Musical Sculpting: An interactive filtering project.

Abstract:

The following project contains the details of our attempt to implement a smart audio tool that ads the ability to interact with music provided by a CD player. With simple hand movements, the system allows a user to modify music in real-time, changing the intensity, tempo, pitch, and timbre. A digital camera detects the change in movement which will then be used to select and modify several digital filters used on the musical input. The filtered music will then be outputted through a set of speakers, and a visual representation of the filter and the filtered music in the frequency domain can be observed on a monitor.

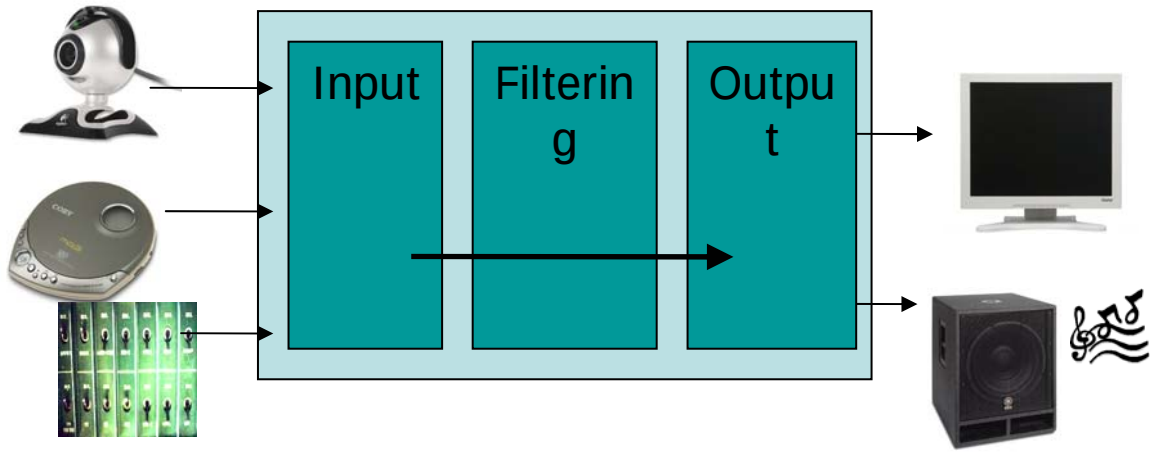
A 6.111 Final Project by:
Chen Li, Clare Davis & Austyn Hill
Spring '04, 2005

1.1 Project Overview:

1.1.1 Basic Synopsis:
1.1.2

The basic idea behind our project is the goal to create an interesting tool for a user to manipulate and filter real-time music in a hands-on and comprehensible way. Most music filtering and mixing tools that these days seem to be software constructions that while comprehensive and very powerful, end up being very complicated – and sometimes, especially when one is new to the field, it is hard to tell what filtering one way, or amplifying in another way, will actually do to the music. Therefore, as is the case with most artistic creations, we felt it important to have more hands-on interactive option. With this tool, the user can not only hear the effect of their changes in real-time to music that they input, but also see the modified music and filter projected on a video screen – really allowing them to really get a better grasp of how their particular modifications are actually impacting and changing the music.

1.1.3 Basic Block Diagram:



As is evident in the basic block diagram above, the whole system structure is divided into three main parts: Inputs, Filtering, and Outputs. These pieces are all modular, and were created separately from the rest and knitted together towards the end of the project.

The Input block takes in data from the cd-player, as well as the video coming from the digital camera. After sampling, the Input data outputs the musical information to the FFT block, where it is stored and used to create the filtered music. The Input block also handles the interpretation of the captured video, which after sampling it sends it to the internal 'Video Analyzation' block. This block then interprets the data, and outputs a vertical and horizontal pixel position to the rest of the system.

The Filtering block is obviously the filtering block of the whole apparatus. This particular modulus block takes in the current pixel position of the user's LED pen, as well as the sampled musical data from the cd player, all signals that are provided from the Input block. In turn, it first stores the sampled musical data in an SRAM. Then, using the pixel position input, it computes the horizontal movement change since it's last sample,

and uses this movement to compute a new filter. This filter is then applied to the stored music by a continuous FFT operation, and outputted as filtered data. Also, the filter itself is outputted, so that the Output block, which is handling the video visualization, can interpret the data and project it to the screen.

The Output block takes the inputs from the Filtering block and interprets them in a way that allows them to be easily visualized upon the screen, as well as actually handling the filtered music and outputting it in a way that allows it to be listened to by the user. Sampling the data at a slower rate, to help eliminate noise, the Output stores the values in its own SRAM. Then, using a generated DCM pixel clock, the block outputs the correct synching controls to the video encoder, as well as the current pixel values. The musical output is routed back to the Input block, where the sampling module simply outputs the copies the data to both PCM slots as it samples the current inputted data from the cd-player.

2.1 Video Input, & Musical Input/Output: Overview

The musical sculpture system requires the input of two different sampled media. The first media is the sampled music incoming from the cd player, playing the musical piece selected by the user. The second input is sampled from a digital camera, containing a picture of the user's movement with a led light pen. This data is analyzed by a block that outputs a pixel location to the FFT filter block. After sampling, the musical data is outputted to the FFT block, where it is stored and then modified by a filter created by change in position of the user's LED pen.

2.1.1 Musical Input/Output

The whole of the audio data, both input and output, is handled by the Audio modulus within the Input block of the system. The Audio module, interacts with the LM4550 chipset, and ac97codec, allowing the sampling of an analog signal, the output of a digital signal, the change in gain for all of all of these signals, as well as the volume of the sampled/outputted channel.

2.1.1 Understanding the Audio Codec

The largest part of the work in this particular module's construction was the comprehension of how to correctly interact with the ac97 audio codec.

The codec works on a serial input/output (named ac97_sdata_in/ac97_sdata_out respectively in the labkit) to and from each of the data ports. This data is divided into frames. Each frame is constructed of 256 ticks of the ac97 bit clock, which is generated by the chip. Each frame is divided into 13 slots, the first slot being the 'tag', or slot 0, for the rest of the frame, and indicates to the chipset which bits of the remaining slots will contain valid data. To indicate the beginning of a frame, the user has to generate a synch signal – a signal that has to go high for all 16 bits of the tag slot, and go low for the rest of the frame. Any synch signal that goes high again before the frame ends will be ignored by the chip.

Each slot pertains to a different part of the whole sampling and outputting data cycle. The first two slots are where the user can change registered values in the codec. The first slot after the tag, pertains to the command address, or address of the register that the user wants to write to/read from. These registers can control the sampling rate of the ADC, the output rate of the DAC, which channels are ignored during sampling or output, and which are amplified or muted. All of these registers and their values can be found on page 16 of the LM4550 datasheet, along with their default values. These values are referred to as the command data, and are sent to the chip in the first 18 bits of the second slot after the tag.

The actual data sampled data is provided from the chipset on the ac27_sdata_in line, during the 3rd and 4th slots. This is the data that will be outputted to the rest of the system for use in filtering and video display.

2.1.1.2 Sampling with the Audio Codec

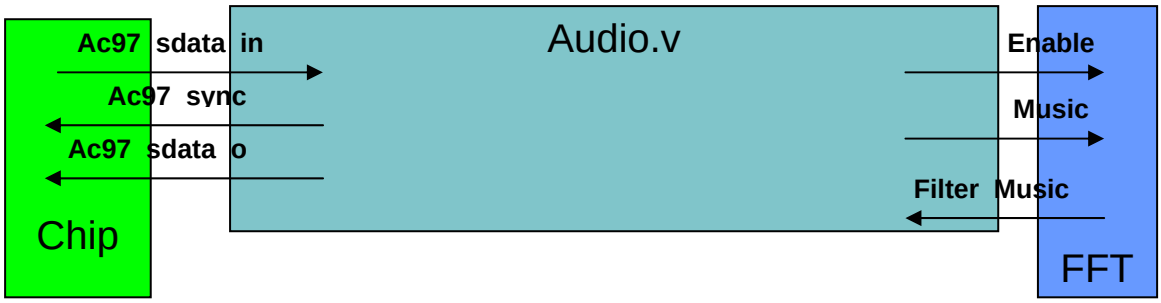


Fig. 2.1

To sample with the Audio codec, I modified code provided in the our 6.111 labkit test URL (http://www-mtl.mit.edu/Courses/6.111/labkit/verilog/004/avtest/doc/avtest.shtml) that was written by Nathan Ickes. I modified the code the drop the right recorded signal, since the FFT was only built to handle a mono value, as well as created an enable signal that pulsed high 2 clock cycles after the music data had been successfully latched, allowing the FFT block to know that the data was valid and ready to be examined and stored.

2.1.1.3 Outputting with the Audio Codec

To output the Audio codec, I simply changed the above audio.v code to use the sampled the 16bit data provided to me by the FFT module and place it into the left_out_data which was written to the PCM slot every synched frame. The music data is latched on every enable of the output. This latching is arbitrary due to the continuous output of the FFT modulus. These 16 bits are obviously smaller than the 18 bits needed by the DAC, therefore they were simply padded with zeros at the end, ensuring that the 16 bits remained the most significant to be read correctly. Otherwise the data was left. Also, future modifications that did not complete during this simplified run – included modification to the sampling time – up-sampling and down-sampling as indicated by the user. As the output would be constant, this would then slow down, or speed up the music respectively.

2.1.2.1 Modifying the Amplitude

Also, along with doing the musical sampling of the system, the audio block is also what handles the output of the filtered music. This is also the block which handles the 'amplification' of the system. In our primitive system, the change horizontal change was bound to the filter modification, and the vertical change was bound to the amplification. Therefore, with an upward movement, the user increases the amplification, which here manifests simply as the volume of the music. To implement this, a simple modulus takes the inputted vertical location, and calculates whether the current location is greater than or less than the last value sampled. If it's greater, it sends out an increase_volume pulse, otherwise it sends out a decrease_volume pulse. These pulses are taken in by the vol_up and vol_down modules written into the avtest.v code, again written by Nathan Ickes.

2.2.1 Video Input

The video input was to be provided by a light pen in front of a camera. Due to the fact that the Video codec transmits the pixel data in the form of luminence and chromiance, it seemed best to pick a bright light so that the color could be completely ignored. The pen light was supposed to be used on a darker background, allowing the great contrast in lumience to easily give away the light's position. However, sadly due to time constraints and some hardware problems – this particular module never came to fruition. Instead, a fake module for the position was used, relying on user input of pressed buttons.

2.2.2 Sampling the Video

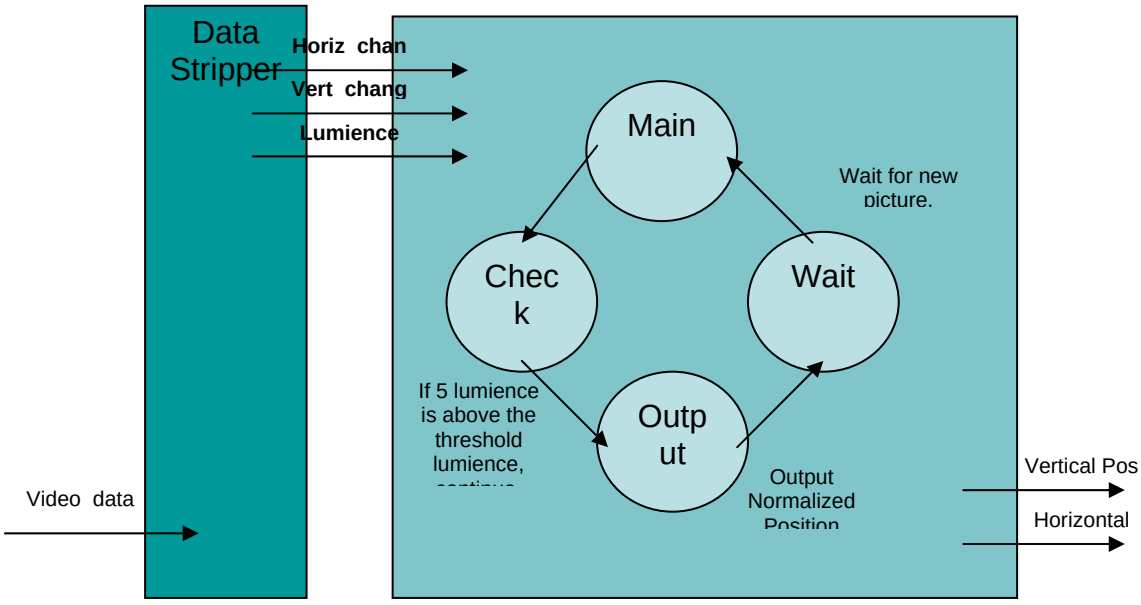


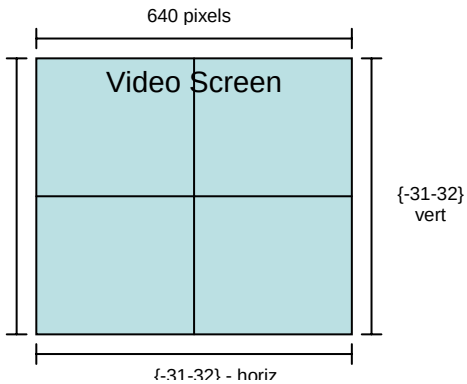
Fig. 2.2.1

Since the incoming decoded data stream used a series of horizontal and vertical synchs to indicate the pixel position, the plan was to use a modulus to strip the data from

the pixels as well as the synchs, and simply send along the lumience value of the pixel, as well as a pulse for each time that the horizontal synch and the vertical synch changed. Thus, the analyzer would simply have to use a simple counter for the two synchs, and a test before outputting a vertical and horizontal position as indicated by the counters.

2.2.3 Analyzing the Video Samples

The plan to analyze the video data was relatively simple. Using the synch pulses inputted from previous module, the analyzer was to create a counter that would check the current lumience value against a threshold value of lumience. If the current value checked out against it, and so did the next 5 horizontal values (to ensure that the value was not simply noise), then the mid point between those five would be the position values outputted. However, to account for some noise, this value was to be outputted to the FFT as only a 6 bit value, both the vertical and horizontal going from -31 to 32 despite the actual camera's resolution of 640 by 320. Thus the pixel detection could waver a bit, but still transmit a relatively correct value.



2.3 Debugging

Debugging, in most cases – consisted of trial and error results with the programmed labkit. For the audio inputs and outputs, I made some repeat inputs and continued to modify the code until I finally had something that was outputting sound on the headphones. Otherwise, I knew that there was a bug and that it did not work. As far as the video – because of the problems, there was very little debugging. Only a small module for the actual analyzation after the stripper, exists.

Sadly, however, while seemingly correct on their own, intertwining both eh FFT and the audio did not work as planned. Attempts at debugging produced mostly noise, and we are under the impression at the current moment that it's mostly due to a sampling rate that creates a problem with the audio storage, thus destroying all further attempts at filtering.

3.1 Audio Input Modification in Frequency Space : Overview

The user can select four different filters through flipping two switches. Once a filter is selected, the filter cutoff frequency changes with the change in the position of the video input. For example, if the user selects a low pass filter, he can increases the coordinate in x direction and thus incorporating higher and higher frequencies of the music. If the music is sung by a female voice, the voice will sounds like a male's at the beginning, and will increasingly sounds like the original female voice at the end. Similarly, selecting a high pass filter will make a male's voice sounds like a child's or a female's voice. By decreasing the coordinate of x, the voice will increasingly sounds like a male's. The all pass filter does not alter the frequency of the music.

The band pass filter can help detect pitch and harmonics. The user can begin by shifting the coordinate of x, which shifts the position of the band pass filter with some width in the frequency space. Once the user thinks that he detects a pitch or a harmonics of an instrument, he can resolve the pitch or the harmonics better by altering the width of the band pass filter by switching a “y enable” signal on. Then he can increase the width by increasing the coordinate of y, and decrease the width by decreasing the width of y. The decreasing in y is “finer” than increasing in y, because the filter width changes are bigger with an increment in y than a decrement in y. In this way, the user can search for a pitch of a human voice or a harmonic of an instrument, course-tune and then fine-tune to hear it.

3.1.1 Matlab

Matlab simulations are used to determine the parameters of the filter. The audio sampling rate was found to be optimized around 46kHz. The FFT transformation length N is found to produce quality music with a sampling width as low as 1000. Based on the simulation, the audio input is sampled and written to a RAM at a rate of 48 kHz (every 563 clock cycle). N is chosen to be 4095 to provide the best resolution. N sets the filter range and the transformation length of the FFT module.

3.1.2 External Inputs

External inputs includes Button_enter and the 2 most significant bits of switch (8-bit). Pushing button_enter resets the filter system. The audio input, data_audio_in, and video inputs, x and y, are sequentially passed into the system.

3.1.3 Synchronizer

Input: Button_enter, switch
Output: reset, syn_switch (2-bit)

Synchronizer registers and synchronizes the external inputs Button_enter and switch. Clock_27mhz and reset (synchronized Button_enter) are global inputs. The two most significant bit of the switch is registered in syn_switch, which selects filter as shown in the table below.

All Pass	Low Pass	High Pass	Band Pass
00	01	10	11

3.2 Audio Input

3.2.1 Control_rom_audio

Inputs: dina_audio_in, enable
Outputs: wea_audio, ena_audio, enb_audio, dina_audio, addra_audio, addrb_audio

The audio input is stored in a 2-port RAM. The writing port writes the audio input, dina_audio_in, whenever enable signal is high. The writing stage occurs at every 563 clock cycle. It writes incrementally, begin with address 0 (addra_audio), and takes 3 clock cycles to complete an address. The reading port reads 4095 addresses (addrb_audio), and outputs the oldest to the newest data (dina_audio). When an address is being written to, the reading stage halts until the completion of the writing stage. Afterwards it continues reading. During this time, 7-8 writing stages may occur. 4000 oldest audio data will be read, which provides enough samples to the FFT.

3.2.2 Rom_audio

Inputs: wea_audio, ena_audio, enb_audio, dina_audio, addra_audio, addrb_audio
Outputs: doutb_audio

This module is a 2-port memory element generated by Coregen. Port A writes the data bits dina_audio to RAM through the address port addra_audio. Port B reads the data bits doutb_audio from RAM through the address port addrb_audio. The writing and reading stages are set by wea_audio, ena_audio, enb_audio as in Lab 3. The two stages cannot occur at the same time. The writing stage and reading stages are controlled by Control_rom_audio. Rom_audio outputs doutb_audio to FFT_forward module.

3.3 Filter Storage

3.3.1 Camera_enable

This module generates the signal xy_enable. The signal pulses 1 at every 20000 clock cycle. The period of the pulse is chosen long enough such that the user, who controls x and y video input, would not be able to change the filter coefficients too fast.

3.3.2 Buffer

Input: x, y
Ouput: delta_x, delta_y, switch, xy_enable

This module detects changes to the video inputs, x and y. X labels the horizontal axis of the video input, and y labels the vertical axis of the input. The video input is divided into a 64 x 32 grid, centered on (0,0). X ranges from -32 to 32, and y ranges from -16 to 16. Their values are expressed in twos-complement, and registered by the Buffer through x_earlier and y_earlier. Whenever xy_enable is 1, x_ealier and y_earlier are updated to x_later and y_later, while x and y are registered in x_later and y_later. The buffer calculates the differences between the earlier values and later values as in Lab 3, and stores the change in delta_x and delta_y as shown in the table below.

No Change	Positive Change	Negative Change
00/11	01	10

3.3.3 Read_rom

Input: wea, edone
Output: enb, addrb

This module inputs wea and edone. It controls the reading stage of the RAM, Rom_filter. It may read the Rom_filter when wea is 0, that is, when Write_rom_filter_final is not writing. Reading begins when FFT_forward finishes one cycle of fast fourier transform by pulsing edone. Reading increments from address 0 to address 4095, and repeats until wea is set to 1.

3.3.4 Write_rom l

Input: delta_x, xy_enable, switch
Output: wea, ena, dina, addra

This module inputs delta_x, switch, and xy_enable. It stores and outputs the cutoff frequency addra of the filter coefficients. The filter is selected by syn_switch. When reset becomes 0, the module initiates the cutoff frequency of the filter coefficients by writing to all addresses either 4'b1111 or 0. The variable, write_count, increments from addresses 0 to 4095, and sets the cutoff frequency addra as specified in the table. At every xy_enable, the module changes addra by +128 or -128. The counting is kept track of by cam_count. When cam_count counts to 128, addra is set to the new cutoff frequency. If addra is near the maximum and minimum value 0 or 4095, further decrements or increments set it to 0 or 4095 respectively. Wea and ena are set to 0 until xy_enable pulses 1.

Name	# of Bits	Description
reset		= 0; Initiates addra by setting write_ena = 1 for 4095 clock cycles
xy_enable		= 1; Changes cutoff frequency by setting wea, ena = 1
syn_switch		Filter selection: = 01; Low pass = 10; High pass = 00; All pass
delta_x		= 01; Increasing cutoff frequency = 10; Decreasing cutoff frequency = 00; No change
addra	12	Cut off frequency
write_count	13	Low pass: < 512; Initiate cutoff frequency to 512, dina = 4'b1111 > 512; dina = 0 High pass:

		< 512; Initiate cutoff frequency to 512, dina = 0 > 512; dina = 4'b1111 All pass: = 4095; Initiate all pass filter cutoff frequency, dina = 4'b1111
write_ena	1	= 1; Initiates addra by writing to all addresses
cam_count	8	< 128; Changes addra by +128 or -128
wea	1	= 1; Write enable
ena	1	= 1; Port enable
dina	4	= 4'b1111 or 0; Filter coefficients

3.3.5 Rom_filter

Inputs: wea, ena, dina, addra, enb
Outputs: addrb, doutb

This module is a 2-port memory element generated by Coregen. Port A writes the data bits dina to RAM through address port addra. Port B reads the data bits doutb from RAM through address port addrb. Writing and reading stages are set by wea(write enable), ena (port a enable), enb (port b enable) as in Lab3 and cannot occur at the same time. The writing stage is controlled by Write_rom module, and the reading stage is controlled by Read_rom module.

3.4 Fast Four Transform

3.4.1 FFT

Relevant inputs: doutb_audio, xn_re, nfft_we, fwd_inv, fwd_inv_we, scale_sch_we, start
Relevant outputs: xk_re, xk_index, edone, done.

This module is generated by Coregen. The transformation length is set to 4096. Upon reset, the module continuously outputs the FFT of the input data. The reset signals are nfft_we, fwd_inv, fwd_inv_we, scale_sch_we, and start. They initiates the module and remain constant throughout the computation. Scale_sch_we is set to 10'b1010101011 by the top module to avoid data overflow. When done is 1, a new set of computation begins at the next clock. The address of the input, xn_index, and the address of the outputs, xk_re, increment in sync. Since the audio output do not have imaginary components, the input xn_im is set to 0 by the top module. Edone precedes done by 1 clock cycle.

The timing diagram of FFT is shown in figure below.

3.4.2 FFT_forward and FFT_inverse

Two instances of the FFT module is generated. They are used to compute forward FFT (fwd_inv = 1) and inverse FFT (fwd_inv = 0).

FFT_forward

Relevant inputs: doutb_audio, xn_re, nfft_we, fwd_inv_we, scale_sch_we, start
Relevant outputs: xk_re, xk_index, edone, done.

This module computes the forward FFT. It receives the audio input doutb_audio from Rom_audio module. It continuously outputs the FFT of the input, xk_re, to MAC module. When it finishes one cycle of FFT, it pulses edone and done1. Read_rom then starts reading from addrb 0 to addrb 4095 repeatedly as described in its module description.

FFT_inverse

Relevant inputs: inv_xn_re, inv_nfft_we, inv_fwd_inv_we, inv_scale_sch_we, inv_start
Relevant outputs: out_xk_re, out_xk_index.

This module computes the inverse FFT. It receives the filtered audio data, inv_xn_re, from MAC module and computes their inverse FFT, out_xk_re, to video output. Upon reset, its inputs, inv_xn_re, inv_nfft_we, inv_fwd_inv_we, inv_scale_sch_we, inv_start, are set by the MAC module in the same way as FFT_forward.

3.4.3 MAC

Inputs: delta_x, delta_y, xy_enable, xk_re, xk_index, done, switch
Outputs: inv_xn_re, inv_xn_im(= 0), inv_nfft_we, inv_fwd_inv_we, inv_scale_sch_we, inv_start

This module initiates FFT_inverse the same way that FFT_forward is initiated. Upon reset, it sets the values of inv_nfft_we, inv_fwd_inv_we, inv_scale_sch_we, and inv_start. Inv_xn_im was set to 0 by Rom_audio. MAC module's main purpose is to computes the filtered version of the audio input, inv_xn_re.

The module begins one cycle of computation when done is 1. It multiplies the FFT of the audio input, xk_re, with the filter coefficients. When computing low, band, and all pass filters, the module sign-extends doutb to 16-bits. Starting from address 0, it multiplies xk_re and doutb, the filter coefficients from Rom_filter.

To compute the filter coefficients of the bandpass filter, MAC modules generates the bandpass filter coefficients by storing the cutoff frequency, which ranges from 0 to 4095, in the registers a and b. The width of the bandpass filter is initiated upon reset to 64. The width ranges from 4 to 512. In this range, it can increase by steps of 28, or decrease by steps of 4. In this way, it is easier to search for pitches and harmonics using a bandpass filter. At every xy_enable signal, when y_enable is 0, MAC module shifts the position of

a and b according to the value of delta_x. When y_enable is 1, the module updates the width of a and b according to the value of delta_y.

Name	# of bits	Description
syn_switch	2	Filter selection: = 01; Low pass = 10; High pass = 00; All pass = 11; Bandpass
done	1	= 1; Register the cutoff frequencies every 4096 clock cycles band_low, band_high
xy_enable	1	= 1; Updates the cutoff frequencies a, b every 20000 clock cycles
band_low band_high	12	band_low = a band_high = b
y_enable	1	=1; Enable shifting operation =0; Expansion/Contraction operation
delta_x	2	Shifting = 01; Shift a, b by +64 (require b < 4032) = 10; Shift a, b by -64 (require a > 64) = 00; No change
delta_y	2	Expansion/Contraction = 01; Expand b by +28 (require b < b_high) = 10; Contracts b by -4 (requires b > b_low) = 00; No change
b_low b_high	12	b_low = a + 4; b_high = a + 484;
doutb	1	= 1; Write enable
a	1	= 1; Port enable
b	4	= 4'b111 or 0; Filter coefficients

3.5 Debugging

The primarily way of debugging the filter part of the circuit is through Altera simulation, Modelsim simulation, and the logic analyzer. Altera and modelsim simulation helps to verify that each individual module works. Modelsim simulation verifies that the RAM generated by Coregen works. Since Modelsim cannot simulate FFT and inverse FFT, the logic analyzer is used to verify their correct operation.

A comprehensive testing through Altera simulation verified that individual modules work. Modelsim simulation verified that modules are correctly writing and reading to the addresses of the RAM that stores filter coefficients. Beside Modelsim simulation, the logic analyzer also verified that the RAM that stored the filter coefficient is correctly written and read to. Separately, the logic analyzer verified that given impulse responses of varying periods and constant responses, at each stage of the time to frequency and frequency to time conversion, the modules FFT_forward, MAC, FFT_inverse gives the correct result. All the filters, all pass, high pass, low pass, and band pass are verified to produce the correct responses given an impulse or a constant.

The only tested samples are impulse and frequency responses. When an audio input, a music file, is tested, only clipping sounds are heard when the all pass filter is selected. This originates most likely in that the resulting audio output contains only some, but not all, of the frequencies. This may result, but unlikely, from that the least 2 significant bit of the audio input are thrown away. Switching to high pass filter results in no sounds being heard. Switching to low pass filter results in quieter clipping sound. Switching to low pass filter produce similar clipping sound as the low pass.

The only tested samples are impulse and frequency responses. When an audio input, a music file, is tested, only clipping sounds are heard when the all pass filter is selected. This originates most likely in that the resulting audio output contains only some, but not all, of the frequencies. This may result, but unlikely, from that the least 2 significant bit of the audio input are thrown away. Switching to high pass filter results in no sounds being heard. Switching to low pass filter results in quieter clipping sound. Switching to low pass filter produce similar clipping sound as the low pass.

The failure to reproduce the audio output may originate from several sources. One source is that there are many parameters in the filter modules that can change, which can greatly enhance or reduce the quality of the music. One such change is the FFT transformation size. Another change is the sampling rate of the input at 48kHz, and the reading rate of the audio input from the RAM, which result in a reduction of FFT transformation size (~4000). Another source is the conversion of the input from time to frequency domain, and then back to time domain. Since transformation length is limited and finite, the conversion is imperfect. It is observed that given a constant, an impulse of ~10 clock cycle, instead of ~1 clock cycle, is generated. Recall that FFT divides the frequency space into 4095 bins at the rate of the 27mHz clock, each bin contains some range of the frequency of the audio signal. It is most likely that, some range of the frequency of the audio input may be distorted or disregarded.

4.1 Video Output

The musical sculpture system also has displays the music on a monitor, before and after it is filtered, along with the filter. The music is displayed in the time domain and while the filter is displayed in the frequency domain. These outputs allow the user to watch how their actions are affecting the filter while they move in front of the camera. The filter switches allow the user to choose which type of filtering they would like to

perform and the record button allows the user to compare the original music to the filtered version at any point.

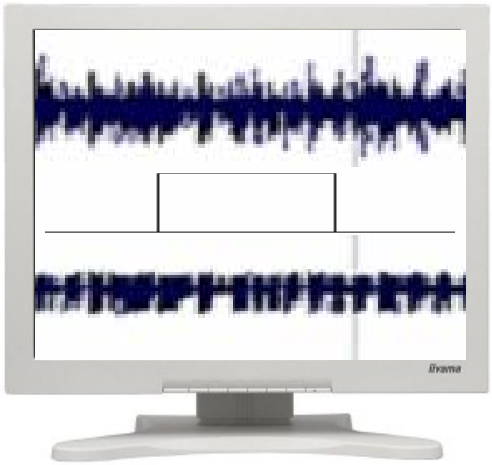
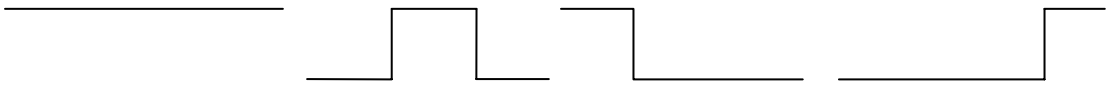


Figure 4.1 Video Display

The music waveforms on the display are updated every time the system is reset or every time the record button is pushed. Ideally it would be possible to have the music waveforms scroll across the screen. However, down-sampling the 48-khz music outputs to the point where it is meaningful to the naked eye renders waveforms that do not even approximate the originals. Therefore, the music waveforms are generated only when requested by the user. Still, the music waveforms are not precisely those generated by the convolution because of the sampling that takes place and because there are 2^16 -1 possible amplitudes for 16-bit music cannot have an individual value with 800x600 VGA output.

The filter is updated at the screen refresh rate, 60 Hz. The filter type is determined by the user-controlled switches. Unfortunately, due to different clock speeds used in different portions of the project and the type of convolution we used, the filter is simply an approximation of the actual filter based upon the change in position of the LIGHT and the boundaries of the band-pass filter. Chen implemented four different filters: the all-pass, the band-pass, the low-pass, and the high-pass. The correct type of filter is always displayed, along with the relative position of the filter. The all-pass filter is simply a straight line with value one. The low and high pass filters are be extended or shortened as the actual filter is altered. Lastly, the position of the band-pass filter changes along with the actual filter. Since we attempted to implement ideal filters and did not change the amplitude of the music, the possible filter values are zero and one.



2’b00 2’b11 2’b01 2’b10

Figure 4.2 Filters and Switches

There are two buttons and two switches that determine the VGA signals. I chose to implement a record button that captures new music for display instead of displaying a new waveform every three to five seconds. The main advantage is the record button allows the person molding the filter to capture new waveforms at whatever frequency is desired. There is also a reset button that resets the system, including the video. Lastly, the two switches determine which type of filter is being used.

4.2 Control signals and RGB

The first step in displaying video using VGA is to create vertical and horizontal sync signals. Additionally, a blanking signal and a sync signal are necessary for the ADV7125 DAC in the lab kit that supplies the RGB values for the VGA output. I chose to operate the monitor at 800x600 pixels and with a 50 Mhz pixel clock. I used a Digital Clock Manager (DCM) to generate the 50 Mhz pixel clock from the 27 Mhz labkit clock. The pixel clock is used for every other module in the video portion. Next, I split the control signals into two modules: the horizontal control signals and the vertical control signals. The first module creates the horizontal sync and sets the horizontal blanking signal low during horizontal sync and the front porch and back porch of the horizontal sync. The second module creates the vertical sync and outputs a signal indicating blanking should be low during the vertical sync and its front porch and back porch. Additionally, both the horizontal sync and the vertical sync should delayed by two clock cycles to account for the RGB delay from the ADV7125. The timing values for a 800x600 display with a 50 Mhz pixel clock are shown in Table 4.1.

	Active Video	Front Porch	Sync (active low)	Back Porch
Horizontal (Pixels)	800	56	120	64
Vertical (Lines)	600	37	6	23

Table 4.1 VGA timing specifications.

I used a finite state machine for both the horizontal and vertical sync generators. The horizontal sync generator is sensitive to the pixel clock, and simply increments a counter as it transitions through the active video, front porch, horizontal sync, and back porch states. It has a synchronous reset and generates a pulse (line_finished) each time it finishes a line. The horizontal sync generator's state diagram is shown in Figure 4.3.

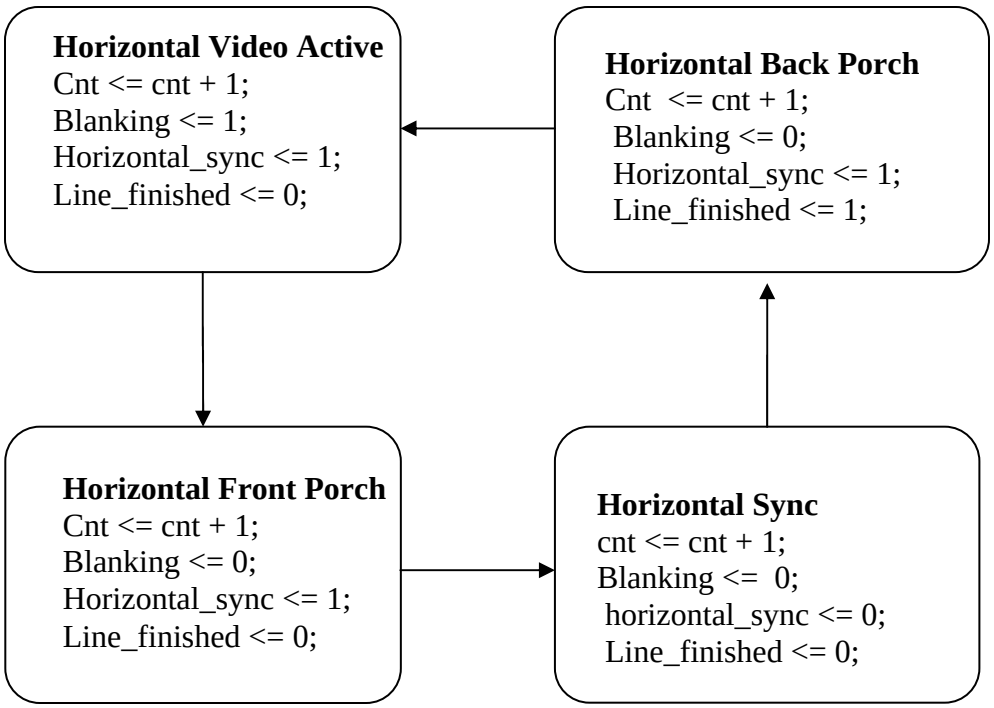


Figure 4.3 State Transition Diagram for the Horizontal Sync

Similarly, the vertical sync generator transitions through active video, front porch, vertical sync, and back porch states, but it is sensitive to reset and line_finished. The vertical counter controls which state the module is in and is also an output. The module also generates a signal (vertical_finished) when it is finished with each vertical frame. The state transition diagram is shown in Figure 4.4

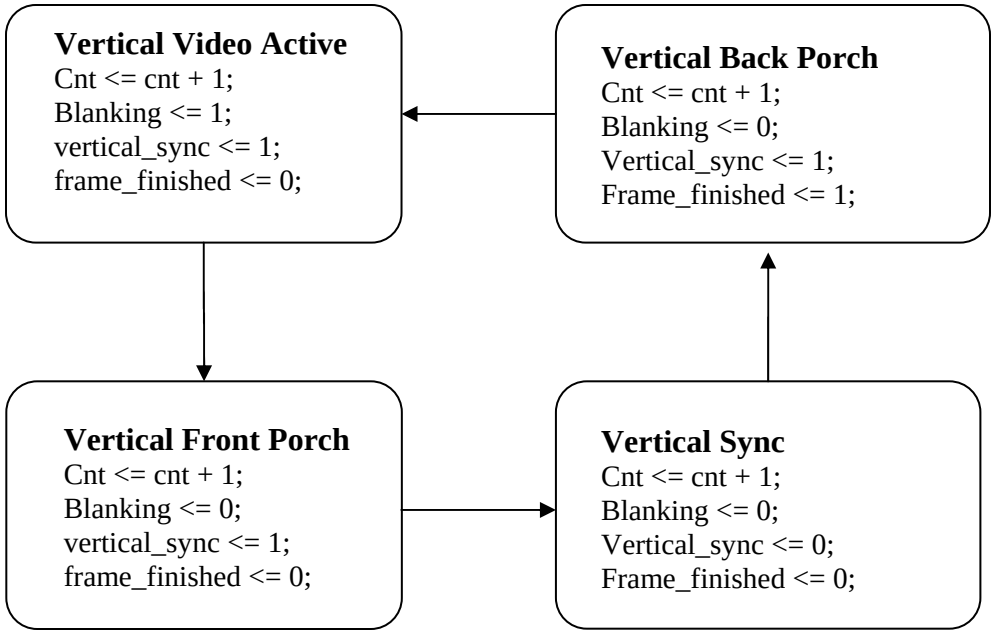


Figure 4.4 State Transition Diagram for the Vertical Sync

The ADV7125 takes three 8-bit color values (RGB) as well as the pixel clock, a blanking signal, and a sync signal. The horizontal and vertical sync signals are XOR'd together to create the sync signal sent to the ADV7125. Lastly, the horizontal and vertical blanking signals are NOR'd and that signal is the blanking signal sent to the ADV7125.

4.3 RGB Generation

The next stage was generating pixel values for the RBG signals. I accomplished this by creating a module that generates the RAM control signals and the filter. The RAM control module also generates signals that designate which part of the screen is currently being scanned. There are two RAMs and they store the music. Since they are constantly being written or read, yet another module (pick_ram) determines which ram values to pass on the pixel generator. The block diagram showing the entire video system is in Figure 4.5. The pixel clock and reset are global to everything but the DCM.

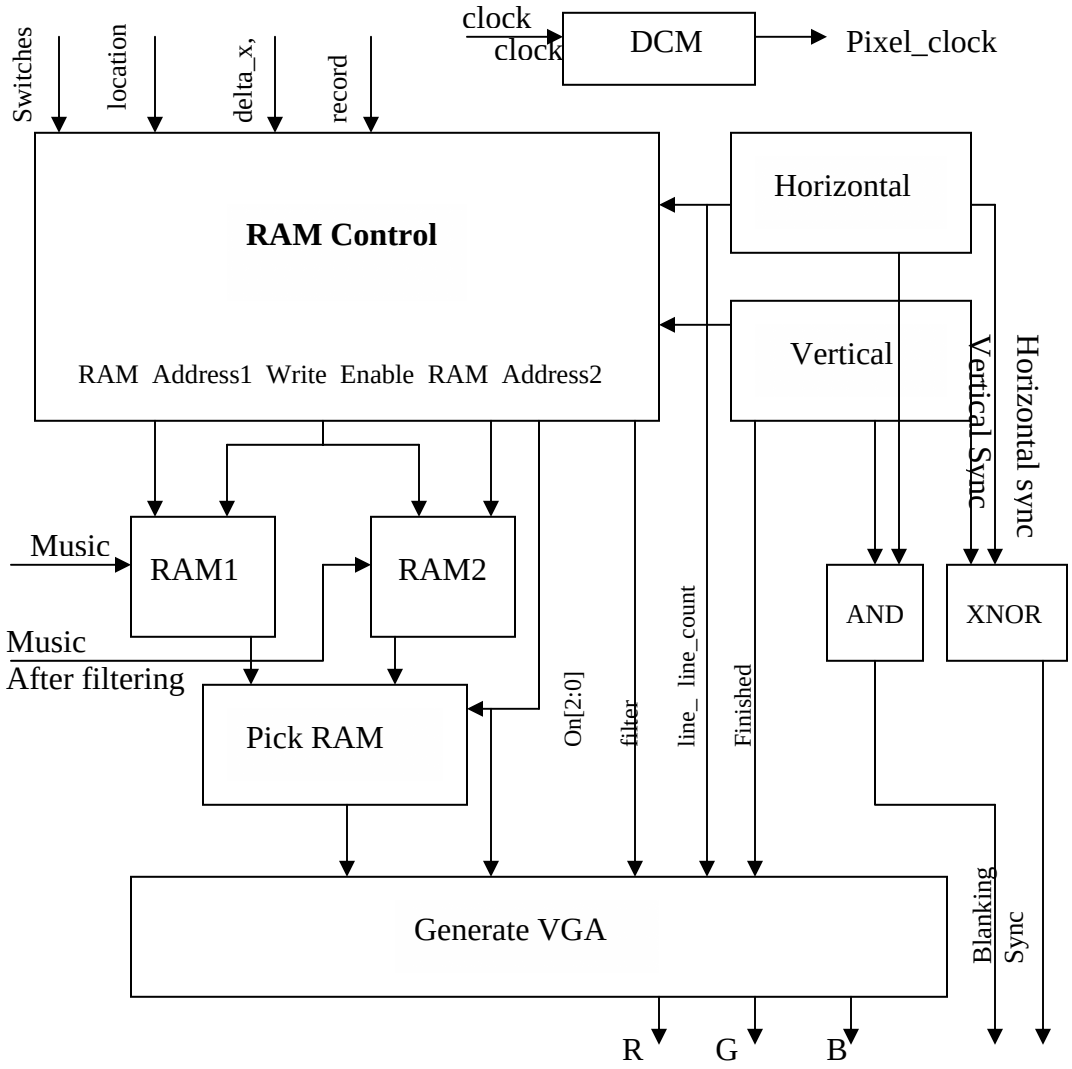


Figure 4.5 VGA Block Diagram

The RAM Control module takes the pixel clock and reset as inputs, along line_finished, vertical_finished, and the vertical count. Not only does the RAM Control module generate the addresses and write enables for the music RAMs, it also generates signals (wave_on signals) that indicate which wave is being shown. The RAM Control module also generates the filter from the input location (coordinates of the band-pass filter), the switches to choose a filter, and the change in the location of the LIGHT. The RAM Control module also writes 800 16-bit properly spaced music samples to each RAM during idle states when prompted by the button record.

The RAM Control module's state machine alternates between being idle between waveforms and setting wave_on 1, 2, and 3 to have the right values as well as supplying the correct RAM addresses, and generating the filter. When supplying RAM addresses, the module begins at 0, then increases to 799, at which point the RAM address resets itself to zero. The RAM Control module then waits until line_finished has been asserted to restart incrementing the RAM addresses. When the filter is to be displayed, the RAM Control module identifies which type of filter to display and displays that filter. The low-pass filter and the high-pass filter are extended by ones and zeros in response to changes in the coordinates of the LIGHT source. The band-pass filter's location varies according to the input location.

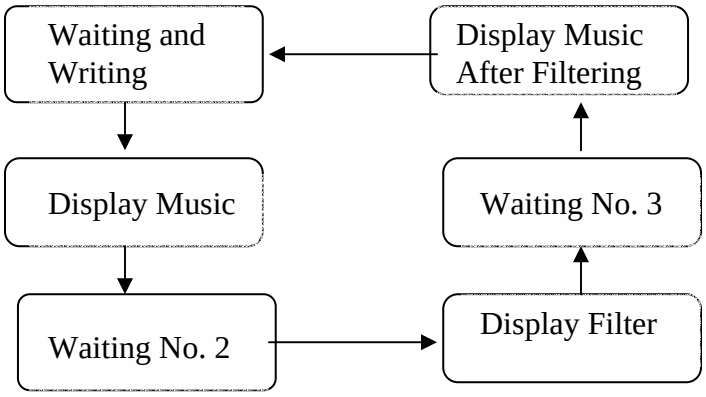


Figure 4.6 RAM Control State Transition Diagram

The outputs from the two RAMs, which were instantiated in the Xilinx Core Generator, are then fed to another module. Using the wave_on signals allows the pick RAM module to identify which RAM's outputs to pass as the ram_output to the Video Generator module. The Video Generator module takes the ram_output signal as well as the filter signal and generates the RGB values for each pixel. The Pixel Generator also takes the vertical count, line_finished, and all three wave_on signals as inputs. If all of the wave_on signals are low, the video generator assigns the background color values to red, green, and blue. If wave_on1 or wave_on3 are high, they correspond to the music before and after filtering. The generator module compares the value of the seven most significant digits from the RAM to a reducing value corresponding to the position within

the region occupied by the waveform. If they are equal, the color of the line are assigned to RGB. Otherwise, the background color is assigned.

The generator module determines the pixel values for the filter region when wave_on2 is high similarly. Each bit of the filter is retained for four clock cycles, then shifted to the back of the filter. This round robin approach allows the module to check for ones at one height in the display and for zeroes at another. Additionally, since the filters are ideal (vertical sides), the current first bit of the filter and the second bit are compared. If they are not equal, then the RBG values are set to the line level instead of the background color. Additionally, to prevent the filter from continuing to shift during the horizontal sync periods, a clock increments to 800, at which point the shifting stops until the next line_finished signal is asserted.

4.4 Design, Trade-Offs and Testing

I began the design process by creating the modules that generate the control signals. However, in the first implementation, I included code that generated RGB values in a pattern. This allowed me to ensure that all the counters and the timing values are correct and that the delays from the ADV715 have been compensated for.

The second part of the design and implementation process involved designing the display function for the music. However, displaying music involved reading from the correct RAM and displaying the signal only on a specific part of the screen. The next step involves identifying which pixels to change to something other than the background color. After the display functions correctly, I had to devise a way to update the music RAMs when appropriate. The original plan was to have music scroll across the screen. However, after calculating the possible frequencies and realizing that the music would either scroll so quickly it would be meaningless or the displayed music would have little relation to the actual music. Therefore, I designed a system that updates the values in the RAMs every time the record button is pushed. This allows the user to update the music being displayed as often or seldom as desired.

The last major part of the video implementation was creating the correct filter based on inputs from the filtering portion of the lab as well as the switches that determine which filter is being used. After the filter is created, it has to be displayed on the correct portion of the screen. The next challenge was implementing another method of displaying the long vector instead of a 16-bit vector at every pixel. Additionally, this had to be implemented so that it could be updated as the location of the **LIGHT** changes.

Unfortunately, the output from the combination of the audio, camera, and convolution was not functional. This caused some difficulties in debugging my section of the lab. Since the RAM Control module is dependent on variables generated in the convolution section of the project, it is impossible for me to guarantee and fully check the functionality of my video implementation. However, I have established that my video implementation does properly display static inputs for the music and that the filter is

indeed dependent on the switches. The limited amount of testing I was able to do indicates that the filter also varies according to delta_x and the location of the filter.

5.1 Conclusion

Sadly, our project did not completely come to fruition. Although most of the parts seemed to work on their own, knitting them together in the end produced errors and an incomplete project. If we had had more time, I believe that it would have been a greater success than it was. Regardless, we all learned a lot, and enjoyed the whole creative process.