matter how much could be added to the system, the existing features make a solid base of features that makes it exciting to dream and aspire to create an even more innovative virtual conducting system in the future.

# 5. Appendix

## 5.1 Low Pass Filter Coefficients

| Segment # | Coefficient Value |
|---|---|
| 0 | 111 |
| 1 | 132 |
| 2 | 184 |
| 3 | 270 |
| 4 | 388 |
| 5 | 538 |
| 6 | 713 |
| 7 | 907 |
| 8 | 1112 |
| 9 | 1318 |
| 10 | 1514 |
| 11 | 1692 |
| 12 | 1841 |
| 13 | 1954 |
| 14 | 2024 |
| 15 | 2048 |
| 16 | 2024 |
| 17 | 1954 |
| 18 | 1841 |
| 19 | 1692 |
| 20 | 1514 |
| 21 | 1318 |
| 22 | 1112 |
| 23 | 907 |
| 24 | 713 |
| 25 | 538 |
| 26 | 388 |
| 27 | 270 |
| 28 | 184 |
| 29 | 132 |
| 30 | 111 |

## 5.2 High Pass Filter Coefficients

| Segment # | Coefficient Value |
|---|---|
| 0 | -111 |
| 1 | -132 |
| 2 | -184 |
| 3 | -270 |
| 4 | -388 |
| 5 | -538 |
| 6 | -713 |
| 7 | -907 |
| 8 | -1112 |
| 9 | -1318 |
| 10 | -1514 |
| 11 | -1692 |
| 12 | -1841 |
| 13 | -1954 |
| 14 | -2024 |
| 15 | 30720 |
| 16 | -2024 |
| 17 | -1954 |
| 18 | -1841 |
| 19 | -1692 |
| 20 | -1514 |
| 21 | -1318 |
| 22 | -1112 |
| 23 | -907 |
| 24 | -713 |
| 25 | -538 |
| 26 | -388 |
| 27 | -270 |
| 28 | -184 |
| 29 | -132 |
| 30 | -111 |

## 5.3 Verilog Code

```
//////////////////////////////////////////////////////////////////////////////
//
// Conducting System
//
// Created: December 10, 2006
// Author: Andy Lin and Brandon Yoshimoto
//
//////////////////////////////////////////////////////////////////////////////
//
// Virtual Conducting System: Uses camera video input to affect the playback of
// music.  This is the top-level file which connects with the labkit


// Sample code template information:

// File:   zbt_6111_sample.v
// Date:   26-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Sample code for the MIT 6.111 labkit demonstrating use of the ZBT
// memories for video display.  Video input from the NTSC digitizer is
// displayed within an XGA 1024x768 window.  One ZBT memory (ram0) is used
// as the video frame buffer, with 8 bits used per pixel (black & white).
//
// Since the ZBT is read once for every four pixels, this frees up time for
// data to be stored to the ZBT during other pixel times.  The NTSC decoder
// runs at 27 MHz, whereas the XGA runs at 65 MHz, so we synchronize
// signals between the two (see ntsc2zbt.v) and let the NTSC data be
// stored to ZBT memory whenever it is available, during cycles when
// pixel reads are not being performed.
//
// We use a very simple ZBT interface, which does not involve any clock
// generation or hiding of the pipelining.  See zbt_6111.v for more info.
//

// MODIFICATIONS: This design includes 16 bits per pixel, and the ZBT is read once for every two
pixels instead.

`include "debounce.v"
`include "video_decoder.v"
`include "zbt_6111.v"
`include "ntsc2zbt.v"


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

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

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

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

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

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

               clock_feedback_out, clock_feedback_in,
```

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

            rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

            mouse_clock, mouse_data, keyboard_clock, keyboard_data,

            clock_27mhz, clock1, clock2,

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

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

            switch,

            led,

            user1, user2, user3, user4,

            daughtercard,

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

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

output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
input  ac97_bit_clock, ac97_sdata_in;

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

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

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

inout  [35:0] ram0_data;
output [18:0] ram0_address;
output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
output [3:0] ram0_bwe_b;

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;
```

```
   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

   output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
   input  disp_data_in;
   output  disp_data_out;

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

   inout [31:0] user1, user2, user3, user4;

   inout [43:0] daughtercard;

   inout  [15:0] systemace_data;
   output [6:0]  systemace_address;
   output systemace_ce_b, systemace_we_b, systemace_oe_b;
   input  systemace_irq, systemace_mpbrdy;

   output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
                 analyzer4_data;
   output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

   ////////////////////////////////////////////////////////////////////////
   //
   // I/O Assignments
   //
   ////////////////////////////////////////////////////////////////////////

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

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

   // Video Input
   //assign tv_in_i2c_clock = 1'b0;
   assign tv_in_fifo_read = 1'b1;
   assign tv_in_fifo_clock = 1'b0;
   assign tv_in_iso = 1'b1;
   //assign tv_in_reset_b = 1'b0;
   assign tv_in_clock = clock_27mhz;//1'b0;
   //assign tv_in_i2c_data = 1'bZ;
   // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
   // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

   // SRAMs

/* change lines below to enable ZBT RAM bank0 */

/*
   assign ram0_data = 36'hZ;
```

```
   assign ram0_address = 19'h0;
   assign ram0_clk = 1'b0;
   assign ram0_we_b = 1'b1;
   assign ram0_cen_b = 1'b0;  // clock enable
*/




   //assign vga_out_red = 10'h0;
   //assign vga_out_green = 10'h0;
   //assign vga_out_blue = 10'h0;
   //assign vga_out_sync_b = 1'b1;
   //assign vga_out_blank_b = 1'b1;
   //assign vga_out_pixel_clock = 1'b0;
   //assign vga_out_hsync = 1'b0;
   //assign vga_out_vsync = 1'b0;



/* enable RAM pins */

   assign ram0_ce_b = 1'b0;
   assign ram0_oe_b = 1'b0;
   assign ram0_adv_ld = 1'b0;
   assign ram0_bwe_b = 4'h0;

/**********/

  /* assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_clk = 1'b0;
   assign ram1_cen_b = 1'b1;
        */
   assign ram1_ce_b = 1'b0;
   assign ram1_oe_b = 1'b0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_bwe_b = 4'h0;

   assign clock_feedback_out = 1'b0;
   // clock_feedback_in is an input

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

   // RS-232 Interface
   assign rs232_txd = 1'b1;
   assign rs232_rts = 1'b1;
   // rs232_rxd and rs232_cts are inputs

   // PS/2 Ports
   // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

   // LED Displays

   assign disp_blank = 1'b1;
   assign disp_clock = 1'b0;
   assign disp_rs = 1'b0;
   assign disp_ce_b = 1'b1;
   assign disp_reset_b = 1'b0;
```

```
assign disp_data_out = 1'b0;

// disp_data_in is an input

// Buttons, Switches, and Individual LEDs
//lab3 assign led = 8'hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

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

// Daughtercard Connectors
assign daughtercard = 44'hZ;

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

// Logic Analyzer
assign analyzer1_data = 16'h0;
assign analyzer1_clock = 1'b1;

assign analyzer2_data = 16'h0;
assign analyzer2_clock = clock_27mhz;

assign analyzer3_data = 16'h0;
assign analyzer3_clock = 1'b1;
assign analyzer4_data = 16'h0;
assign analyzer4_clock = 1'b1;



    // VIDEO CODE BEGINS:

    // Camera input storage and retrieval:

//////////////////////////////////////////////////////////////////////////
// Demonstration of ZBT RAM as video memory

// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
wire clock_65mhz_unbuf,clock_65mhz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 10
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

wire clk = clock_65mhz;

// power-on reset generation
wire power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clk), .Q(power_on_reset),
               .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// ENTER button is user reset
wire reset,user_reset;
debounce db1(power_on_reset, clk, ~button_enter, user_reset);
assign reset = user_reset | power_on_reset;

 // generate basic XVGA video signals
```

```
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   xvga xvga1(clk,hcount,vcount,hsync,vsync,blank);

   // wire up to ZBT ram

   wire [35:0] vram_write_data;
   wire [35:0] vram_read_data;
   wire [18:0] vram_addr;
   wire        vram_we;

   zbt_6111 zbt1(clk, 1'b1, vram_we, vram_addr,
                 vram_write_data, vram_read_data,
                 ram1_clk, ram1_we_b, ram1_address, ram1_data, ram1_cen_b);

   // generate pixel value from reading ZBT memory
   wire [15:0]        vr_pixel;
   wire [18:0]        vram_addr1;

   vram_display vd1(reset,clk,hcount,vcount,vr_pixel,
                    vram_addr1,vram_read_data);

   // ADV7185 NTSC decoder interface code
   // adv7185 initialization module
   adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz),
                       .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
                       .tv_in_i2c_clock(tv_in_i2c_clock),
                       .tv_in_i2c_data(tv_in_i2c_data));

   wire [29:0] ycrcb;  // video data (luminance, chrominance)
   wire [2:0] fvh;     // sync for field, vertical, horizontal
   wire       dv;      // data valid

   ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
                       .tv_in_ycrcb(tv_in_ycrcb[19:10]),
                       .ycrcb(ycrcb), .f(fvh[2]),
                       .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

   // code to write NTSC data to video memory

   wire [18:0] ntsc_addr;
   wire [35:0] ntsc_data;
   wire        ntsc_we;
   ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, ycrcb[29:0],
                    ntsc_addr, ntsc_data, ntsc_we, 1'b0);//switch[6]);

   // code to write pattern to ZBT memory
   reg [31:0]  count;
   always @(posedge clk) count <= reset ? 0 : count + 1;

   wire [18:0]        vram_addr2 = count[0+18:0];
   wire [35:0]        vpat = ( (1'b1) ? {4{count[3+3:3],4'b0}}
                         : {4{count[3+4:4],4'b0}} );    //switch[1]

   // mux selecting read/write to memory based on which write-enable is chosen

   wire        sw_ntsc = 1; // ~switch[7];
   wire        my_we = sw_ntsc ? (hcount[0]==1'd0) : blank;
   wire [18:0]        write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
   wire [35:0]        write_data_1 = sw_ntsc ? ntsc_data : vpat;

//   wire      write_enable = sw_ntsc ? (my_we & ntsc_we) : my_we;
//   assign    vram_addr = write_enable ? write_addr : vram_addr1;
//   assign    vram_we = write_enable;

   assign      vram_addr = my_we ? write_addr : vram_addr1;
   assign      vram_we = my_we;
   assign      vram_write_data = write_data_1;

   // select output pixel data
```

```
   wire        b,hs,vs;

   delayN dn1(clk,hsync,hs);  // delay by 3 cycles to sync with ZBT read
   delayN dn2(clk,vsync,vs);
   delayN dn3(clk,blank,b);


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

       // Video Modules:

       // Modules for debouncing the button_up and button_down signals:
       wire bup, bdown;
   debounce bup_mod(reset, clock_27mhz, button_up, bup);
   debounce bdown_mod(reset, clock_27mhz, button_down, bdown);

       // MODULE (color_decision): Decides if a pixel is of the desired color.  Also outputs
camera video image as RGB

       wire pixel_video;                          // 1 if current pixel is of the desired
color
       wire [23:0] cam_image;        // Camera input image in format {R,G,B}

       color_decision color_decision1(
       clk,reset,vr_pixel,pixel_video,cam_image,switch[7:0]);

       // MODULE (video_processor): Takes in camera video and outputs coordinates of hands

       wire [10:0] left_x, right_x;  // The average x positions of left and right hands
respectively
       wire [9:0]     left_y,right_y;        // The average y positions of left and right hands
       respectively
       wire                   desired_color;         // desired_color = 1 if the current pixel is
registered as blue

       video_processor vp(    reset, clock_27mhz,
                                                          pixel_video,
                                                          hcount, vcount, vsync,
                                                          left_x, left_y,
                                                          right_x, right_y,
                                                          desired_color);

       // MODULE (motion_analyzer): Takes in hand positions and analyzes motion over time

       wire                           beat_start;
                              // High for one clock cycle when beat starts, low otherwise.
       wire                           beat_end;
                              // High for one clock cycle when beat ends, low otherwise.
       wire [10:0]    beat_start_x, beat_end_x;                                              // The
x position at the start and end of a beat respectively
       wire [9:0]             beat_start_y, beat_end_y;
       // The y position at the start and end of a beat respectively
       wire [10:0]    amp_left, amp_right;
       // The distance in pixels between a beat start and end
       wire [10:0]            beat_period;
              // A count of how many 0.04s intervals a beat takes
       wire [10:0]            acceleration_left, acceleration_right;              // A measure
of the calculated acceleration for a beat

       motion_analyzer motion_analy1(clock_27mhz,reset,
                                                                                  left_x,left_y,

       right_x,right_y,

       beat_start,beat_end,

       beat_start_x,beat_start_y,

       beat_end_x,beat_end_y,
```

```
        amp_left,amp_right,
                                                                                beat_period,

        acceleration_left,acceleration_right,
                                                                                switch[7:6],
switch[5:4], switch[3:2], switch[1:0]);

        // MODULE (generate_visualization): Decides what to output to the screen

        // The output wires to the vga display:
        wire [7:0]     display_out_r, display_out_g, display_out_b;

        generate_visualization gen_vis1(clk,clock_27mhz,reset,hcount,vcount,

        bup,bdown,cam_image,
                                                                                left_x,
left_y, right_x, right_y,

        beat_start_x, beat_start_y,

        beat_end_x,beat_end_y,

        amp_left, amp_right,

        beat_period,

        acceleration_left,acceleration_right,

        desired_color,

        display_out_r, display_out_g, display_out_b);

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.
   assign vga_out_red  =       display_out_r;
   assign vga_out_green        =       display_out_g;
   assign vga_out_blue =       display_out_b;
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_pixel_clock = ~clock_65mhz;
   assign vga_out_blank_b = ~b;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;


////////////////////////////////////////////////////////////////////////////////////////
        // AUDIO

   wire [7:0] from_ac97_data, to_ac97_data;
   wire ready;

   // allow user to adjust volume
   wire vup,vdown;
   reg old_vup,old_vdown;
   debounce bright(reset, clock_27mhz, ~button_right, vup);
   debounce bleft(reset, clock_27mhz, ~button_left, vdown);
        debounce bbeat(reset, clock_27mhz, ~button0, beatprelim);

//      reg beat;
//      reg beatprelimold;
//      always @ (posedge clock_27mhz) begin
//              beatprelimold <= beatprelim;
//              beat <= (beatprelim != beatprelimold)        & beatprelim;
//      end

        wire [6:0] amp_left_fixed;
        wire [6:0] amp_right_fixed;
        wire [4:0] volume;
  // reg [4:0] volume;

 //  always @ (posedge clock_27mhz) begin
```

```
//     if (reset) volume <= 5'd8;
//     else begin
//      if (vup & ~old_vup & volume != 5'd31) volume <= volume+1;
//      if (vdown & ~old_vdown & volume != 5'd0) volume <= volume-1;
//     end
//          old_vup <= vup;
//          old_vdown <= vdown;


//               if (((amp_left_fixed + amp_right_fixed) >> 2) >= 5'b11111)
//                       volume <= 5'b11111;
//                     else
//                         volume <= (amp_left_fixed + amp_right_fixed)>>2;
 //     end
//      end
   // AC97 driver
   lab4audio a(clock_27mhz, reset, volume, from_ac97_data, to_ac97_data, ready,
               audio_reset_b, ac97_sdata_out, ac97_sdata_in,
               ac97_synch, ac97_bit_clock);

   // push ENTER button reset
   wire playback;
   debounce benter(reset, clock_27mhz, button_enter, playback);
        debounce bone(reset, clock_27mhz, ~button1, program);
        debounce btwo(reset, clock_27mhz, ~button2, which);
   debounce bthree(reset, clock_27mhz, ~button3, reset_metronome);
   // light up LEDs when recording, show volume during playback.
   // led is active low
   assign led = playback ? ~{3'b000, volume} : 8'h00;

        wire we, din, dout;
        wire [22:0] addr;
        wire [2:0] counter;

   wire weZBT;
        wire [18:0] addrZBT;

   defparam reset_sr.INIT = 16'hFFFF;


        wire [1:0] fop;
   wire [22:0] faddress;
   wire [15:0] fwdata, frdata;
   wire fbusy;
   wire [639:0] dots;
   wire [7:0] beataudio;
        wire [5:0] zbtState;
        wire [35:0] write_data, read_data;
        wire continue;
        wire access_enable;
        wire signed [15:0] addr_mod;
        wire [2:0] divisionCount;
        wire [2:0] skip;
        wire [2:0] interval;
        wire beginning;
        wire add;
        wire [10:0] TempBeatPeriod, BeatPeriod, NewBeatPeriod;
        //reg [15:0] offset = 5000;
        //reg [15:0] musicbeatperiod = 21900;
        wire [15:0] beatcount;
   wire tempbeatcount;

   wire nd, rdy, rfd , rdyhp, rfdhp;            //hp lp stuff
        wire signed [7:0] to_filter;
        wire signed [28:0] filter_out, filter_outhp;
        wire signed [8:0] sum;
        reg signed [7:0] low_out;
        reg signed [7:0] high_out;
        wire [15:0] offset;
        wire [15:0] musicbeatperiod;
   wire firstbeat;
```

```
        assign beat = beat_start;
        wire [1:0] acceleration_left_out, acceleration_right_out;

        //module test(samplecount, musicbeat);
        //test mytest(beatcount, musicbeat);

       //module BeatGenerator2(reset, clock_27mhz, beginning, offset, samplecount, access_enable,
access_reset, musicbeat, beatperiod, toggle);
       BeatGenerator2 myBeatGenerator(reset|~playback, clock_27mhz, beginning, offset, beatcount,
sample_access, access_reset, musicbeat, musicbeatperiod, switch[2]);

        //module ROMFSM(clock_27mhz, reset, playback, ready, from_ac97_data, outdata, switch, we,
din, dout, addr, counter, playbackchange, faddress, frdata, fwdata, fbusy, fop, dots, beat,
musicbeat, continue, beginning, offset, beatperiod, access_enable, access_reset, samplecount);
        ROMFSM myROMFSM(clock_27mhz, ~playback|reset, playback, ready, from_ac97_data, beataudio,
1, we, din, dout, addr, counter, playbackchange, faddress, frdata, fwdata, fbusy, fop, dots, beat,
musicbeat, continue, beginning, offset, musicbeatperiod, sample_access, access_reset, beatcount,
firstbeat);

        //module ZBTFSM(reset, clock_27mhz, inaudio, outaudio, beat, musicbeat, we, write_data,
read_data, addr, ready);
        ZBTFSM myZBTFSM(~playback|reset, clock_27mhz, beataudio, to_filter, beat, musicbeat,
weZBT, write_data, read_data, addrZBT, ready, zbtState, continue, access_enable, addr_mod,
BeatPeriod, 1, 0, musicbeatperiod, firstbeat);

        //****the Division Converter and the Division Counter make up the Tempo Modulator
"Module"
        //DivisionCounter myDivisionCounter(reset, clock_27mhz, access_enable, addr_mod,
switch[5:3], switch[2:0], switch[6], divisionCount);
        DivisionCounter myDivisionCounter(~playback|reset, clock_27mhz, access_enable, addr_mod,
skip, interval, add, divisionCount);

        //module DivisionConverter(reset, clock_27mhz, BeatPeriod, beat, access_enable,
OriginalBeatPeriod, skip, interval, add, NewBeatPeriod, NewOriginalBeatPeriod);
        DivisionConverter myDivisionConverter(reset, clock_27mhz, BeatPeriod, beat,
OriginalBeatPeriod, skip, interval, add, NewBeatPeriod, NewOriginalBeatPeriod, TempBeatPeriod,
musicbeatperiod);

        //module BeatPeriodCounter(reset, clock_27mhz, enable, beat, BeatPeriod, offset,
beginning);
        BeatPeriodCounter myBeatPeriodCounter(~playback|reset, clock_27mhz, ready, beat,
BeatPeriod, beginning);

        //lp and hp filters
        lpfilter lpf (ready, rdy, clock_27mhz, rfd, to_filter, filter_out);
        hpfilter hpf (ready, rdyhp, clock_27mhz, rfdhp, to_filter, filter_outhp);


        //module SignalTamer (reset, clock_27mhz, volume_in, volume_out, acceleration_in,
acceleration_out,
//                                          beat, sample_count, musicbeatperiod,
interpolationswitch);

        SignalTamer SignalTamerLeft(reset, clock_27mhz, amp_left,
        amp_left_fixed, acceleration_left, acceleration_left_out, beat, addrZBT, musicbeatperiod,
switch[5]);

        SignalTamer SignalTamerRight(reset, clock_27mhz, amp_right,
        amp_right_fixed, acceleration_right, acceleration_right_out, beat, addrZBT,
musicbeatperiod, switch[5]);


//      ArticulationVolumeModulator myArticulationVolumeModulator(clock_27mhz, reset, filter_out,
filter_outhp,
//                                      to_ac97_data, {9'b0, switch[1:0]}, {9'b0, switch[3:2]},
amp_left_fixed,
//
amp_right_fixed, musicbeatperiod, rfd, rfdhp, {7'b0, switch[1]}, switch[0], addrZBT, switch[7],
volume);
//
```

```
       ArticulationVolumeModulator myArticulationVolumeModulator(clock_27mhz, reset, filter_out,
filter_outhp,
                                        to_ac97_data, acceleration_left_out, acceleration_right_out,
amp_left_fixed,

amp_right_fixed, musicbeatperiod, rfd, rfdhp, {7'b0, switch[1]}, switch[0], addrZBT, switch[7],
volume, switch[6]);

   //module MetronomeProgrammer(clock_27mhz, reset, program_select, program, value,
music_beat_period,
   //                           offset);
   MetronomeProgrammer MyMetronomeProgrammer(clock_27mhz, reset|reset_metronome, which, program,
switch[7:0], musicbeatperiod,
                              offset);




       // output useful things to the logic analyzer connectors
   //assign analyzer1_clock = clock_27mhz;
        // assign analyzer1_clock = ac97_bit_clock;
       // assign analyzer1_data[0] = audio_reset_b;
       // assign analyzer1_data[1] = ac97_sdata_out;
       // assign analyzer1_data[2] = ac97_sdata_in;
       // assign analyzer1_data[3] = ac97_synch;
   //assign analyzer1_data[15:8] = read_data[7:0];
   //assign analyzer1_data[15:8] = addr_mod[9:2];

       //assign analyzer1_data[15:8] = NewBeatPeriod[7:0];
   //assign analyzer1_data[7:0] = write_data;
       //assign analyzer1_data[7:7] = add;
       //assign analyzer1_data[6] = beat;
       //assign analyzer1_data[5:3] = interval;
       //assign analyzer1_data[2:0] = skip;

       //assign analyzer1_data[5:0] = zbtState;

   //assign analyzer2_clock = clock_27mhz;
       //assign analyzer2_data = {from_ac97_data[7:1], ready, to_ac97_data};
       //assign analyzer2_data = {beataudio, addrZBT[7:0]};
       //assign analyzer4_data[0] = we;
       //assign analyzer4_data[15:5] = 0;
       //assign analyzer4_data[3:1] = counter;
//     assign analyzer4_data[4] =   playbackchange;


   //ZBT Stuff
       zbt_6111 myzbt(clock_27mhz, 1'b1, weZBT, addrZBT, write_data, read_data, ram0_clk,
ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

       //ROM Stuff
   flash_int flashint1 (reset, clock_27mhz, fop, faddress, fwdata, frdata,
                        fbusy, flash_data, flash_address, flash_ce_b,
                        flash_oe_b, flash_we_b, flash_reset_b, 1'b1,
                        flash_byte_b);







endmodule



//////////////////////////////////////////////////////////////////////////////
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
```

```
module xvga(vclock,hcount,vcount,hsync,vsync,blank);
   input vclock;
   output [10:0] hcount;
   output [9:0] vcount;
   output      vsync;
   output      hsync;
   output      blank;

   reg    hsync,vsync,hblank,vblank,blank;
   reg [10:0]   hcount;    // pixel number on current line
   reg [9:0] vcount;   // line number

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   wire      hsyncon,hsyncoff,hreset,hblankon;
   assign    hblankon = (hcount == 1023);
   assign    hsyncon = (hcount == 1047);
   assign    hsyncoff = (hcount == 1183);
   assign    hreset = (hcount == 1343);

   // vertical: 806 lines total
   // display 768 lines
   wire      vsyncon,vsyncoff,vreset,vblankon;
   assign    vblankon = hreset & (vcount == 767);
   assign    vsyncon = hreset & (vcount == 776);
   assign    vsyncoff = hreset & (vcount == 782);
   assign    vreset = hreset & (vcount == 805);

   // sync and blanking
   wire      next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always @(posedge vclock) begin
      hcount <= hreset ? 0 : hcount + 1;
      hblank <= next_hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

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

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

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


//ROM constants

`define STATUS_RESET             4'h0
`define STATUS_READ_ID           4'h1
`define STATUS_CLEAR_LOCKS       4'h2
`define STATUS_ERASING           4'h3
`define STATUS_WRITING           4'h4
`define STATUS_READING           4'h5
`define STATUS_SUCCESS           4'h6
`define STATUS_BAD_MANUFACTURER  4'h7
`define STATUS_BAD_SIZE          4'h8
`define STATUS_LOCK_BIT_ERROR    4'h9
`define STATUS_ERASE_BLOCK_ERROR 4'hA
`define STATUS_WRITE_ERROR       4'hB
`define STATUS_READ_WRONG_DATA   4'hC

`define NUM_BLOCKS 128
`define BLOCK_SIZE 64*1024
`define LAST_BLOCK_ADDRESS ((`NUM_BLOCKS-1)*`BLOCK_SIZE)
`define LAST_ADDRESS (`NUM_BLOCKS*`BLOCK_SIZE-1)

`define FLASHOP_IDLE  2'b00
```

```
`define FLASHOP_READ  2'b01
`define FLASHOP_WRITE 2'b10

////////////////////////////////////////////////////////////////////////////
//ROM FSM reads from the flash ROM one beat at a time.      Beat specifies when a new user
specified
//beat has arrived.  ROM FSM will stop reading after musicbeatperiod of samples has been accessed
//and interacts with BeatGenerator to determine this.  ROM FSM also interacts with RAM FSM to
output
//1 beat of audio at one time.  ROM FSM will also output useful data such as beginning, firstbeat,
//access_enable, access_reset.
////////////////////////////////////////////////////////////////////////////

module ROMFSM(clock_27mhz, reset, playback, ready, from_ac97_data, outdata, switch, we, din, dout,
       addr, counter, playbackchange, faddress, frdata, fwdata, fbusy, fop, dots, beat,
musicbeat,
       continue, beginning, offset, beatperiod, access_enable, access_reset, samplecount,
firstbeat);
  parameter SEGMENTS = 16;                    //number of segments of 65536 samples stored in
the ROM.

       input [15:0]  frdata;                  //ROM interaction
   input fbusy;                                          //ROM interaction
       input [15:0] offset;
       input [15:0] beatperiod;
       input clock_27mhz;              // 27mhz system clock
   input reset;                    // 1 to reset to initial state
   input playback;                 // true when playback is enabled (always).  if playback is
false, playback is reset to address 0.
   input ready;                    // debug input
   input [7:0] from_ac97_data;     // debug input
       input switch;                                         // turns on and off interpolation.
Feature taken out
       input beat;
   output signed [7:0] outdata;      // 8-bit PCM data (6.75 MHz) to rest of system
       input musicbeat;
   output beginning;
       output continue;                                      // high when the next beat should
be outputted

       //ROM Stuff
   output [1:0] fop;
   output [22:0] faddress;
   output [15:0] fwdata;
   output [639:0] dots;                                      //debug output
       output firstbeat;                                            //lets other modules know
when first beat is
       output we;
       output [15:0] addr;
       output [7:0] dout, din;
       output [0:0] counter;
   output playbackchange;
       output access_enable, access_reset;
       output [15:0] samplecount;

   reg [1:0] fop;
   reg [22:0] faddress = 0;
   reg [15:0] fwdata;
   reg [639:0] dots;
       reg [7:0] state = 5;
   reg [3:0] status;
       reg signed [7:0] outdata;
       reg we;
       //reg [7:0] din, dout;
       reg [15:0] addr = 0;
       reg  counter = 0;
       reg [15:0] lastaddr = 0;
       reg [7:0] insample = 0;
       reg signed [7:0] outsample = 0;
       //previous sample used for potential anti-aliasing
       reg signed [7:0] oldsample = 0;
```

```
        reg [7:0] to_ac97_data = 0;
        reg lastplayback = 0;
        reg statewrite,  switchwriteold;
        reg [15: 0] samplecount = 0;
        reg continue = 1;
        reg read = 1;
        wire musicbeat2;
        wire beginning;
        wire switchwritechange;
    wire musicbeat;
        wire [7:0] din, dout;
        wire playbackchange;

    always @ (posedge clock_27mhz) begin
                //reset the memory address when playback first goes high
                continue <= (beat? 1: musicbeat? 0: continue);
                if (~playback|beginning) begin
                        addr <= 0;
                        faddress <= 0;
                        counter <=0;
                        state <= 5;
                        samplecount <= offset;
                        continue <= 1;
                        status <= `STATUS_READING;
                        fwdata <= 16'hFF;             // Issue "read array" command
                        fop <= `FLASHOP_WRITE;
                end
                        case (state)
                                0:
                                        //continue to read from ROM after beat is declared
                                        if (continue) begin
                                                samplecount <=  samplecount + 1;
                                                state <= 4;
                                                status <= `STATUS_READING;
                                             fop <= `FLASHOP_READ;
                                          end
                                          else begin
                                                samplecount <= 0;
                                          end
                                 4:     //delay to allow right data to be read from ROM
                                   begin
                                      state <= state+1;
                                   end
                                 5:
                                   begin
                                      state <= state+1;
                                   end
                                 6:
                                   begin
                                                faddress <= (faddress >= 65536*SEGMENTS - 1)? 0:
faddress + 1;
                                                outdata <= frdata[7:0];
                                                state <= 0;
                                          end
                                default:
                                          fop <= `FLASHOP_IDLE;
                        endcase
    end

assign access_enable = continue & (state == 0);                                 //true when
new sample is accessed
assign access_reset = ~continue & (state == 0);                                 //true when
access counts should be reset
assign musicbeat2 = (samplecount == beatperiod);
assign beginning = (faddress == 0);
        //specifies beginning of ROM
assign firstbeat = (faddress <= beatperiod - offset+4);     //specifies first beat
endmodule

////////////////////////////////////////////////////////////////////////////
//ZBTFSM  writes to ZBT SRAM and reads from ZBT SRAM.  During the write cycle, data
```

```
//is streamed in from ROM FSM at 6.75 MHz.  During the read cycle, data is streamed out
//at 24 KHz.  ZBT FSM interacts with ROM FSM to read out data music one beat
//at a time.  ZBT FSM interacts with Tempo Modulator Module, which provides addr_mod to let
//ZBTFSM know how many addresses to skip or replay in order to modulate the tempo.
//ZBTFSM also creates "note extension" which replays samples at the end of each beat to create
//continuous playback.  ExtendOn and Extend2 allow 2 different modes of note extension to be
enabled
//inaudio will be the input from the ROM FSM, outaudio will be the output to Volume and
Articulation Modulator
//////////////////////////////////////////////////////////////////////////////

module ZBTFSM(reset, clock_27mhz, inaudio, outaudio, beat, musicbeat, we, write_data, read_data,
addr, ready, state, continue, access_enable,addr_mod, BeatPeriod, ExtendOn, ExtendOn2,
musicbeatperiod, firstbeat);

       input continue;
       input reset, clock_27mhz;
       input   [7:0] inaudio;                          //from ROM FSM
       input [35:0] read_data;                              //interaction with ZBT SRAM
       input beat, musicbeat;
       input ready;
       input [10:0] BeatPeriod;                       //Beat period (user specified)
       input firstbeat;                                       //true when first beat is
being written in
       input signed [15:0] addr_mod;
       input ExtendOn, ExtendOn2;
       input [15:0] musicbeatperiod;
       output access_enable;                                          //true when a new address is
read out
       output [5:0] state;
       output  [7:0] outaudio;                                        //audio output at 24 KHz
       output we;
       output [35:0] write_data;
       output [18:0] addr;

       reg [35:0] write_data;
       reg signed [7:0] outaudio;
       reg read, we;
   reg [18:0] tempoaddr = 0;
       reg [18:0] mainfsmaddr = 0;
       reg [18:0] lasttempoaddr = 0;
       reg readcontinue= 1;
       reg [5:0] state =0;
       reg count = 0;
       reg [7:0] temp_data;
       reg [18:0] readAddr = 0;
       reg [18:0] writeAddr = 0;
       reg mode= 1;
       reg signed [7:0] oldsample = 0;
       reg firstbeatlatch;                          //latches first beat to use later in cycle
       wire access_enable;

       always @ (posedge clock_27mhz) begin
                readcontinue <= (beat? 1: (~continue & readAddr >= musicbeatperiod)? 0:
readcontinue);
                if (reset) begin
                       state <= 0;
                       writeAddr <= 0;
                       readAddr <= 0;
                       mode <= 1;
                       oldsample <= 0;
                end
                if (beat) begin
                       writeAddr <= 0;
                       readAddr <= 0;
                       state <= 0;
                end
                else

                case (state)
                0:                    begin
```

```
                        //write into RAM
                        if (continue) begin
                                we <= 1;
                                //using 8 bit audio format for now
                                write_data[7:0] <= inaudio;
                                write_data[35:8] <= 0;
                                state <= 1;
                                firstbeatlatch <= firstbeat;
                         end
                         else
                               state <= 5;
                 end
                1:  //delay
                        state <= 3;

                3: state <= state+1;
                4:  begin
                         if (~continue) begin
                                state <= state + 1;
                         end

                         else begin
                                writeAddr <= writeAddr + 1;
                                state <= 0;
                         end
                 end
                 5: begin
                                //playback
                                we <= 0;
                                if (ready) begin                        //only play back at each
ready cycle from AC97
                                       if (count == 0)       begin     //only increment when
count == 0 for 24 KHz
                                              if (readAddr + 1+ addr_mod <= 0)
                                                     readAddr <= 0;
                                              else
                                                     if (readcontinue)             //only
increment when on read mode
                                                            readAddr <= (readAddr >=
musicbeatperiod - firstbeatlatch * 5000)? - firstbeatlatch * 5000: readAddr + 1+ addr_mod;
                                                     else if (~continue)  begin
                                                            //note extension      option 1.
Plays last few thousand samples forward and backward
                                                            //firstbeatlatch deals with when we
are at the first beat - exceptions.
                                                        if (ExtendOn)       begin
                                                                   if (readAddr >=
musicbeatperiod-500 - firstbeatlatch * 5000) begin
                                                                          mode <= 0;
                                                                          readAddr <=
musicbeatperiod-501 - firstbeatlatch * 5000;
                                                                   end
                                                                   else if (readAddr <=
musicbeatperiod - 3000 - firstbeatlatch * 5000)       begin
                                                                          mode <= 1;
                                                                          readAddr <=
musicbeatperiod-2999 - firstbeatlatch * 5000;
                                                                   end
                                                                   else begin
                                                                          readAddr <= mode?
readAddr + 1: readAddr - 1;
                                                                          mode <= mode;
                                                                   end
                                                            end
                                                            //note extension option 2
                                                            else if (ExtendOn2)
                                                                   readAddr <= (readAddr >=
musicbeatperiod - 500)? musicbeatperiod-3000: readAddr + 1;
                                                            else
                                                                   readAddr <= readAddr;
                                                     end
```

```
                                        oldsample <= temp_data;
                                        temp_data <= read_data[7:0];
                                    end
                                    outaudio <= temp_data;
                                    count <= count +1;
                                end
                end
                endcase

        end

        assign access_enable = ready & (count==0) & readcontinue;           //specifies when a
new address in RAM is addressed
        assign addr = continue? writeAddr: readAddr;
//toggles between writing and reading

endmodule


//////////////////////////////////////////////////////////////////////////
// generate display pixels from reading the ZBT ram
// note that the ZBT ram has 2 cycles of read (and write) latency
//
// We take care of that by latching the data at an appropriate time.
//
// Note that the ZBT stores 36 bits per word; we use only 32 bits here,
// decoded into four bytes of pixel data.

// *Modifications for handling storage of two 16-bit pieces of data per pixel

module vram_display(reset,clk,hcount,vcount,vr_pixel,
                    vram_addr,vram_read_data);

   input reset, clk;
   input [10:0] hcount;
   input [9:0]        vcount;
   output [15:0] vr_pixel;
   output [18:0] vram_addr;
   input [35:0]  vram_read_data;

   wire [18:0]         vram_addr = {vcount, ~hcount[9:1]};

   wire                        hc2 = ~hcount[0];
   reg [15:0]   vr_pixel;
   reg [35:0]   vr_data_latched;
   reg [35:0]   last_vr_data;

   always @(posedge clk)
     last_vr_data <= (hc2==1'd1) ? vr_data_latched : last_vr_data;

   always @(posedge clk)
     vr_data_latched <= (hc2==1'd0) ? vram_read_data : vr_data_latched;

   always @(*)        // each 36-bit word from RAM is decoded to 4 bytes
     case (hc2)
       1'd1: vr_pixel = last_vr_data[15:0];
       1'd0: vr_pixel = last_vr_data[15+16:0+16];
     endcase

endmodule // vram_display


//////////////////////////////////////////////////////////////////////////
// parameterized delay line

module delayN(clk,in,out);
   input clk;
   input in;
   output out;

   parameter NDELAY = 3;
```

```
   reg [NDELAY-1:0] shiftreg;
   wire            out = shiftreg[NDELAY-1];

   always @(posedge clk)
     shiftreg <= {shiftreg[NDELAY-2:0],in};

endmodule // delayN



//////////////////////////////////////////////////////////////////////////////
//
// bi-directional monaural interface to AC97
//
//////////////////////////////////////////////////////////////////////////////

module lab4audio (clock_27mhz, reset, volume,
                  audio_in_data, audio_out_data, ready,
                 audio_reset_b, ac97_sdata_out, ac97_sdata_in,
                  ac97_synch, ac97_bit_clock);

   input clock_27mhz;
   input reset;
   input [4:0] volume;
   output [7:0] audio_in_data;
   input [7:0] audio_out_data;
   output ready;

   //ac97 interface signals
   output audio_reset_b;
   output ac97_sdata_out;
   input ac97_sdata_in;
   output ac97_synch;
   input ac97_bit_clock;

   wire [2:0] source;
   assign source = 0;    //mic

   wire [7:0] command_address;
   wire [15:0] command_data;
   wire command_valid;
   wire [19:0] left_in_data, right_in_data;
   wire [19:0] left_out_data, right_out_data;

   reg audio_reset_b;
   reg [9:0] reset_count;

   //wait a little before enabling the AC97 codec
   always @(posedge clock_27mhz) begin
      if (reset) begin
         audio_reset_b = 1'b0;
         reset_count = 0;
      end else if (reset_count == 1023)
        audio_reset_b = 1'b1;
      else
        reset_count = reset_count+1;
   end

   wire ac97_ready;
   ac97 ac97(ac97_ready, command_address, command_data, command_valid,
             left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,
             right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
             ac97_bit_clock);

   // ready: one cycle pulse synchronous with clock_27mhz
   reg [2:0] ready_sync;
   always @ (posedge clock_27mhz) begin
     ready_sync <= {ready_sync[1:0], ac97_ready};
   end
   assign ready = ready_sync[1] & ~ready_sync[2];
```

```
   reg [7:0] out_data;
   always @ (posedge clock_27mhz)
     if (ready) out_data <= audio_out_data;
   assign audio_in_data = left_in_data[19:12];
   assign left_out_data = {out_data, 12'b000000000000};
   assign right_out_data = left_out_data;

   // generate repeating sequence of read/writes to AC97 registers
   ac97commands cmds(clock_27mhz, ready, command_address, command_data,
                     command_valid, volume, source);
endmodule

// assemble/disassemble AC97 serial frames
module ac97 (ready,
             command_address, command_data, command_valid,
             left_data, left_valid,
             right_data, right_valid,
             left_in_data, right_in_data,
             ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

   output ready;
   input [7:0] command_address;
   input [15:0] command_data;
   input command_valid;
   input [19:0] left_data, right_data;
   input left_valid, right_valid;
   output [19:0] left_in_data, right_in_data;

   input ac97_sdata_in;
   input ac97_bit_clock;
   output ac97_sdata_out;
   output ac97_synch;

   reg ready;

   reg ac97_sdata_out;
   reg ac97_synch;

   reg [7:0] bit_count;

   reg [19:0] l_cmd_addr;
   reg [19:0] l_cmd_data;
   reg [19:0] l_left_data, l_right_data;
   reg l_cmd_v, l_left_v, l_right_v;
   reg [19:0] left_in_data, right_in_data;

   initial begin
      ready <= 1'b0;
      // synthesis attribute init of ready is "0";
      ac97_sdata_out <= 1'b0;
      // synthesis attribute init of ac97_sdata_out is "0";
      ac97_synch <= 1'b0;
      // synthesis attribute init of ac97_synch is "0";

      bit_count <= 8'h00;
      // synthesis attribute init of bit_count is "0000";
      l_cmd_v <= 1'b0;
      // synthesis attribute init of l_cmd_v is "0";
      l_left_v <= 1'b0;
      // synthesis attribute init of l_left_v is "0";
      l_right_v <= 1'b0;
      // synthesis attribute init of l_right_v is "0";

      left_in_data <= 20'h00000;
      // synthesis attribute init of left_in_data is "00000";
      right_in_data <= 20'h00000;
      // synthesis attribute init of right_in_data is "00000";
   end

   always @(posedge ac97_bit_clock) begin
```

```
      // Generate the sync signal
      if (bit_count == 255)
        ac97_synch <= 1'b1;
      if (bit_count == 15)
        ac97_synch <= 1'b0;

      // Generate the ready signal
      if (bit_count == 128)
        ready <= 1'b1;
      if (bit_count == 2)
        ready <= 1'b0;

      // Latch user data at the end of each frame. This ensures that the
      // first frame after reset will be empty.
      if (bit_count == 255)
        begin
           l_cmd_addr <= {command_address, 12'h000};
           l_cmd_data <= {command_data, 4'h0};
           l_cmd_v <= command_valid;
           l_left_data <= left_data;
           l_left_v <= left_valid;
           l_right_data <= right_data;
           l_right_v <= right_valid;
        end

      if ((bit_count >= 0) && (bit_count <= 15))
        // Slot 0: Tags
        case (bit_count[3:0])
          4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
          4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
          4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
          4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
         4'h4: ac97_sdata_out <= l_right_v; // Right data valid
          default: ac97_sdata_out <= 1'b0;
        endcase

      else if ((bit_count >= 16) && (bit_count <= 35))
        // Slot 1: Command address (8-bits, left justified)
        ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;

      else if ((bit_count >= 36) && (bit_count <= 55))
        // Slot 2: Command data (16-bits, left justified)
        ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;

      else if ((bit_count >= 56) && (bit_count <= 75))
        begin
           // Slot 3: Left channel
           ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
           l_left_data <= { l_left_data[18:0], l_left_data[19] };
        end
      else if ((bit_count >= 76) && (bit_count <= 95))
        // Slot 4: Right channel
           ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
      else
        ac97_sdata_out <= 1'b0;

      bit_count <= bit_count+1;

   end // always @ (posedge ac97_bit_clock)

   always @(negedge ac97_bit_clock) begin
      if ((bit_count >= 57) && (bit_count <= 76))
        // Slot 3: Left channel
        left_in_data <= { left_in_data[18:0], ac97_sdata_in };
      else if ((bit_count >= 77) && (bit_count <= 96))
        // Slot 4: Right channel
        right_in_data <= { right_in_data[18:0], ac97_sdata_in };
   end

endmodule
```

```
// issue initialization commands to AC97
module ac97commands (clock, ready, command_address, command_data,
                     command_valid, volume, source);

   input clock;
   input ready;
   output [7:0] command_address;
   output [15:0] command_data;
   output command_valid;
   input [4:0] volume;
   input [2:0] source;

   reg [23:0] command;
   reg command_valid;

   reg [3:0] state;

   initial begin
      command <= 4'h0;
      // synthesis attribute init of command is "0";
      command_valid <= 1'b0;
      // synthesis attribute init of command_valid is "0";
      state <= 16'h0000;
      // synthesis attribute init of state is "0000";
   end

   assign command_address = command[23:16];
   assign command_data = command[15:0];

   wire [4:0] vol;
   assign vol = 31-volume;  // convert to attenuation

   always @(posedge clock) begin
      if (ready) state <= state+1;

      case (state)
        4'h0: // Read ID
          begin
             command <= 24'h80_0000;
             command_valid <= 1'b1;
          end
        4'h1: // Read ID
          command <= 24'h80_0000;
        4'h3: // headphone volume
          command <= { 8'h04, 3'b000, vol, 3'b000, vol };
        4'h5: // PCM volume
          command <= 24'h18_0808;
        4'h6: // Record source select
          command <= { 8'h1A, 5'b00000, source, 5'b00000, source};
        4'h7: // Record gain = max
         command <= 24'h1C_0F0F;
        4'h9: // set +20db mic gain
          command <= 24'h0E_8048;
        4'hA: // Set beep volume
          command <= 24'h0A_0000;
        4'hB: // PCM out bypass mix1
          command <= 24'h20_8000;
        default:
          command <= 24'h80_0000;
      endcase // case(state)
   end // always @ (posedge clock)
endmodule // ac97commands
```

---

```
//////////////////////////////////////////////////////////////////////
//
// analyzer_bar: generate rectangle on screen with adjustable width
//                                          x, y determine the upper left hand corner of block
```

```
//
//////////////////////////////////////////////////////////////////////
module analyzer_bar(x,y,width,hcount,vcount,pixel);
   parameter HEIGHT = 30;     // default height: 30 pixels
   parameter COLOR_R = 8'd0;
   parameter COLOR_G = 8'd255;
   parameter COLOR_B = 8'd0;

   input [10:0] x,hcount,width;  // Variable width in this sprite
   input [9:0] y,vcount;
   output [23:0] pixel;

   reg [23:0] pixel;
   always @ (x or y or hcount or vcount or width) begin
      if ((hcount >= x && hcount < (x+width)) &&
         (vcount >= y && vcount < (y+HEIGHT)))
       pixel = {COLOR_R,COLOR_G,COLOR_B};
      else pixel = 0;
   end
endmodule
```

---

```
//////////////////////////////////////////////////////////////////////
//
// beat_markers: Takes in coordinates of right hand and decides when
//                                          a beat starts and ends.  Outputs beat_start and
beat_end
//                                          are each high for one clock cycle when there's the
start or end
//                                          of a beat detected.
//
//////////////////////////////////////////////////////////////////////

module beat_markers(   clk,reset,right_x,right_y,
                                                    beat_start,beat_end,time_scale,  tol,  tol_end,
tol_start);

   input clk,reset;
   input [10:0] right_x;
   input [9:0]  right_y;

        input [1:0]     time_scale;
        input [1:0]  tol, tol_end, tol_start;

   output beat_start,beat_end;


        // Define bound parameters for detecting the potential end of a beat:
   parameter TOLERANCE_X = 2;
   parameter TOLERANCE_Y = 2;
        // Define bound parameters for detecting the actual end of a beat:
   parameter TOLERANCE_END_X = 5;
   parameter TOLERANCE_END_Y = 5;
        // Definte bound parameters for detecting the start of another beat:
   parameter TOLERANCE_START_X = 25;
   parameter TOLERANCE_START_Y = 25;

        parameter STATIONARY_TIME = 7*(1048575/8);


// TESTING

        // Define bound parameters for detecting the potential end of a beat:
   //wire [9:0] TOLERANCE_X = tol; //5*tol; //10'd10;
  //wire [9:0] TOLERANCE_Y = tol; //5*tol;//9'd10;
        // Define bound parameters for detecting the actual end of a beat:
   //wire [9:0] TOLERANCE_END_X = 5*tol_end; //5*tol_end;//10'd15;
   //wire [9:0] TOLERANCE_END_Y = 5*tol_end; //5*tol_end;//9'd15;
        // Definte bound parameters for detecting the start of another beat:
   //wire [9:0] TOLERANCE_START_X = 5*tol_start; //10*tol_start;//10'd25;
   //wire [9:0] TOLERANCE_START_Y = 5*tol_start; //10*tol_start;//9'd25;
```

```
        // Stores previous samples of coordinates:
    reg [10:0] x_prev;
    reg [9:0]  y_prev;

        // Stores potential stop coordinates:
    reg [10:0] temp_x;
    reg [9:0]  temp_y;

    reg [21:0] count_stationary;      // Count how many clk cycles the hand must remain
stationary to register as a beat end

    reg [1:0] state = 0;                              // State of FSM

    reg beat_start, beat_end;         // Holds value for outputs of beat start and end

        // Define bounds for detecting a potential stop motion:
    wire [10:0] upper_x = x_prev + TOLERANCE_X;
    wire [10:0] lower_x = x_prev - TOLERANCE_X;
    wire [9:0]  upper_y = y_prev + TOLERANCE_Y;
    wire [9:0]  lower_y = y_prev - TOLERANCE_Y;

        // Define bounds for detecting the end of a beat:
    wire [10:0] upper_stop_x = temp_x + TOLERANCE_END_X;
    wire [10:0] lower_stop_x = temp_x - TOLERANCE_END_X;
    wire [9:0]  upper_stop_y = temp_y + TOLERANCE_END_Y;
    wire [9:0]  lower_stop_y = temp_y - TOLERANCE_END_Y;

        // Define bounds for detecting the start of a beat:
    wire [10:0] upper_start_x = temp_x + TOLERANCE_START_X;
    wire [10:0] lower_start_x = temp_x - TOLERANCE_START_X;
    wire [9:0]  upper_start_y = temp_y + TOLERANCE_START_Y;
    wire [9:0]  lower_start_y = temp_y - TOLERANCE_START_Y;


        // Detect potential stop: Check if the current coordinates are within certain bounds of
the previous coordinates
    wire within_bounds = (right_x > lower_x) && (right_x < upper_x) && (right_y > lower_y) &&
(right_y < upper_y);

        // Detect actual stop: Check if current coordinates are within certain bounds of the
coordinates of the potential stop
    wire stationary_test = (right_x > lower_stop_x) && (right_x < upper_stop_x) && (right_y >
lower_stop_y) && (right_y < upper_stop_y);

        // Detect start: Check if current coordinates are within certain bounds of the
coordinates at the beat end
    wire start_test = (right_x > lower_start_x) && (right_x < upper_start_x) && (right_y >
lower_start_y) && (right_y < upper_start_y);



    always @ (posedge clk) begin
                if (reset) begin
                x_prev                          <= 0;
                y_prev                          <= 0;
                        temp_x                          <= 0;
                        temp_y                          <= 0;
                        count_stationary        <= 0;
                state                           <= 0;
                beat_start                      <= 0;
                beat_end                        <= 0;
            end
            else begin
                        // Store coordinates like a small memory
                x_prev <= right_x;
                y_prev <= right_y;
                if (state == 0) begin // State 00: Beat started, wait for stop
                                beat_start <= 0;
```

```
                        if (within_bounds) begin      // Potential stop detected
                state <= 1;
                                        temp_x <= right_x;
                                        temp_y <= right_y;
                end
                end
                else if (state == 1) begin // State 01: Stop detected, test for real stop
                if (stationary_test) begin    // Still potentially stopped, increase the time
count
                        count_stationary <= count_stationary + 1;
                end
                                else begin            // Wasn't stopped for long enough, return to
state 00
                        state <= 0;
                                        count_stationary <= 0;
                end

                if (count_stationary == STATIONARY_TIME) begin      // Only if stopped for a long
time, register as beat end
                        state <= 2;
                                        beat_end <= 1;
                                        count_stationary <= 0;
                                        // Update with the beat end coordinates
                                        temp_x <= right_x;
                                        temp_y <= right_y;
                end

        end
                        else if (state == 2) begin // State 10: Beat ended, wait for movement.
                                beat_end <= 0;
                if (~start_test) begin // If movement in the hand begins again, return to state 00
                        state <= 0;
                                        beat_start <= 1;      // Start beat
                end
                end
                        else begin                    // For unused state 11
                                state <= 0;
                        end
                end
        end
endmodule
```

---

```
//////////////////////////////////////////////////////////////////////
//
// centered_block: generate rectangle on screen     centered at x, y
//
//////////////////////////////////////////////////////////////////////
module centered_block(x,y,hcount,vcount,pixel);
   parameter WIDTH = 30;      // default width: 30 pixels
   parameter HEIGHT = 30;     // default height: 30 pixels
   parameter COLOR_R = 8'd0;
   parameter COLOR_G = 8'd255;
   parameter COLOR_B = 8'd0;

   input [10:0] x,hcount;
   input [9:0] y,vcount;
   output [23:0] pixel;

   reg [23:0] pixel;
   always @ (x or y or hcount or vcount) begin
      if ((hcount >= (x-(WIDTH/2)) && hcount < (x+(WIDTH/2))) &&
         (vcount >= (y-(HEIGHT/2)) && vcount < (y+(HEIGHT/2))))
       pixel = {COLOR_R,COLOR_G,COLOR_B};
      else pixel = 0;
   end
endmodule
```

---

```
//////////////////////////////////////////////////////////////////////
```

```
//
// color_decision: Takes in camera from video and outputs a signal
//                                          (pixel_video) which goes high when the desired blue
color
//                                          is detected.  Also outputs RGB video to be used in
displaying
//                                          the camera video on screen.
//
//////////////////////////////////////////////////////////////////

module color_decision( clk,reset,vr_pixel,pixel_video,cam_image,switch);

   input                              clk,reset;
   input       [15:0]  vr_pixel;
        input          [7:0]          switch;

   output                             pixel_video;
        output  [23:0]  cam_image;

        // Define minimum cb (blue chrominance) and maximum cr:
        wire [4:0]     color_cb_min   = 5'b10010;
        wire [4:0]     color_cr_max   = 5'b10000;


        // Y and Cb from camera input:
        wire [5:0]     Y                    = vr_pixel[15:10];
        wire [4:0]     Cr                   = vr_pixel[9:5];
        wire [4:0]     Cb                   = vr_pixel[4:0];

        // Module (YCrCb2RGB): Convert from YCrCb color space to the RGB color space:
        wire [7:0]     R_cam, G_cam, B_cam;   // Outputs for color converter module
        YCrCb2RGB color_convert( R_cam, G_cam, B_cam, clk, reset, {Y, 4'b0000}, {Cr, 5'b00000},
{Cb, 5'b00000} );

        // Detect the desired blue color:
        wire                   color_found    = (Cb >= color_cb_min) && (Cr <= color_cr_max);

        // Initial processing of camera input data at 65mhz.  pixel_video = 1 if blue detected
        reg pixel_video;

        //reg [7:0]    R_cam_reg, G_cam_reg, B_cam_reg;
        reg [7:0]      temp_R1, temp_G1, temp_B1;
        reg [7:0]      temp_R2, temp_G2, temp_B2;


        // Dim the display a little by dividing RGB values:
        wire [7:0] scaled_R_cam = (R_cam/2);
        wire [7:0] scaled_G_cam = (G_cam/2);
        wire [7:0] scaled_B_cam = (B_cam/2);

        // Used for display of the camera video input as RGB data:
        assign  cam_image = {scaled_R_cam, scaled_G_cam, scaled_B_cam};

        always @(posedge clk) begin
                pixel_video <= (color_found) ? 1'b1 : 1'b0;
        end

endmodule
```

---

```
//////////////////////////////////////////////////////////////////
//
// color_detection: Uses information from the camera to detect if the
//                                                  desired color is in the left or right hand
plane.
//
//////////////////////////////////////////////////////////////////

module color_detection(       reset,clk,pixel_video,hcount,vcount,
                                                        left_en,right_en,desired_color);
```

```
   input                           reset, clk;
   input                                  pixel_video;   // Camera input.  High if initially
detected as the desired color.
   input              [10:0]  hcount;
   input              [9:0]   vcount;
   output                          left_en;                    // High if desired color
detected and pixel is in left hand plane
       output                      right_en;             // High if desired color detected and
pixel is in right hand plane
       output                      desired_color;               // High if registered as the
desired color


       // Define borders of the video area on screen:
   parameter RIGHT_BORDER     = 867;
   parameter LEFT_BORDER      = 157;
   parameter TOP_BORDER       = 27;
   parameter BOTTOM_BORDER = 505;
       parameter CENTER_X            = (LEFT_BORDER + RIGHT_BORDER)/2;     // Find the middle of
the window

       // Decide if detected pixel is in left half of the screen
   wire left_side = (LEFT_BORDER < hcount) && (hcount < CENTER_X) && (TOP_BORDER < vcount) &&
(vcount < BOTTOM_BORDER);

       // Decide if detected pixel is in right half of the screen
   wire right_side = (CENTER_X < hcount) && (hcount < RIGHT_BORDER) && (TOP_BORDER < vcount) &&
(vcount < BOTTOM_BORDER);

       // Used for storing old pixel values
       reg pix1, pix2;
   // Error reduction: desired_color only asserts high if three pixels in a row are blue:
       wire desired_color_temp = pix1 && pix2;

       // Divide into left and right halves of the screen
       assign left_en = left_side && desired_color_temp;
   assign right_en = right_side && desired_color_temp;

       assign desired_color = desired_color_temp;

       // Shift registers to compare consecutive pixel samples:
       always @(posedge clk) begin
              if (reset) begin
                     pix1 <= 0;
                     pix2 <= 0;
              end
              else begin
                     pix1 <= pix2;
                     pix2 <= pixel_video;
              end
       end

endmodule
```

---

```
////////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module
//
////////////////////////////////////////////////////////////////////////////////

module debounce (reset, clk, noisy, clean);
   input reset, clk, noisy;
   output clean;

   parameter NDELAY = 650000;
   parameter NBITS = 20;

   reg [NBITS-1:0] count;
   reg xnew, clean;
```

```
   always @(posedge clk)
     if (reset) begin xnew <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != xnew) begin xnew <= noisy; count <= 0; end
     else if (count == NDELAY) clean <= xnew;
     else count <= count+1;

endmodule
```

---

```
//////////////////////////////////////////////////////////////////////
//
// find_distance: Calculates the distance in pixels between points
//                                          A and B
//
//////////////////////////////////////////////////////////////////////

module find_distance(clk,reset,Ax,Ay,Bx,By,distance);

   input                            clk,reset;
   input       [10:0]  Ax, Bx;
   input       [9:0]   Ay, By;

   output      [10:0]  distance;

        reg [10:0]     distance;

   // Differences along x and y axes:
        wire [10:0] diff_x = (Ax > Bx) ? (Ax - Bx) : (Bx - Ax);
   wire [10:0] diff_y = (Ay > By) ? (Ay - By) : (By - Ay);

        // Calculate sum of squared differences:
   wire [19:0] sum_of_squares = diff_x*diff_x + diff_y*diff_y;

        // Wires for square_root module:
        wire ce               = 1;                          // input:      ce = 1 means module is
enabled
        wire aclr      = reset;                     // input:      aclr will reset the square
root module
        wire [10:0] distance_temp;    // output:     result of square root calculation
        wire rdy;                                           // output:     rdy = 1 means new data
is available from sqrt

        // MODULE: Finds the square root of an input
        square_root sqrt1(sum_of_squares,clk,ce,aclr,distance_temp,rdy);

        always @(posedge clk) begin
                if (reset) begin
                        distance <= 0;
                end
                else if (rdy) begin    // Only update the distance calculation when square root
finishes
                        distance <= distance_temp;
                end
        end

endmodule
```

---

```
//////////////////////////////////////////////////////////////////////
//
// generate_visualization: Decides what R,G,B signals to send for final
//                                                          display on the monitor
//
//////////////////////////////////////////////////////////////////////

module generate_visualization(       clk,clock_27mhz,reset,hcount,vcount,
                                                                                button_up,
button_down,
                                                                                cam_image,
```

```
                                                                                left_x,  left_y,
right_x, right_y,
                                                                                beat_start_x,
beat_start_y,

       beat_end_x,beat_end_y,
                                                                                amp_left,
amp_right,
                                                                                beat_period,

       acceleration_left,acceleration_right,
                                                                                desired_color,
                                                                                display_out_r,
display_out_g, display_out_b);

       input                           clk, clock_27mhz, reset;
       input          [10:0]  hcount;
   input       [9:0]   vcount;
       input                           button_up, button_down;
       input          [23:0]          cam_image;

       input          [10:0]  left_x, right_x;
       input          [9:0]           left_y, right_y;
       input          [10:0]  beat_start_x, beat_end_x;
       input          [9:0]           beat_start_y, beat_end_y;
       input          [10:0]  amp_left, amp_right, beat_period, acceleration_left,
acceleration_right;


   input                           desired_color;
       output  [7:0]          display_out_r, display_out_g, display_out_b;


       reg [10:0] moving_amp_left, moving_amp_right;


       // Define borders of the video area on screen:
   parameter RIGHT_BORDER     = 867;
   parameter LEFT_BORDER      = 157;
   parameter TOP_BORDER       = 27;
   parameter BOTTOM_BORDER = 505;
       parameter CENTER_X             = (LEFT_BORDER + RIGHT_BORDER)/2;     // Find the middle of
the window

       parameter BORDER_WIDTH = 10;  // The width of the border separating camera from analyzer
visualization

       // High at pixels along the center line of the video:
       wire middle = (hcount == CENTER_X);

       // High at pixels outside the video frame:
   wire outside_frame = (hcount < LEFT_BORDER) || (hcount > RIGHT_BORDER) || (vcount < TOP_BORDER)
|| (vcount > BOTTOM_BORDER);

   wire border = ((hcount == LEFT_BORDER) || (hcount == RIGHT_BORDER) || (vcount == TOP_BORDER)
|| (vcount == BOTTOM_BORDER));
       // High at pixels on the border and center of the video:
       wire thin_border = (~outside_frame) && (border || middle);


       // Signals for the background:

       // Display border around video section:
       wire border_display = (vcount <= (BOTTOM_BORDER + BORDER_WIDTH)) && outside_frame;

       // Display bottom bar:
       wire bottom_bar  = (vcount > (BOTTOM_BORDER + BORDER_WIDTH)) && (vcount <= (BOTTOM_BORDER
+ 2*BORDER_WIDTH));

  // Registers for holding values used in display:
```

```
            reg [23:0]     pixel;                        //     The actual video displayed on
the screen
            reg [23:0]     border_pixel;          // The border of the top video section
            reg [23:0]     bottom_bar_pixel;      // The bar between the video section and the
motion analyzer visualization

            // Determines the current state of the video display:
            reg                   menu_state = 0;



       // Generate Sprites:

       wire   [23:0]  left_hand_pixel, right_hand_pixel;    // {R,G,B} for display of blocks
following left and right hands
       wire   [23:0]  beat_start_pixel, beat_end_pixel;     // {R,G,B} for display of blocks at
start and end of a beat

       // Block following left hand:
       centered_block left_hand(left_x,left_y,hcount,vcount,left_hand_pixel);
       defparam left_hand.WIDTH             = 20;
       defparam left_hand.HEIGHT            = 20;
       defparam left_hand.COLOR_R    = 8'd0;
       defparam left_hand.COLOR_G    = 8'd255;
       defparam left_hand.COLOR_B    = 8'd0;

       // Block following right hand:
       centered_block right_hand(right_x,right_y,hcount,vcount,right_hand_pixel);
       defparam right_hand.WIDTH            = 20;
       defparam right_hand.HEIGHT    = 20;
       defparam right_hand.COLOR_R   = 8'd255;
       defparam right_hand.COLOR_G   = 8'd0;
       defparam right_hand.COLOR_B   = 8'd0;




       // Block to display at the start of a beat:
      centered_block  beat_start_block(beat_start_x,beat_start_y,hcount,vcount,beat_start_pixel);
       defparam beat_start_block.WIDTH              = 10;
       defparam beat_start_block.HEIGHT     = 10;
       defparam beat_start_block.COLOR_R    = 8'd0;
       defparam beat_start_block.COLOR_G    = 8'd255;
       defparam beat_start_block.COLOR_B    = 8'd0;

       // Block to display at the end of a beat:
       centered_block beat_end_block(beat_end_x,beat_end_y,hcount,vcount,beat_end_pixel);
       defparam beat_end_block.WIDTH        = 10;
       defparam beat_end_block.HEIGHT       = 10;
       defparam beat_end_block.COLOR_R      = 8'd255;
       defparam beat_end_block.COLOR_G      = 8'd0;
       defparam beat_end_block.COLOR_B      = 8'd0;

       // Display motion analyzer results:
       wire [23:0] amp_left_bar;
       wire [10:0] amp_left_x = 11'd0;
       wire [9:0]     amp_left_y = 10'd550;

       analyzer_bar amp_left_block(amp_left_x,amp_left_y,amp_left,hcount,vcount,amp_left_bar);
       defparam amp_left_block.HEIGHT       = 20;
       defparam amp_left_block.COLOR_R      = 8'd0;
       defparam amp_left_block.COLOR_G      = 8'd255;
       defparam amp_left_block.COLOR_B      = 8'd0;

       wire [23:0] amp_right_bar;
       wire [10:0] amp_right_x = 11'd0;
       wire [9:0]     amp_right_y    = 10'd580;

       analyzer_bar
amp_right_block(amp_right_x,amp_right_y,amp_right,hcount,vcount,amp_right_bar);
```

```
        defparam amp_right_block.HEIGHT      = 20;
        defparam amp_right_block.COLOR_R = 8'd255;
        defparam amp_right_block.COLOR_G = 8'd0;
        defparam amp_right_block.COLOR_B = 8'd0;


        wire [23:0] beat_period_bar;
        wire [10:0] beat_period_x = 11'd0;
        wire [9:0]     beat_period_y = 10'd610;

        wire [10:0]    scaled_beat_period = 4*beat_period;

        analyzer_bar
beat_period_block(beat_period_x,beat_period_y,scaled_beat_period,hcount,vcount,beat_period_bar);
        defparam beat_period_block.HEIGHT    = 20;
        defparam beat_period_block.COLOR_R   = 8'd0;
        defparam beat_period_block.COLOR_G   = 8'd0;
        defparam beat_period_block.COLOR_B   = 8'd255;

        wire [23:0] acceleration_left_bar;
        wire [10:0] acceleration_left_x = 11'd0;
        wire [9:0]     acceleration_left_y = 10'd640;

        analyzer_bar
acceleration_left_block(acceleration_left_x,acceleration_left_y,acceleration_left,hcount,vcount,a
cceleration_left_bar);
        defparam acceleration_left_block.HEIGHT      = 20;
        defparam acceleration_left_block.COLOR_R     = 8'd0;
        defparam acceleration_left_block.COLOR_G     = 8'd255;
        defparam acceleration_left_block.COLOR_B     = 8'd0;

        wire [23:0]    acceleration_right_bar;
        wire [10:0]    acceleration_right_x = 11'd0;
        wire [9:0]     acceleration_right_y = 10'd670;

        analyzer_bar
acceleration_right_block(acceleration_right_x,acceleration_right_y,acceleration_right,hcount,vcou
nt,acceleration_right_bar);
        defparam acceleration_right_block.HEIGHT     = 20;
        defparam acceleration_right_block.COLOR_R = 8'd255;
        defparam acceleration_right_block.COLOR_G = 8'd0;
        defparam acceleration_right_block.COLOR_B = 8'd0;

        // All the motion analyzer bars grouped together:
   wire [23:0] analyzer_bar_group = amp_left_bar + amp_right_bar + beat_period_bar +
acceleration_left_bar + acceleration_right_bar;




 // Generates the menu on the display:

        // Parameters for placing the sprites of the menu buttons on screen:
        parameter MODE1_X = 970;
        parameter MODE1_Y = 570;
        parameter MODE2_X = 970;
        parameter MODE2_Y = 650;

  // Generating sprites for menu:

        wire [23:0] mode1_pixel;
        wire [10:0] mode1_x = MODE1_X;
        wire [9:0]     mode1_y = MODE1_Y;

        // Block to display mode 1:
        centered_block mode1_block(mode1_x,mode1_y,hcount,vcount,mode1_pixel);
        defparam mode1_block.WIDTH    = 60;
        defparam mode1_block.HEIGHT   = 60;
        defparam mode1_block.COLOR_R  = 8'd255;
        defparam mode1_block.COLOR_G  = 8'd236;
```

```
        defparam mode1_block.COLOR_B  = 8'd139;

   wire [23:0] mode2_pixel;
        wire [10:0] mode2_x = MODE2_X;
        wire [9:0]     mode2_y = MODE2_Y;
        // Block to display mode 2:
        centered_block mode2_block(mode2_x,mode2_y,hcount,vcount,mode2_pixel);
        defparam mode2_block.WIDTH    = 60;
        defparam mode2_block.HEIGHT   = 60;
        defparam mode2_block.COLOR_R  = 8'd255;
        defparam mode2_block.COLOR_G  = 8'd236;
        defparam mode2_block.COLOR_B  = 8'd139;

   wire [23:0] current_mode_pixel;
        wire [10:0] current_mode_x = (menu_state == 0) ? MODE1_X : MODE2_X;
        wire [9:0]     current_mode_y = (menu_state == 0) ? MODE1_Y : MODE2_Y;

        // Block to display the background border around the button of the current mode:
        centered_block
current_mode_block(current_mode_x,current_mode_y,hcount,vcount,current_mode_pixel);
        defparam current_mode_block.WIDTH            = 70;
        defparam current_mode_block.HEIGHT           = 70;
        defparam current_mode_block.COLOR_R   = 8'd255;
        defparam current_mode_block.COLOR_G   = 8'd127;
        defparam current_mode_block.COLOR_B   = 8'd0;

        wire [23:0]    number1_pixel, number2a_pixel, number2b_pixel;

        parameter NUM_WIDTH = 8;
        parameter NUM_HEIGHT = 40;

        // Block to display number 1:
        centered_block number1_block(mode1_x,mode1_y,hcount,vcount,number1_pixel);
        defparam number1_block.WIDTH         = NUM_WIDTH;
        defparam number1_block.HEIGHT        = NUM_HEIGHT;
        defparam number1_block.COLOR_R       = 8'd255;
        defparam number1_block.COLOR_G       = 8'd36;
        defparam number1_block.COLOR_B       = 8'd0;

        wire [10:0] num2a_x = mode2_x - NUM_WIDTH;

        // Block to display number 2a (the first bar of the II):
        centered_block number2a_block(num2a_x,mode2_y,hcount,vcount,number2a_pixel);
        defparam number2a_block.WIDTH        = NUM_WIDTH;
        defparam number2a_block.HEIGHT       = NUM_HEIGHT;
        defparam number2a_block.COLOR_R      = 8'd255;
        defparam number2a_block.COLOR_G      = 8'd36;
        defparam number2a_block.COLOR_B      = 8'd0;

        wire [10:0] num2b_x = mode2_x + NUM_WIDTH;

        // Block to display number 2b (the second bar of the II):
        centered_block number2b_block(num2b_x,mode2_y,hcount,vcount,number2b_pixel);
        defparam number2b_block.WIDTH        = NUM_WIDTH;
        defparam number2b_block.HEIGHT       = NUM_HEIGHT;
        defparam number2b_block.COLOR_R      = 8'd255;
        defparam number2b_block.COLOR_G      = 8'd36;
        defparam number2b_block.COLOR_B      = 8'd0;


        //     Generating blocks to follow the path of the hand motion:

        wire   [23:0]  left_hand_pixel1, right_hand_pixel1;  // {R,G,B} for display of closer
trailing blocks for left and right hands
        wire   [23:0]  left_hand_pixel2, right_hand_pixel2;  // {R,G,B} for display of farther
trailing blocks for left and right hands

        // Registers to keep track of the old hand positions:
        reg [10:0]     left_x_old1, left_x_old2, right_x_old1, right_x_old2;
        reg [9:0]      left_y_old1, left_y_old2, right_y_old1, right_y_old2;
```

```
        // Trailing block 1 following left hand:
        centered_block left_hand1(left_x_old1,left_y_old1,hcount,vcount,left_hand_pixel1);
        defparam left_hand1.WIDTH            = 20;
        defparam left_hand1.HEIGHT    = 20;
        defparam left_hand1.COLOR_R   = 8'd0;
        defparam left_hand1.COLOR_G   = 8'd200;
        defparam left_hand1.COLOR_B   = 8'd0;

        // Trailing block 1 following right hand:
        centered_block right_hand1(right_x_old1,right_y_old1,hcount,vcount,right_hand_pixel1);
        defparam right_hand1.WIDTH    = 20;
        defparam right_hand1.HEIGHT   = 20;
        defparam right_hand1.COLOR_R  = 8'd200;
        defparam right_hand1.COLOR_G  = 8'd0;
        defparam right_hand1.COLOR_B  = 8'd0;

        // Trailing block 2 following left hand:
        centered_block left_hand2(left_x_old2,left_y_old2,hcount,vcount,left_hand_pixel2);
        defparam left_hand2.WIDTH            = 20;
        defparam left_hand2.HEIGHT    = 20;
        defparam left_hand2.COLOR_R   = 8'd0;
        defparam left_hand2.COLOR_G   = 8'd100;
        defparam left_hand2.COLOR_B   = 8'd0;

        // Trailing block 2 following right hand:
        centered_block right_hand2(right_x_old2,right_y_old2,hcount,vcount,right_hand_pixel2);
        defparam right_hand2.WIDTH    = 20;
        defparam right_hand2.HEIGHT   = 20;
        defparam right_hand2.COLOR_R  = 8'd100;
        defparam right_hand2.COLOR_G  = 8'd0;
        defparam right_hand2.COLOR_B  = 8'd0;

   reg [18:0]  count;  // maximum: 2^19, which is about 0.2 of a second

        always @(posedge clock_27mhz) begin
                        count <= count + 1;
                        if (count == 0) begin         // Take position samples every 0.2 seconds

                                // Shift registers to hold hand positions:

                                left_x_old2 <= left_x_old1;
                                left_y_old2 <= left_y_old1;

                                left_x_old1 <= left_x;
                                left_y_old1 <= left_y;

                                right_x_old2 <= right_x_old1;
                                right_y_old2 <= right_y_old1;

                                right_x_old1 <= right_x;
                                right_y_old1 <= right_y;

                        end
        end



        always @(posedge clk)
        begin
                        if (menu_state == 0) begin            // State 0: Display black background
in video
                                pixel <= ((desired_color && ~outside_frame) || thin_border) ?
24'b111111111111111111111111 : 24'b0; //switch[0]
                                if (~button_down) begin                       // On button down
press, go to state 1
                                        menu_state <= 1;
                                end
                        end
                        else if (menu_state == 1) begin       // State 1: Display camera input in
background of video
```

```
                            pixel <= (thin_border) ? 24'b111111111111111111111111 :
((~outside_frame) ? cam_image : 24'b0);
                            if (~button_up) begin  // On button up press, return to state 0
                                    menu_state <= 0;
                            end
                        end

                        // Generate the background of the video part of the screen:
                        border_pixel[23:16]           <= (border_display)    ? 8'd58        : 8'd0;
                        border_pixel[15:8]            <= (border_display)    ? 8'd95 : 8'd0;
                        border_pixel[7:0]             <= (border_display)    ? 8'd205 : 8'd0;
                        bottom_bar_pixel[23:16] <= (bottom_bar)              ? 8'd0  : 8'd0;
                        bottom_bar_pixel[15:8]        <= (bottom_bar)                ? 8'd0  : 8'd0;
                        bottom_bar_pixel[7:0]  <= (bottom_bar)               ? 8'd139 : 8'd0;

   end



       wire top_layer_zero = (left_hand_pixel == 0) && (right_hand_pixel == 0);
       wire second_layer_zero = (left_hand_pixel1 == 0) && (right_hand_pixel1 == 0);

       wire [23:0] border_group                      = border_pixel + bottom_bar_pixel;
       // Display of background of video section
       wire [23:0] hand_pixel_group         = (~top_layer_zero) ? (left_hand_pixel +
right_hand_pixel) : ((~second_layer_zero) ? (left_hand_pixel1 + right_hand_pixel1) :
(left_hand_pixel2 + right_hand_pixel2));     // Display of blocks following hands
       wire [23:0] beat_markers_group       = beat_start_pixel + beat_end_pixel;  // Display of
beat start and end blocks

       // Display of the video displayed on screen.
                       //      Note: to prevent odd colors in overlap, the layers for display are
(from top to bottom):
                       //                      hand_pixel_group > beat_markers_group > pixel
       wire [23:0] cam_display_vid = (hand_pixel_group == 0) ? ((beat_markers_group == 0) ?
pixel : beat_markers_group) : hand_pixel_group;

       // Display of the numbers for the mode buttons:
       wire [23:0]    numbers_group = number1_pixel + number2a_pixel + number2b_pixel;

       // Tests used for deciding how to layer the sprites in the menu display:
       wire not_button = (mode1_pixel == 0) && (mode2_pixel == 0); // The current pixel has no
unpressed button area
       wire not_number = (numbers_group == 0);
       // The current pixel has no number area

       // Display of the unpressed buttons:
       wire [23:0] button_group = mode1_pixel + mode2_pixel;

       // Display of the final menu:
                       //      Note: to prevent odd colors in overlap, the layers for display are
(from top to bottom):
                       //                      numbers_group > button_group > current_mode_pixel
       wire [23:0] menu_vid = (~not_number) ? numbers_group : ((~not_button) ? button_group :
current_mode_pixel);


       // Signals to send to the display, separated into R, G, B:
       assign display_out_r = cam_display_vid[23:16] + analyzer_bar_group[23:16] +
border_group[23:16] + menu_vid[23:16];
       assign display_out_g = cam_display_vid[15:8] + analyzer_bar_group[15:8] +
border_group[15:8] + menu_vid[15:8];
       assign display_out_b = cam_display_vid[7:0] + analyzer_bar_group[7:0] + border_group[7:0]
+ menu_vid[7:0];

endmodule
```

---

```
//////////////////////////////////////////////////////////////////
//
```

```
// motion_analyzer: Generates qualities of amplitude, beat period,
//                                                 and acceleration by interpreting the x and y
coordinates
//                                                 of the left and right hands over time.
//
//////////////////////////////////////////////////////////////////////

module motion_analyzer(       clk,reset,left_x,left_y,right_x,right_y,
                                                        beat_start,beat_end,
                                                        beat_start_x,beat_start_y,
                                                        beat_end_x,beat_end_y,
                                                        amp_left,amp_right,
                                                        beat_period,
                                                        acceleration_left,acceleration_right,
                                                        time_scale, tol, tol_end, tol_start);

   input                              clk,reset;
   input              [10:0]  left_x, right_x;
   input              [9:0]   left_y, right_y;

        input [1:0]     time_scale;
        input [1:0]  tol, tol_end, tol_start;


        output                        beat_start,beat_end;
   output      [10:0]  beat_start_x, beat_end_x;
   output      [9:0]   beat_start_y, beat_end_y;
   output      [10:0]  amp_left, amp_right, beat_period, acceleration_left, acceleration_right;

        wire beat_start_wire;                       // High for one clock cycle when beat starts,
low otherwise.
        wire beat_end_wire;                                 // High for one clock cycle when beat
ends, low otherwise.

        assign beat_start = beat_start_wire;
        assign beat_end = beat_end_wire;


        // Coordinates of where the beat started:
        reg [10:0]     beat_start_x;
        reg [9:0]      beat_start_y;

        // Coordinates of where the beat ended:
        reg [10:0]     beat_end_x;
        reg [9:0]      beat_end_y;

        // MODULE: Determines the start and end of a beat
        beat_markers beat_markers_module(clk,reset,right_x,right_y,

        beat_start_wire,beat_end_wire,

        time_scale, tol, tol_end, tol_start);


        // MODULE: Generates the amplitude, period, and acceleration qualities based on beat
markers

        wire [10:0] amp_left, amp_right, beat_period, acceleration_left, acceleration_right;
        qualities_generator qual_gen1(clk,reset,

        left_x,left_y,right_x,right_y,

        beat_start_wire,beat_end_wire,

        amp_left,amp_right,
                                                                                beat_period,

        acceleration_left,acceleration_right);
```

```
        // Update coordinates of beat start and end:
        always @(posedge clk) begin
                if (reset) begin
                        beat_start_x   <= 0;
                        beat_start_y   <= 0;
                        beat_end_x             <= 0;
                        beat_end_y             <= 0;
                end
                else begin
                        if (beat_start_wire) begin    // Only update start coordinates on the
start of a new beat
                                beat_start_x <= right_x;
                                beat_start_y <= right_y;
                        end
                        if (beat_end_wire) begin      // Only update end coordinates on the end of
a beat
                                beat_end_x <= right_x;
                                beat_end_y <= right_y;
                        end
                end
        end


endmodule
```

---

```
//
// File:   ntsc2zbt.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Example for MIT 6.111 labkit showing how to prepare NTSC data
// (from Javier's decoder) to be loaded into the ZBT RAM for video
// display.
//

//  **** MODIFICATIONS: This version uses 32 bits of each location in the ZBT memory.
//                  Each location stores information for two pixels of 16 bits each.
//                  Storage for each pixel:
//                     Highest 8 bits used for Y
//                     Next 4 bits used for Cr
//                     Loweest 4 bits used for Cb
//               Code modifications following (*) in code below
//////////////////////////////////////////////////////////////////////////
// Prepare data and address values to fill ZBT memory with NTSC data

module ntsc_to_zbt(clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data, ntsc_we, sw);

   input        clk;   // system clock
   input        vclk;  // video clock from camera
   input [2:0]          fvh;
   input        dv;
   input [29:0]         din;
   output [18:0] ntsc_addr;
   output [35:0] ntsc_data;
   output       ntsc_we;      // write enable for NTSC data
   input        sw;           // switch which determines mode (for debugging)

   parameter    COL_START = 10'd160;
   parameter    ROW_START = 10'd0;

   // here put the luminance data from the ntsc decoder into the ram
   // this is for 1024 x 768 XGA display

   reg [9:0]    col = 0;
   reg [9:0]    row = 0;
   reg [15:0]   vdata = 0;
   reg          vwe;
   reg          old_dv;
   reg          old_frame;    // frames are even / odd interlaced
```

```
   reg          even_odd;     // decode interlaced frame to this wire

   wire         frame = fvh[2];
   wire         frame_edge = frame & ~old_frame;

   always @ (posedge vclk) //LLC1 is reference
     begin
        old_dv <= dv;
        vwe <= dv && !fvh[2] & ~old_dv; // if data valid, write it
        old_frame <= frame;
        even_odd = frame_edge ? ~even_odd : even_odd;

        if (!fvh[2])
          begin
             col <= fvh[0] ? COL_START :
                    (!fvh[2] && !fvh[1] && dv && (col < 1024)) ? col + 1 : col;
             row <= fvh[1] ? ROW_START :
                    (!fvh[2] && fvh[0] && (row < 768)) ? row + 1 : row;
             vdata <= (dv && !fvh[2]) ? {din[29:24],din[19:15],din[9:5]} : vdata;   // * Store
16-bit Y Cr Cb information in form described above
          end
     end

   // synchronize with system clock

   reg [9:0] x[1:0],y[1:0];
   reg [15:0] data[1:0];
   reg       we[1:0];
   reg       eo[1:0];

   always @(posedge clk)
     begin
        {x[1],x[0]} <= {x[0],col};
        {y[1],y[0]} <= {y[0],row};
        {data[1],data[0]} <= {data[0],vdata};
        {we[1],we[0]} <= {we[0],vwe};
        {eo[1],eo[0]} <= {eo[0],even_odd};
     end

   // edge detection on write enable signal

   reg old_we;
   wire we_edge = we[1] & ~old_we;
   always @(posedge clk) old_we <= we[1];

   // shift each set of four bytes into a large register for the ZBT

   reg [31:0] mydata;
   always @(posedge clk)
     if (we_edge)
       mydata <= { mydata[15:0], data[1] };      // *Store 16 bit data per clock

   // compute address to store data in

   wire [18:0] myaddr = {y[1][8:0], eo[1], x[1][9:1]};   // *Modification to address

   // alternate (256x192) image data and address
   wire [31:0] mydata2 = {data[1],data[1],data[1],data[1]};
   wire [18:0] myaddr2 = {1'b0, y[1][8:0], eo[1], x[1][7:0]};

   // update the output address and data only when four bytes ready

   reg [18:0] ntsc_addr;
   reg [35:0] ntsc_data;
   wire       ntsc_we = sw ? we_edge : (we_edge & (x[1][0]==1'b0)); // *Write every two times

   always @(posedge clk)
     if ( ntsc_we )
       begin
          ntsc_addr <= sw ? myaddr2 : myaddr; // normal and expanded modes
          ntsc_data <= sw ? {4'b0,mydata2} : {4'b0,mydata};
```

```
       end

endmodule // ntsc_to_zbt
```

---

```
//////////////////////////////////////////////////////////////////////
//
// position_calculator: Calculate the average x and y positions of the
//                                                    hands over a frame.  Left_en high
when the current
//                                                    pixel should be included in the
weighted average
//                                                    calculation for the left hand.  The
same for right_en
//                                                    with the right hand
//
//////////////////////////////////////////////////////////////////////

module position_calculator(reset,clk,

       left_en,right_en,hcount,vcount,
                                                                    vsync, left_x, left_y,
right_x, right_y);

   input                                    reset, clk, left_en, right_en,vsync;
   input              [10:0]  hcount;
   input              [9:0]          vcount;
   output      [10:0]  left_x, right_x;
   output      [9:0]          left_y,right_y;


       // Define borders of the video area on screen:
   parameter RIGHT_BORDER     = 867;
   parameter LEFT_BORDER      = 157;
   parameter TOP_BORDER       = 27;
   parameter BOTTOM_BORDER = 505;
       parameter CENTER_X            = (LEFT_BORDER + RIGHT_BORDER)/2;     // Find the middle of
the window

   wire [10:0] left_x_1, right_x_1,left_y_1, right_y_1;


     // Decide if in left half of the screen
   wire left_in_left_side = (LEFT_BORDER < left_x_1) && (left_x_1 < CENTER_X) && (TOP_BORDER <
left_y_1) && (left_y_1 < BOTTOM_BORDER);

    // Decide if in right half of the screen
   wire right_in_right_side = (CENTER_X < right_x_1) && (right_x_1 < RIGHT_BORDER) && (TOP_BORDER
< right_y_1) && (right_y_1 < BOTTOM_BORDER);


        // Modules to calculate weighted sums:

       weighted_sum ws1(reset, clk, left_en, vsync, hcount, left_x_1,vcount);
       // Left hand: x position
   weighted_sum ws2(reset, clk, left_en, vsync, {1'b0,vcount}, left_y_1,vcount);          // Left
hand: y position
   weighted_sum ws3(reset, clk, right_en, vsync, hcount, right_x_1,vcount);
       // Right hand: x position
   weighted_sum ws4(reset, clk, right_en, vsync, {1'b0,vcount}, right_y_1,vcount); // Right hand:
y position

       // Registers to hold  the x and y positions of the hands:
       reg [10:0]     left_x  = 0;
       reg [10:0]     right_x = 0;
   reg [9:0]   left_y  = 0;
       reg [9:0]      right_y        = 0;


       // Used in distance calculation between samples:
```

```
       wire [10:0] diff_right_x = (right_x > right_x_1) ? (right_x - right_x_1) : (right_x_1 -
right_x);
      wire [10:0] diff_left_x = (left_x > left_x_1) ? (left_x - left_x_1) : (left_x_1 - left_x);
       wire [10:0] diff_right_y = (right_y > right_y_1) ? (right_y - right_y_1) : (right_y_1 -
right_y);
      wire [10:0] diff_left_y = (left_y > left_y_1) ? (left_y - left_y_1) : (left_y_1 - left_y);

       // Error correction: eliminate jumps in light positions for smoother motion
       // Detecting whether two successive x,y positions are within reasonable distance of each
other.
       parameter JUMP_THRESHOLD = 75;
       wire no_left_jump = (diff_left_x < JUMP_THRESHOLD) && (diff_left_y < JUMP_THRESHOLD);
       // = 1 if reasonable distance between successive left hand coordinates
       wire no_right_jump = (diff_right_x < JUMP_THRESHOLD) && (diff_right_y < JUMP_THRESHOLD);
       // = 1 if reasonable distance between successive right hand coordinates

       // High at pixels outside the video frame:
   wire outside_frame_left = (left_x < LEFT_BORDER) || (left_x > RIGHT_BORDER) || (left_y <
TOP_BORDER) || (left_y > BOTTOM_BORDER);
   wire outside_frame_right = (right_x < LEFT_BORDER) || (right_x > RIGHT_BORDER) || (right_y <
TOP_BORDER) || (right_y > BOTTOM_BORDER);


       // Tests before updating coordinates:
       wire update_left = left_in_left_side && (no_left_jump || outside_frame_left);
       wire update_right = right_in_right_side && (no_right_jump || outside_frame_right);

   // FSM state
       reg state = 0;

       always @(posedge clk) begin
              if (reset) begin
                     left_x  <= 0;
                     right_x <= 0;
                     left_y  <= 0;
                     right_y <= 0;
                     state          <= 0;

              end
              else begin
                     if (state == 0) begin         //     State 0: Frame ended, wait for new
frame to start
              if (vsync == 1) begin         // When vsync goes high, store results of weighted
averages
                                    state <= 1;
              if (update_left) begin  //

                                                  left_x     <= left_x_1;
       // Store the left hand x position
                     left_y     <= left_y_1[9:0];         // Store the left hand y position


              end
              if (update_right) begin                //

                                                  right_x     <= right_x_1;
       // Store the right hand x position
                     right_y     <= right_y_1[9:0];       // Store the right hand y position

              end
                              end
                     end
       else if (state == 1) begin    // State 1: Frame started, wait for frame to end
              if (vsync == 0) begin
              state <= 0;
          end
                     end


              end
       end
```

```
endmodule
```

---

```
//////////////////////////////////////////////////////////////////////
//
// qualities_generator: Uses beat start and end information to calculate
//                                                      amplitude, beat period, and
acceleration qualities
//                                                      for each hand
//
//////////////////////////////////////////////////////////////////////

module qualities_generator(clk,reset,
                                                                left_x,left_y,right_x,right_y,
                                                                beat_start,beat_end,
                                                                amp_left,amp_right,
                                                                beat_period,

        acceleration_left,acceleration_right);

   input                            clk,reset;
   input             [10:0]  left_x, right_x;
   input       [9:0]   left_y, right_y;
   input                            beat_start,beat_end;

   output      [10:0]  amp_left, amp_right, beat_period, acceleration_left, acceleration_right;



        // Holds the motion qualities for output:
   reg [10:0] amp_left, amp_right, beat_period, acceleration_left, acceleration_right;

        // Store the location of the hands at beat start and end in these registers:
   reg [10:0] left_x1, right_x1, left_x2, right_x2;
   reg [9:0]  left_y1, right_y1, left_y2, right_y2;


        // Store 3 sets of samples of coordinates for each hand for use in acceleration
calculations:
        reg [10:0] left_xt1, right_xt1, left_xt2, right_xt2, left_xt3, right_xt3;
   reg [9:0]  left_yt1, right_yt1, left_yt2, right_yt2, left_yt3, right_yt3;

        // Used for generating a slower clock
   reg [18:0]  count;  // maximum: 2^19, which is about 0.2th of a second

        // FSM state
   reg                state = 0;

        // Holds a count of time a beat is taking (each count is 2^21 clock cycles):
        reg [10:0]     time_count = 0;

        // Wires for the find_distance modules:
        wire [10:0] dist_right, dist_left;
        wire [10:0] accel_left_dist1, accel_left_dist2, accel_right_dist1, accel_right_dist2;

        wire dist_calc_clr = (reset || beat_end);    // Reset distance calc on reset button or
end of beat

        // Amplitude calculation:
        find_distance dist1(clk,dist_calc_clr,left_x1,left_y1,left_x2,left_y2,dist_left);
        find_distance dist2(clk,dist_calc_clr,right_x1,right_y1,right_x2,right_y2,dist_right);

   // Acceleration calculation
                // Left hand:
        find_distance
dist3(clk,dist_calc_clr,left_xt1,left_yt1,left_xt2,left_yt2,accel_left_dist1);
        find_distance
dist4(clk,dist_calc_clr,left_xt3,left_yt3,left_xt2,left_yt2,accel_left_dist2);

                // Right hand:
```

```
       find_distance
dist5(clk,dist_calc_clr,right_xt1,right_yt1,right_xt2,right_yt2,accel_right_dist1);
       find_distance
dist6(clk,dist_calc_clr,right_xt3,right_yt3,right_xt2,right_yt2,accel_right_dist2);

       // Take the difference to get the change of speed.  If the movement is slowing down,
output 0 acceleration
       wire [10:0] accel_left        =      (accel_left_dist1 >= accel_left_dist2) ? 11'b0 :
2*(accel_left_dist2 - accel_left_dist1);
       wire [10:0] accel_right =     (accel_right_dist1 >= accel_right_dist2) ? 11'b0 :
2*(accel_right_dist2 - accel_right_dist1);

   always @ (posedge clk) begin

         if (reset) begin

             amp_left                                    <= 0;
                    amp_right                            <= 0;
                    beat_period                          <= 0;
                    acceleration_left              <= 0;
                    acceleration_right     <= 0;

                    left_x1                <= 0;
                    right_x1               <= 0;
                    left_x2                <= 0;
                    right_x2               <= 0;
             left_y1                <= 0;
                    right_y1               <= 0;
                    left_y2                <= 0;
                    right_y2               <= 0;

             left_xt1               <= 0;
                    right_xt1      <= 0;
                    left_xt2               <= 0;
                    right_xt2      <= 0;
                    left_xt3               <= 0;
                    right_xt3      <= 0;
             left_yt1               <= 0;
                    right_yt1      <= 0;
                    left_yt2               <= 0;
                    right_yt2      <= 0;
                    left_yt3               <= 0;
                    right_yt3      <= 0;

                    count                  <= 0;

             state          <= 0;
             time_count     <= 0;

         end

         else if (state == 0) begin  // State 0: Beat ended, waiting to start    a beat
           if (beat_start) begin           // Switch to state 1 when beat ends

             state <= 1;
                    time_count <= 0;

                    count <= 0;

             // Store coordinates of start position:
                    left_x1        <= left_x;
                    left_y1        <= left_y;
                    right_x1 <= right_x;
                    right_y1 <= right_y;

             // Update with new beat qualities:
                    amp_left                                    <= dist_left;
                    amp_right                                   <= dist_right;
                    beat_period                          <= time_count;
                    acceleration_left      <= accel_left;
                    acceleration_right     <= accel_right;
```

```
                    end
          end
          else if (state == 1) begin  // State 1: Beat started, waiting to end

                     // Calculating acceleration and time count:
                count <= count + 1;
                if (count == 0) begin                     // Count only every 0.02 seconds
                          time_count <= time_count + 1;
                   if (time_count <= 6) begin      // Only keep sample info within the first
6*0.02 seconds

                                                  // Store successive coordinates for acceleration
calculations:
                                                  left_xt1       <= left_xt2;
                                left_yt1       <= left_yt2;
                                right_xt1      <= right_xt2;
                                right_yt1      <= right_yt2;

                                left_xt2       <= left_xt3;
                                left_yt2       <= left_yt3;
                                right_xt2      <= right_xt3;
                                right_yt2      <= right_yt3;

                                left_xt3       <= left_x;
                                left_yt3       <= left_y;
                                right_xt3      <= right_x;
                                right_yt3      <= right_y;
                                       end
                    end

               if (beat_end) begin  // Switch to state 0 when beat ends
                       state <= 0;
                       // Store coordinates where the beat motion ended
                       left_x2        <= left_x;
                       left_y2        <= left_y;
                       right_x2 <= right_x;
                       right_y2 <= right_y;
                       end
      end
        end
endmodule
```

---

```
//////////////////////////////////////////////////////////////////////
//
// video_processor: Uses the camera input to determine the coordinates
//                                                  of each hand.
//
//////////////////////////////////////////////////////////////////////

module video_processor(       reset,clk,pixel_video,hcount,vcount,vsync,
                                                          left_x, left_y, right_x,
right_y,desired_color);

   input                              reset, clk, vsync;
   input       [10:0]  hcount;
   input       [9:0]          vcount;
   input                      pixel_video;
        output  [10:0]  left_x, right_x;      // The weighted average X position of each hand
        output  [9:0]          left_y,right_y;       // The weighted average Y position of each
hand
        output                         desired_color;        // Used for display of the
coordinates detected by the color_detection module


        // left_en = 1 when a pixel of the desired color is in the left hand plane
        // right_en = 1 when a pixel of the desired color is in the right hand plane
   wire left_en, right_en;
```

```
        // MODULE: Detects when the desired color appears in each half of the screen
    color_detection cd1(       reset, clk, pixel_video, hcount, vcount,
                                                              left_en, right_en,desired_color);

        // MODULE: Calculates the weighted average position of each hand
    position_calculator pc1(reset,clk,left_en,right_en,hcount,vcount,vsync,
                                                                    left_x, left_y, right_x,
right_y);


endmodule
```

---

```
//////////////////////////////////////////////////////////////////////
//
// weighted_sum: Calculate the weighted sum of position every frame.
//                                          Only count values when enable is high will be
included
//                                          in the calculation.
//
//////////////////////////////////////////////////////////////////////

module weighted_sum(reset,clk,enable,vsync,count,avg,vcount);

    input             reset, clk, enable, vsync;
    input               [10:0]  count;
        input           [9:0]           vcount;

    output   [10:0]     avg;

    parameter BOTTOM_BORDER = 505;

        // Store how many pixels of the desired color were detected
        reg         [17:0]   pixel_count      = 0;
    // Keep a running sum of the x or y coordinate at the detected pixels
        reg         [27:0]   sum                      = 0;

        // FSM state
    reg                                         state       = 0;

        // Output of divide module:
    wire     [27:0]            divider_result;

        // The actual result of divide:
        assign avg = divider_result[10:0];

        // Wires for divide module:
    wire     [17:0]            remd;   // Remainder
    wire                 rfd;  // Ready for new data

    wire                 aclr = 0;     // Asynchronous clear, disabled
        wire                 ce = 1;          // Clock enable, disabled

        reg                  sclr = 0;      // Synchronous clear, enabled


     // MODULE: Used for division of the running sum with the sample count
    divider div1 (
         sum,
         pixel_count,
         divider_result,
         remd,
         clk,
         rfd,
         aclr,
         sclr,
         ce);


    always @(posedge clk) begin
```

```
        if (reset) begin
            pixel_count <= 0;
            sum <= 0;
            state <= 0;
            sclr <= 0;
        end
        else begin
            if (state == 0) begin            // State 0: Tabulate the running sum and pixel count
                if (vcount >  BOTTOM_BORDER) begin                  // Once the frame is no
longer in the video range, start calculating divide
                                               sclr <= 0;
                                        end
                                        if (vsync == 0) begin             // Once the frame is
done, go to state 1
                    state <= 1;
                end
                else if (enable) begin                                    // If pixel detected,
increase count and running sum
                    pixel_count <= pixel_count + 1;
                    sum <= sum + count;
                end
            end
            else if (state == 1) begin        // State 1: Wait until the next frame starts.

                                        if (vsync == 1) begin        // Once a new frame begins,
return to state 0
                    sclr <= 1;
                    pixel_count <= 0;
                    sum <= 0;
                    state <= 0;
                end
            end
        end
   end
endmodule
```

---

```
//
// File:   zbt_6111.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Simple ZBT driver for the MIT 6.111 labkit, which does not hide the
// pipeline delays of the ZBT from the user.  The ZBT memories have
// two cycle latencies on read and write, and also need extra-long data hold
// times around the clock positive edge to work reliably.
//

////////////////////////////////////////////////////////////////////////////
// Ike's simple ZBT RAM driver for the MIT 6.111 labkit
//
// Data for writes can be presented and clocked in immediately; the actual
// writing to RAM will happen two cycles later.
//
// Read requests are processed immediately, but the read data is not available
// until two cycles after the intial request.
//
// A clock enable signal is provided; it enables the RAM clock when high.

module zbt_6111(clk, cen, we, addr, write_data, read_data,
                ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);

   input clk;                // system clock
   input cen;                // clock enable for gating ZBT cycles
   input we;                 // write enable (active HIGH)
   input [18:0] addr;        // memory address
   input [35:0] write_data;  // data to write
   output [35:0] read_data;  // data read from memory
   output       ram_clk;     // physical line to ram clock
   output       ram_we_b;    // physical line to ram we_b
   output [18:0] ram_address; // physical line to ram address
```

```
   inout [35:0]  ram_data;    // physical line to ram data
   output       ram_cen_b;    // physical line to ram clock enable

   // clock enable (should be synchronous and one cycle high at a time)
   wire         ram_cen_b = ~cen;

   // create delayed ram_we signal: note the delay is by two cycles!
   // ie we present the data to be written two cycles after we is raised
   // this means the bus is tri-stated two cycles after we is raised.

   reg [1:0]   we_delay;

   always @(posedge clk)
     we_delay <= cen ? {we_delay[0],we} : we_delay;

   // create two-stage pipeline for write data

   reg [35:0]  write_data_old1;
   reg [35:0]  write_data_old2;
   always @(posedge clk)
     if (cen)
       {write_data_old2, write_data_old1} <= {write_data_old1, write_data};

   // wire to ZBT RAM signals

   assign      ram_we_b = ~we;
   assign      ram_clk = ~clk;     // RAM is not happy with our data hold
                                   // times if its clk edges equal FPGA's
                                   // so we clock it on the falling edges
                                   // and thus let data stabilize longer
   assign      ram_address = addr;

   assign      ram_data = we_delay[1] ? write_data_old2 : {36{1'bZ}};
   assign      read_data = ram_data;

endmodule // zbt_6111
```

---

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    22:23:15 12/04/06
// Design Name:
// Module Name:    ArticulationModulatorUnit
// Project Name:
// Target Device:
// Tool versions:
// Description:        This module performs the articulation modulation for the audio.  The
module
//                                            outputs a coefficient that is used to multiply the
audio in order to
//                  create a stronger first part of a beat and a weaker second part of a beat.
//                  sample_count should be the sample number that is currently being addressed.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module ArticulationModulatorUnit(reset, clock_27mhz, musicbeatperiod, Acc,
                                 coefficient, sample_count, division);
    input reset;
    input clock_27mhz;
    input [15:0] musicbeatperiod;
    input [10:0] Acc;
        input [15:0] sample_count;
        output [7:0] coefficient;
```

```
	output [7:0] division;                    //debug output
	reg [7:0] division;
	wire [1:0] NewAcc;

	reg [7:0] coefficient;

	//figure out what part of beat we are at.  Divide beat into 16 sections
	always @ (sample_count or musicbeatperiod) begin
		if (sample_count <= musicbeatperiod >> 4)
			division = 0;
		else if (sample_count <= (musicbeatperiod >> 4) * 2)
			division = 1;
		else if (sample_count <= (musicbeatperiod >> 4) * 3)
			division = 2;
		else if (sample_count <= (musicbeatperiod >> 4) * 4)
			division = 3;
		else if (sample_count <= (musicbeatperiod >> 4) * 5)
			division = 4;
		else if (sample_count <= (musicbeatperiod >> 4) * 6)
			division = 5;
		else if (sample_count <= (musicbeatperiod >> 4) * 7)
			division = 6;
		else if (sample_count <= (musicbeatperiod >> 4) * 8)
			division = 7;
		else if (sample_count <= (musicbeatperiod >> 4) * 9)
			division = 8;
		else if (sample_count <= (musicbeatperiod >> 4) * 10)
			division = 9;
		else if (sample_count <= (musicbeatperiod >> 4) * 11)
			division = 10;
		else if (sample_count <= (musicbeatperiod >> 4) * 12)
			division = 11;
		else if (sample_count <= (musicbeatperiod >> 4) * 13)
			division = 12;
		else if (sample_count <=(musicbeatperiod >> 4)* 14)
			division = 13;
		else if (sample_count <= (musicbeatperiod >> 4) * 15)
			division = 14;
		else
			division = 15;
	end


	//ROM to store the functions that the audio will be multiplied with to create
articulation effect
	always @ (NewAcc or sample_count) begin
		if (NewAcc ==0)
			coefficient = 255;
	//legato
		else if (NewAcc == 0)
			case (division)
				0: coefficient = 153;
				1: coefficient = 230;
				2: coefficient = 255;
				3: coefficient = 255;
				4: coefficient = 255;
				5: coefficient = 255;
				6: coefficient = 255;
				7: coefficient = 255;
				8: coefficient = 255;
				9: coefficient = 255;
				10: coefficient = 255;
				11: coefficient = 255;
				12: coefficient = 230;
				13: coefficient = 179;
				14: coefficient = 153;
				15: coefficient = 102;
			endcase
		else if (NewAcc == 1)
			case (division)
				0: coefficient = 153;
```

```
                                1: coefficient = 230;
                                2: coefficient = 255;
                                3: coefficient = 255;
                                4: coefficient = 255;
                                5: coefficient = 255;
                                6: coefficient = 255;
                                7: coefficient = 255;
                                8: coefficient = 255;
                                9: coefficient = 255;
                                10: coefficient = 217;
                                11: coefficient = 179;
                                12: coefficient = 153;
                                13: coefficient = 102;
                                14: coefficient = 51;
                                15: coefficient = 26;
                        endcase
                else if (NewAcc == 2)
                                        case (division)
                                0: coefficient = 26;
                                1: coefficient = 230;
                                2: coefficient = 255;
                                3: coefficient = 255;
                                4: coefficient = 255;
                                5: coefficient = 230;
                                6: coefficient = 179;
                                7: coefficient = 153;
                                8: coefficient = 102;
                                9: coefficient = 51;
                                10: coefficient = 26;
                                11: coefficient = 26;
                                12: coefficient = 26;
                                13: coefficient = 26;
                                14: coefficient = 26;
                                15: coefficient = 26;
                        endcase
        //staccato
                else
                        case (division)
                                0: coefficient = 26;
                                1: coefficient = 230;
                                2: coefficient = 255;
                                3: coefficient = 230;
                                4: coefficient = 153;
                                5: coefficient = 51;
                                6: coefficient = 51;
                                7: coefficient = 51;
                                8: coefficient = 26;
                                9: coefficient = 26;
                                10: coefficient = 26;
                                11: coefficient = 26;
                                12: coefficient = 26;
                                13: coefficient = 26;
                                14: coefficient = 26;
                                15: coefficient = 26;
                        endcase
         end

         assign NewAcc = Acc[1:0];
endmodule
```

---

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    21:21:20 12/03/06
// Design Name:
// Module Name:    ArticulationVolumeModulator
// Project Name:
// Target Device:
```

```
// Tool versions:
// Description:        Given left and right velocities and acceleration, the Articulation
//                                          And Volume Modulator will change the input hp and
lp audio data
//                                          so that greater velocities will result in a greater
amplitude output
//                                          and a greater acceleration will result in a
choppier playback.
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module ArticulationVolumeModulator(clock_27mhz, reset, lp_audio, hp_audio,
                                   FinalAudio, AccL, AccR, VelL, VelR, musicbeatperiod, rfd,
rfdhp, lp, hp, sample_count, volumeswitch, volume, accelerationswitch);
    input clock_27mhz;
    input reset;
    input signed [28:0] lp_audio;
    input signed [28:0] hp_audio;
    output signed [7:0] FinalAudio;
    input [1:0] AccL;
    input [1:0] AccR;
    input [6:0] VelL;
    input [6:0] VelR;
        input [15:0] musicbeatperiod;
        input  [15:0] sample_count;
        input rfd;
        input rfdhp;
        input lp;
        input hp;
        input volumeswitch;
        input accelerationswitch;
        output [4:0] volume;

        reg [4:0] volume;
        reg signed [7:0] low_out, high_out = 0;
        reg signed [7:0] low_out_articulation, high_out_articulation = 0;
        reg [24:0] TempFinalAudio;

        wire [7:0] low_coefficient;
        wire [7:0] high_coefficient;
        wire [1:0] AccLInput;
        wire [1:0] AccRInput;



        ArticulationModulatorUnit LPUnit(reset, clock_27mhz, musicbeatperiod, AccLInput,
                                  low_coefficient, sample_count, division);

        ArticulationModulatorUnit HPUnit(reset, clock_27mhz, musicbeatperiod, AccRInput,
                                  high_coefficient, sample_count, division);

    //ArticulationModulatorUnit LPUnit(reset, clock_27mhz, BeatPeriod, AccL[0],  sample_count,
low_coefficient);
        always @ (posedge clock_27mhz) begin
               if (rfd)
                               low_out <= {lp_audio[28], lp_audio[22:22-6]};
               if (rfdhp)
                                high_out <= {hp_audio[28], hp_audio[22:22-6]};

              //assign the raw audio volume to be equal to greatest of the two velocities
shifted by 1
              //if velocity exceeds maximum, assign it to be 5'b11111
              if (volumeswitch)
                     if (VelL > VelR)
                         volume <= ((VelL >> 1) > 5'b11111)? 5'b11111: (VelL >> 1);
                     else
                              volume <= ((VelR >> 1) > 5'b11111)? 5'b11111: (VelR >> 1);
```

```
           else
                       volume <= 5'b11000;
        end

        //assign TempFinalAudio = VelL* low_coefficient * low_out+ VelR* high_coefficient*
high_out;
        always @ (volumeswitch or VelL or VelR or low_coefficient or high_coefficient or
high_out or low_out) begin
              if (volumeswitch)
                      TempFinalAudio = VelL* low_coefficient * low_out+ VelR* high_coefficient*
high_out;
              else
                      TempFinalAudio = 7'b1111111 * low_coefficient * low_out+ 7'b1111111 *
high_coefficient* high_out;
       end

       assign FinalAudio = {TempFinalAudio[24], TempFinalAudio[21: 21-6]};

       //allow switch to switch on and off articulation feature
       assign AccLInput = accelerationswitch? AccL: 0;
       assign AccRInput = accelerationswitch? AccR: 0;

endmodule
```

---

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:27:16 12/03/06
// Design Name:
// Module Name:    BeatGenerator2
// Project Name:
// Target Device:
// Tool versions:
// Description:         BeatGenerator produces a 1-clock-period musicbeat signal which signifies
//                                          the divisions between beats in the audio in the
flash ROM.  Access_enable
//                                          should be a 1 clock-period enable signal and
should be on only when a new
//                                          address in ROM has been accessed.  access_reset is
used to reset the count.
//                                          (this is done after every beat)
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module BeatGenerator2(reset, clock_27mhz, beginning, offset, samplecount, access_enable,
access_reset, musicbeat, beatperiod, toggle);
input [15:0] samplecount;     //debug input
input clock_27mhz, reset, access_enable, access_reset;
input beginning;
input [15:0] offset;
input [15:0] beatperiod;
input toggle;                                 //debug input
output musicbeat;
reg [15:0] sample_count;

always @ (posedge clock_27mhz) begin
       if (reset|beginning) begin
              sample_count <= offset;
       end
       else begin
              //regular increment
              if (access_enable) begin
                      sample_count <= sample_count + 1;
              end
```

```
                //reset when needed
                else if (access_reset) begin
                        sample_count <= 0;
                end
        end
end

//musicbeat enabled only when sample_count reaches beatperiod
assign musicbeat = (sample_count == beatperiod);
endmodule
```

---

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:09:36 12/01/06
// Design Name:
// Module Name:    BeatPeriodCounter
// Project Name:
// Target Device:
// Tool versions:
// Description:         BeatPeriodCounter counts how long the previous user specified beat period
was.
//                                              BeatPeriod is different from other signals
(musicbeatperiod) in the fact that it
//                                             does not correspond to one data access from the ROM.
Instead, it is 1 data access
//                                              from rom multiplied by 100.  BeatPeriod and
musicbeatperiod/100 will give identical
//                                              representations.
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module BeatPeriodCounter(reset, clock_27mhz, enable, beat, BeatPeriod, beginning);
input reset, clock_27mhz, enable, beat;
output [10:0] BeatPeriod;
reg [10:0] BeatPeriod = 0;
reg [10:0] TempBeatPeriod = 0;
reg [7:0] count = 0;
input beginning;


always @(posedge clock_27mhz) begin
        if (reset|beginning) begin
                TempBeatPeriod <= 0;
                count <= 0;
        end
        else    begin
                if (enable) begin
                        if (count >= 199)                       begin
                                TempBeatPeriod <= (TempBeatPeriod >= 2047)? 2047: TempBeatPeriod +
1;
                                count <=0;
                        end
                        else
                                count <= count +1;
                end
                if (beat) begin
                        BeatPeriod <= TempBeatPeriod;
                        TempBeatPeriod <= 0;
                        count <=0;
                end
        end
end
```

```
endmodule
```

---

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    19:21:35 11/30/06
// Design Name:
// Module Name:    DivisionConverter
// Project Name:
// Target Device:
// Tool versions:
// Description:         DivisionConverter Converts the ratio between OriginalBeatPeriod, and
BeatPeriod
//                                          to something more useful: skip, interval, add.
Every add divisions accessed,
//                                          skip divisions will be either added, or subtracted
from the ZBT address to be
//                                          accessed next.  This helps produce the tempo
modulating effect.
//                                          If add is high, skip divisions will be added.  If
add is low, skip divisions are
//                                          subtracted.
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module DivisionConverter(reset, clock_27mhz, BeatPeriod, beat, OriginalBeatPeriod, skip, interval,
add, NewBeatPeriod, NewOriginalBeatPeriod, TempBeatPeriod, musicbeatperiod);
input reset, clock_27mhz;
input beat;
input [10:0] BeatPeriod;
output [10:0] OriginalBeatPeriod;
output [2:0] skip;
output [2:0] interval;
output add;
output [10:0] NewBeatPeriod;                          //debug output
output [10:0] NewOriginalBeatPeriod;   //debug output
output [10:0] TempBeatPeriod;                         //debug output
input [15:0] musicbeatperiod;

reg [10:0] OriginalBeatPeriod;
reg [2:0] tempDifference;
reg [3:0]state = 0;
reg [10:0] NewBeatPeriod =0;
reg [10:0] NewOriginalBeatPeriod = 0;
reg [3:0] LeftBound = 10;
reg [2:0] skip=0;
reg [2:0] interval=0;
reg [10:0] TempBeatPeriod = 0;
reg add=0;
reg [2:0] denom = 7;
reg [2:0] nom = 6;
reg [4:0] count = 0;
always @ (posedge clock_27mhz) begin
        if (reset) begin
                state <= 0;
        end
        if (beat)
                state <= 0;
        else
        case (state)
        0:      begin
                        OriginalBeatPeriod <= musicbeatperiod *10 /1024;
                        NewBeatPeriod <= BeatPeriod;
```

```
                        NewOriginalBeatPeriod <= musicbeatperiod *10 /1024;         //unit
conversion
                        state <= 1;
                    end
        1:      begin
                        if ((NewOriginalBeatPeriod <= 7) & (NewBeatPeriod <= 7)) begin
                             state <= 2;
                        end
                        else begin
                            NewBeatPeriod <= NewBeatPeriod  >> 1;
                            NewOriginalBeatPeriod <= NewOriginalBeatPeriod >> 1;
                        end
                    end
        2:          begin
                        //slower playback
                        if (BeatPeriod > OriginalBeatPeriod) begin
                            add <= 0;               //address are subtracted every
interval
                            if (BeatPeriod >> 1 >= OriginalBeatPeriod) begin
                                 interval <= 2;
                                 skip <= 1;
                            end
                            else if (NewBeatPeriod == NewOriginalBeatPeriod) begin
                                 interval <= 0;
                                 skip <= 0;
                            end
                            else begin            //ratios should be between 1/2 and 1.
keep skip at 1.
                                interval <= NewOriginalBeatPeriod[2:0];
                                skip <= 1;
                            end
                        end
                        //faster playback
                        else if (BeatPeriod < OriginalBeatPeriod) begin
                            if (NewBeatPeriod == 0 ) begin
                                interval <= 1;
                                skip <= 7;
                                add <= 1;
                            end
                            else if (NewBeatPeriod == NewOriginalBeatPeriod) begin
                                 add <= 0;
                                 interval <= 0;
                                 skip <= 0;
                            end
                            else begin
                                add <= 1;      //addresses are added every interval
                                interval <= NewBeatPeriod[2:0];
                                //interval <= 1;
                                skip <= NewOriginalBeatPeriod[2:0] -
NewBeatPeriod[2:0];
                            end
                        end
                        else  begin
                            add <= 0;               //same beat period.  insure plays at
1x
                            interval <= 0;
                            skip <= 0;
                        end
                    end
        default: state <= 0;
        endcase

        //assign skip = tempDifference;
end

endmodule
```

---

```
//DivisionCounter counts the number of division accessed.  Access_enable is high everytime a new
//address is accessed from RAM.   The parameter, DIVISONLENGTH specifies the length of each
division
```

```
//in samples.  800 samples corresponds to 800/24000 of a second per division.  Addr_mod will
display
//the correct value every interval of divisions accessed, in which it will either add or subtract
//skip divisions depending on add.  If add is high, addresses are added.  If add is low,
addresses are
//subtracted(slows down music).

module DivisionCounter(reset, clock_27mhz, access_enable, addr_mod, skip, interval, add,
divisionCount);
parameter DIVISIONLENGTH = 800;
input reset, clock_27mhz;
input access_enable, add;
input [2:0] skip;
input [2:0] interval;
output [15:0] addr_mod;
output divisionCount;            //debug output

reg signed [15:0] addr_mod = 0;
reg [2:0] divisionCount = 0;
reg [15:0] addr_count;
wire division_enable;

always @ (posedge clock_27mhz) begin
       if (reset) begin
              divisionCount <= 0;
              addr_mod <= 0;
              addr_count <= 0;
       end
       else begin
              if (access_enable)
              begin
                     addr_count <= (addr_count >= DIVISIONLENGTH-1)? 0: addr_count+1;
                     //count number of divisions accessed.
                     if (division_enable) begin
                            if (divisionCount < (interval - 1)) begin
                                   addr_mod <= 0;
                                   divisionCount <= divisionCount + 1;
                            end
                            else begin
                                   //if add is high, skip divisions are added, if low, skip
divisions are subtracted
                                   if (add)
                                          addr_mod <= (DIVISIONLENGTH) * skip;
                                  else
                                          addr_mod <= -DIVISIONLENGTH * skip;
                                   divisionCount <= 0;
                            end
                     end
                     else
                            addr_mod <= 0;
              end
       end
end


assign division_enable = (addr_count >= DIVISIONLENGTH-1);

endmodule
```

---

```
//////////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Flash ROM Interface
//
// For Labkit Revision 004
//
//
// Created: January 22, 2005
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////////////////
```

```
`define FLASHOP_IDLE  2'b00
`define FLASHOP_READ  2'b01
`define FLASHOP_WRITE 2'b10

module flash_int(reset, clock, op, address, wdata, rdata, busy, flash_data,
                 flash_address, flash_ce_b, flash_oe_b, flash_we_b,
                 flash_reset_b, flash_sts, flash_byte_b);

   parameter access_cycles = 5;
   parameter reset_assert_cycles = 1000;
   parameter reset_recovery_cycles = 30;

   input reset, clock; // Reset and clock for the flash interface
   input [1:0] op; // Flash operation select (read, write, idle)
   input [22:0] address;
   input [15:0] wdata;
   output [15:0] rdata;
   output busy;
   inout [15:0] flash_data;
   output [23:0] flash_address;
   output flash_ce_b, flash_oe_b, flash_we_b;
   output flash_reset_b, flash_byte_b;
   input  flash_sts;

   reg [1:0] lop;
   reg [15:0] rdata;
   reg busy;
   reg [15:0] flash_wdata;
   reg flash_ddata;
   reg [23:0] flash_address;
   reg flash_oe_b, flash_we_b, flash_reset_b;

   assign flash_ce_b = flash_oe_b && flash_we_b;
   assign flash_byte_b = 1; // 1 = 16-bit mode (A0 ignored)

   assign flash_data = flash_ddata ? flash_wdata : 16'hZ;

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

   initial
     flash_reset_b <= 1'b1;

   reg [9:0] state;

   always @(posedge clock)
     if (reset)
       begin
          state <= 0;
          flash_reset_b <= 0;
          flash_we_b <= 1;
          flash_oe_b <= 1;
          flash_ddata <= 0;
          busy <= 1;
       end
     else if (flash_reset_b == 0)
       if (state == reset_assert_cycles)
         begin
            flash_reset_b <= 1;
            state <= 1023-reset_recovery_cycles;
         end
       else
         state <= state+1;
     else if ((state == 0) && !busy)
       // The flash chip and this state machine are both idle. Latch the user's
       // address and write data inputs. Deassert OE and WE, and stop driving
       // the data buss ourselves. If a flash operation (read or write) is
```

```
      // requested, move to the next state.
      begin
         flash_address <= {address, 1'b0};
         flash_we_b <= 1;
         flash_oe_b <= 1;
         flash_ddata <= 0;
         flash_wdata <= wdata;
         lop <= op;
         if (op != `FLASHOP_IDLE)
           begin
              busy <= 1;
              state <= state+1;
           end
         else
           busy <= 0;
      end
    else if ((state==0) && flash_sts)
      busy <= 0;
    else if (state == 1)
      // The first stage of a flash operation. The address bus is already set,
      // so, if this is a read, we assert OE. For a write, we start driving
      // the user's data onto the flash databus (the value was latched in the
      // previous state.
      begin
         if (lop == `FLASHOP_WRITE)
           flash_ddata <= 1;
         else if (lop == `FLASHOP_READ)
           flash_oe_b <= 0;
         state <= state+1;
      end
    else if (state == 2)
      // The second stage of a flash operation. Nothing to do for a read. For
      // a write, we assert WE.
      begin
         if (lop == `FLASHOP_WRITE)
           flash_we_b <= 0;
         state <= state+1;
      end
    else if (state == access_cycles+1)
      // The third stage of a flash operation. For a read, we latch the data
      // from the flash chip. For a write, we deassert WE.
      begin
         if (lop == `FLASHOP_WRITE)
           flash_we_b <= 1;
         if (lop == `FLASHOP_READ)
           rdata <= flash_data;
         state <= 0;
      end
    else
      begin
         if (!flash_sts)
           busy <= 1;
         state <= state+1;
      end

endmodule
```

---

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    20:50:04 12/04/06
// Design Name:
// Module Name:    MetronomeProgrammer
// Project Name:
// Target Device:
// Tool versions:
// Description:         Allows for the reprogram of the beat period in the music that is loaded
in ROM.
```

```
//                                              The module acts as a mini ROM, storing values for
musicbeatperiod, and offset.
//                                              Offset specifies the initial beatperiod count,
while musicbeatperiod specifies the
//                                              regular beat period.  Program must be high to
reprogram these signals.
//                                              If program_select is high, musicbeatperiod is
reprogrammed, if not, offset is reprogrammed.
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module MetronomeProgrammer(clock_27mhz, reset, program_select, program, value, music_beat_period,
                           offset);
    input clock_27mhz;
    input reset;
    input program_select;
        input program;
    input [7:0] value;
    output [15:0] music_beat_period;
    output [15:0] offset;

        reg [15:0] music_beat_period = 21900;
        reg [15:0] offset = 5000;

        always @ (posedge clock_27mhz) begin
                if (reset) begin
                        music_beat_period <= 21900;
                        offset <= 5000;
                end
                else begin
                        //does not change value unless program is selected
                        if (program)
                                //selects either offset or value to program
                                if (program_select)
                                        offset<= value*100;
                                else
                                     music_beat_period <= value * 100;
                end
        end

endmodule
```

---

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    17:12:19 12/09/06
// Design Name:
// Module Name:    SignalTamer
// Project Name:
// Target Device:
// Tool versions:
// Description:        SignalTamer takes the signals from the Video Analysis section of the
project
//                                          and converts them into signals that are easier to
use in the audio side.
//                                      Signals fed in are volume_in, and acceleration_in.
Signals out are volume_in
//                                          and acceleration_out, which will be "tamer" than
the raw inputs.  The volume will be
//                                          smoothed.  On a beat transition edge, the first
half of the
//                                          beat will be dedicated to smoothing the transition
using linear interpolation.
```

```
//                                          The beat will be divided into 32 sections, and the
first 16 will be used to smooth.
//                                      Prior to smoothing, choppy playback had resulted.
//                                          The acceleration data will be averaged with the
previous result.  Both signals are
//                                          scaled to be easier to use on the audio side.
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//performs a rudimentary low pass filtering, and sets a minimum level for the volume


module SignalTamer (reset, clock_27mhz, volume_in, volume_out, acceleration_in, acceleration_out,
                                            beat, sample_count, musicbeatperiod,
interpolationswitch);

input reset, clock_27mhz, beat;
input [10:0] volume_in;
input [10:0] acceleration_in;
input [15:0] sample_count;
output [1:0] acceleration_out;
output [6:0] volume_out;
input [15:0] musicbeatperiod;
input interpolationswitch;

wire [6:0] volume_out;
reg [6:0] volume_out_noninterpolated;
reg [6:0] volume_out_interpolated;
reg [6:0] temp_volume;
reg [6:0] current_volume;
reg [6:0] volume_out_old;
reg [1:0] acceleration_out;
reg [1:0] temp_acceleration;
reg [1:0] current_acceleration;
//reg [6:0] temp_volume2;
wire [8:0] sum;
reg [4:0] division;
always @ (posedge clock_27mhz) begin
       if (beat) begin

              //volume stuff

              //temp_volume2 <= temp_volume;
              //temp_volume <= current_volume;
              current_volume <= volume_in  >>4;


              //averages volume_out and gives it a minimum value so audio is always audible
              if (sum > 7'b1111111)
                     volume_out_noninterpolated <= 7'b1111111;
              else if (sum < 7'b0010000)
                     volume_out_noninterpolated <= 7'b0000000;
              else
                     volume_out_noninterpolated <= sum[6:0] ;

              volume_out_old <= volume_out_noninterpolated;

              //acceleration/articulation stuff

              if (acceleration_in > 100)
                     current_acceleration <= 3;
              else if (acceleration_in > 75)
                     current_acceleration <= 2;
              else if (acceleration_in > 50)
                     current_acceleration <= 1;
              else
                     current_acceleration <= 0;
```

```
                temp_acceleration <= current_acceleration;

                //averages volume_out and gives it a minimum value so audio is always audible
                acceleration_out <= ((temp_acceleration + current_acceleration) >>1);

        end
        //interpolates volumes so it sounds better in beat transitions
        //volume_out_interpolated
        volume_out_interpolated <= (volume_out_noninterpolated*division + volume_out_old * (16-
division))>>4;
end

//interpolates volumes so it sounds better in beat transitions.  divides beat into 32's and
//changes volumes ever 1/32 of a beat.  End transition at middle of next beat.
always @ (sample_count or musicbeatperiod) begin
                if (sample_count <= musicbeatperiod >> 5)
                                division = 0;
                        else if (sample_count <= (musicbeatperiod >> 5) * 2)
                                division = 1;
                        else if (sample_count <= (musicbeatperiod >> 5) * 3)
                                division = 2;
                        else if (sample_count <= (musicbeatperiod >> 5) * 4)
                                division = 3;
                        else if (sample_count <= (musicbeatperiod >> 5) * 5)
                                division = 4;
                        else if (sample_count <= (musicbeatperiod >> 5) * 6)
                                division = 5;
                        else if (sample_count <= (musicbeatperiod >> 5) * 7)
                                division = 6;
                        else if (sample_count <= (musicbeatperiod >> 5) * 8)
                                division = 7;
                        else if (sample_count <= (musicbeatperiod >> 5) * 9)
                                division = 8;
                        else if (sample_count <= (musicbeatperiod >> 5) * 10)
                                division = 9;
                        else if (sample_count <= (musicbeatperiod >> 5) * 11)
                                division = 10;
                        else if (sample_count <= (musicbeatperiod >> 5) * 12)
                                division = 11;
                        else if (sample_count <= (musicbeatperiod >> 5) * 13)
                                division = 12;
                        else if (sample_count <=(musicbeatperiod >> 5)* 14)
                                division = 13;
                        else if (sample_count <= (musicbeatperiod >> 5) * 15)
                                division = 14;
                        else
                                division = 15;
end

//scaling
assign sum = (current_volume) * 5;
//can potentially turn off interpolation
assign volume_out = interpolationswitch? volume_out_interpolated: volume_out_noninterpolated;

endmodule
```