The Singing River Project

Kyle Gilpin, Kushan Surana, Andrew Wong
Spring 2004 - 6.111

A system was designed to dynamically modify the pitch and volume of any sound. The system allows the user to increase or decrease the pitch of a sound to arbitrary frequencies without affecting the time-domain characteristics of the sound. To determine the pitch and volume, a vision subsystem tracks a user's hand movements in two dimensions. Vertical movement varies the volume of the reproduced sound, and horizontal movements vary the pitch. A second subsystem estimates the fundamental frequency of the sound using a time-domain (autocorrelation) method. This subsystem and the vision subsystem, transmit pitch data serially to a pitch shifting subsystem. The pitch shifting subsystem reproduces the input sound at the frequency selected by the user.

While the hardware implementation was not ultimately successful, the system functioned correctly when simulated. More importantly, the project suggests realistic methods to accomplish complex tasks.

Introduction

The Singing River Project converges both video and audio processing to create a new musical instrument. The idea is to input an audio signal and video of hand movements, and output audio transformed in frequency and volume. The concept is similar to that of the Theremin, where the pitch of an electronic sound is changed with when the hand moves from side to side. This system captures the information of the hand optically, and interprets the horizontal displacement as frequency and the vertical displacement as volume. The input audio source can be any sound with a fundamental frequency. This frequency is determined through audio processing and then pitch shifted to the desired frequency signaled by the hand. The volume or gain of the output is then changed accordingly.

The Singing River Project can be used to harness natural period signals and turn them into musical sounds, creating a new interface of musical expression between man and nature. The system also allows multi-user control over a single music instrument, as separate users can control the timbre quality of the input sound and the output frequency and volume independently.

Vision Subsystem

Overview

The vision subsystem is the user interface for the Singing River Project. The subsystem translates a user's hand movements into variations in the volume and pitch of the input sound. By moving his hand vertically, the user controls the volume. Lateral movement controls pitch. Both volume and pitch can be modified simultaneously.

Physical Setup

To detect a user's hand movements, the vision subsystem uses a laser and a black and white CCD camera. The laser includes a beam splitter that fans a beam of laser light out over 90°. Unlike a typical laser, when pointed at a wall, the laser in the vision subsystem projects a line of light. The camera is a black and white CCD camera sold as a "Surveillance Camera" at Radio Shack (#49-2515). The camera and laser both point upward. The camera, placed to the side of the laser, is tilted by an angle, θ, as shown in Figure 1.

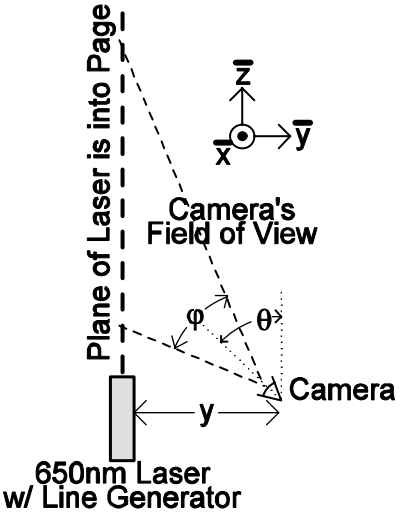


Figure 1: Laser and Camera Physical Arrangement

The laser projects a line that is normal to the page. For an isometric view of the setup, consult Figure 2. Notice that the camera is rotated 90° about the normal to its lens. As a result, data will be transmitted one column at a time as opposed to the traditional row at a time. The rationale behind this setup will be explained in the Implementation section below.

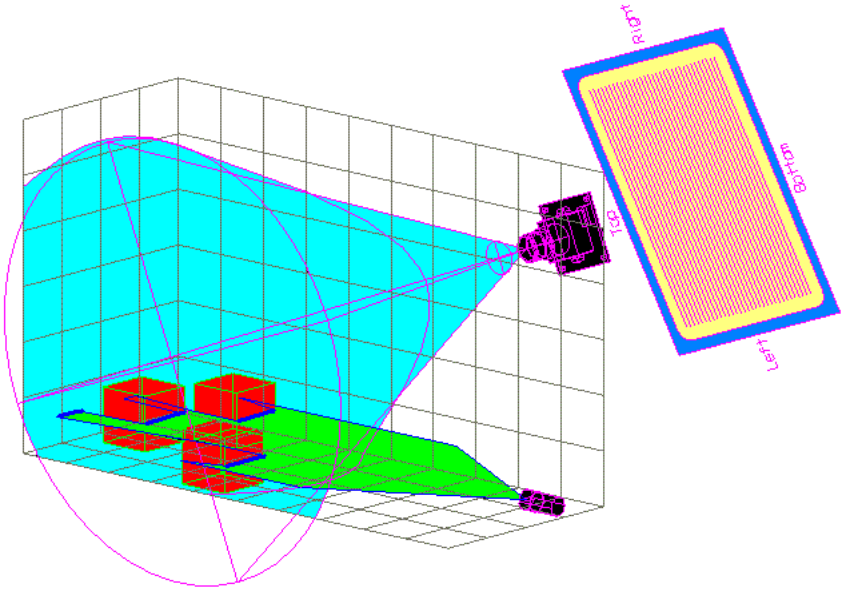


Figure 2: Isometric View of Laser and Camera[1]

Theory of Operation

By separating the camera from the plane of the laser, the system can detect vertical movements of the user's hand. Consider if the camera were centered below the plane of the laser and pointing directly upward. Such a setup corresponds to y=0 and $\theta = 0$ in Figure 2. With this setup, reflections of the laser off the user's hand appear as a line in the center of the image produced by the camera.

Moving the camera out of the plane of the laser (increasing y and θ) allows vertical motion of the user's hand to be translated into changing the placement of the reflected light in the image produced by the camera. As the user moves her hand downward, the point of reflection moves toward the lower edge of the camera's field of view. Hence, the reflection appears lower in the picture. Conversely, as the point of reflection moves upward, it reaches the other extreme of the camera's field of vision.

Finally, lateral movements of the user's hand in the in x direction (normal to the plane of the paper as shown in Figure 1), move the reflected light from side to side in the camera image. For an example of how the vision subsystem operates, examine Figure 3 and Figure 4. Figure 3 shows a laser line projected onto five objects. Figure 4 shows just the laser with all other incident light removed. Objects that are closer to the camera and laser appear as lines lower on the image than those objects that are far away. The vision subsystem is responsible for filtering and analyzing the resulting image.

---

[1] Maxon, Kenneth. "A Real-time Laser Range Finding Vision System." *The Encoder*. October 2001. <http://www.seattlerobotics.org/encoder/200110/vision.htm

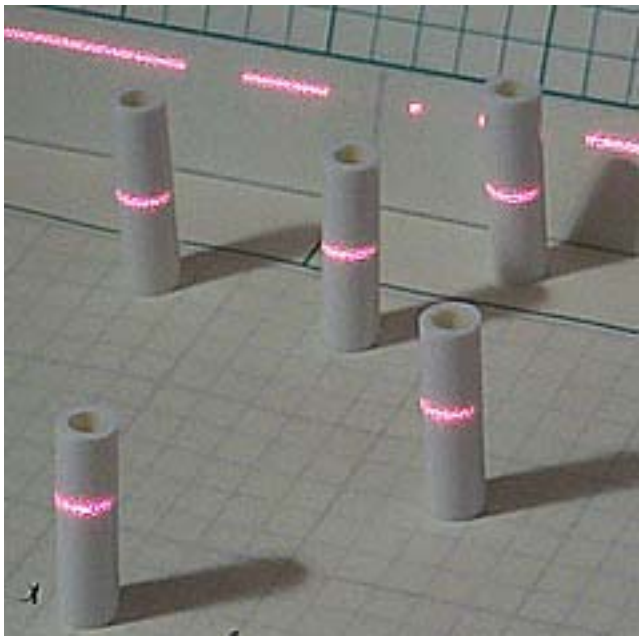


Figure 3: Laser Reflections[2]

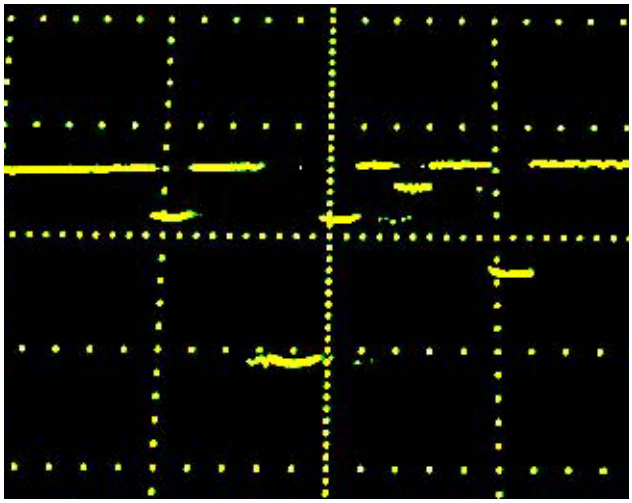


Figure 4: Reflections Alone[3]

Theoretical Implementation

Two strategies were attempted to extract useful information from the image captured by the camera. First, a band pass filter with high Q was placed directly in front of the camera lens. The filter was chosen to only pass the laser light. This is possible because the light emitted from the laser propagated almost exclusively at 650nm. It was hoped that the filter would restrict the camera seeing only see the laser's reflection. The optical filter failed for two reasons. First, the ambient intensity of 650nm radiation is remarkably large. Natural sunlight, incandescent lights and fluorescent lights all produce enough 650nm radiation to drown out the laser under certain conditions. Second, and more important, the filter had a field of view that was narrower than the camera's. Any light incident on the filter from more than a 20° angle was rejected regardless of its wavelength. If the band pass filter had been employed, the user would have been constrained to moving his hand in a very small space.

The second strategy involved capturing two image frames, one with the laser on, one with the laser off, and differencing them. By subtracting the two frames, captured only a fraction of a second apart, it was hoped that the result would be an image containing just the laser as in Figure 4. To implement such a strategy, one image must be stored while a second is captured. Additionally, the resolution of the stored image must be great enough to preserve the high frequency components of the image corresponding to the laser. Instead of storing the whole image, the vision subsystem stored one "metapixel" for every five pixels sampled by the analog to digital converter. The value of a metapixel was the sum of the two brightest pixels out of every five pixels sampled. The image was reduced to two fifths of its initial size. More specifically, the stored image contained 330 lines, each which contained 100 metapixels.

---

[2] Ibid.
[3] Ibid.

After one image has been captured and stored in RAM, data from the next frame is filtered and subtracted from the stored image in real time. Now, one can see the advantage of rotating the camera by 90°. Each column should only contain one pixel of reflected light. Therefore, every column can be processed independently of the others by finding the most intense metapixel in each column

Averaging the position of the brightest metapixels over all the columns produces an estimate of the vertical position of one's hand. As each column is processed, the locations of the brightest metapixels, assuming they are brighter than a present threshold, are summed. After the frame has ended, the sum is divided by number of columns that contained metapixels exceeding the aforementioned intensity threshold.

The lateral position of a user's hand is produced in a similar manner. In this case, the numbers of the columns containing metapixels (exceeding a threshold) are summed. The camera produced 330 columns of data. The sum of the column numbers is again divided by the number of columns containing metapixels meeting the threshold requirement.

Hardware Implementation

The vision subsystem was composed an analog to digital converter (AD775), a video sync separator (GS4981), two 32Kx8 RAM chips (HM62256), and an FPGA as shown by the block diagram of Figure 5.

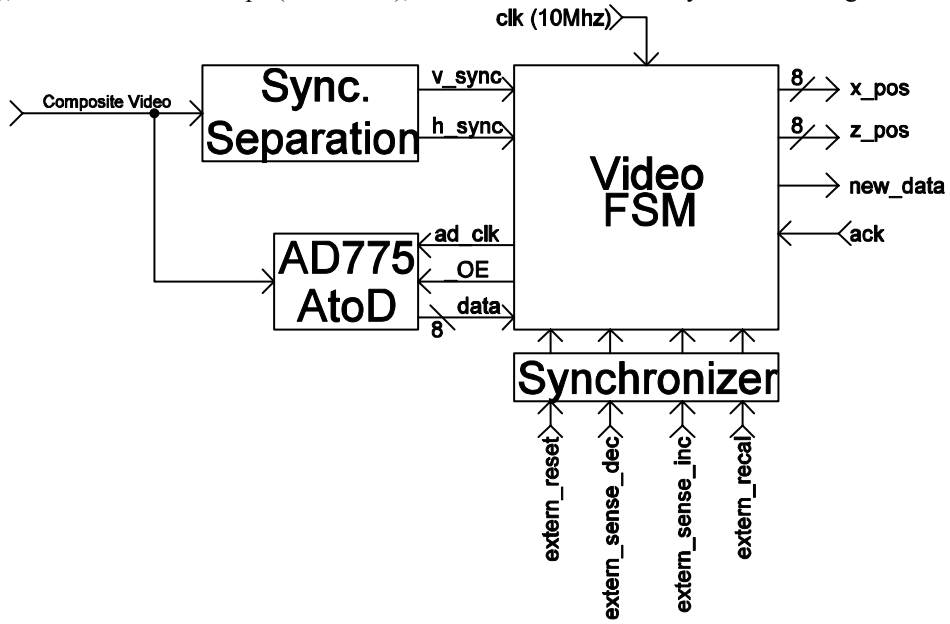


Figure 5: Vision Subsystem Block Diagram

The video FSM was composed a several hierarchical modules which allowed for extremely fast processing of data from the camera. The system ran at *10MHz,* and completed an analog to digital conversion each clock cycle. The RAM was not fast enough to store all of this data but the filtering algorithm described in the Theoretical Implementation section above worked well. The tightly interwoven sampling and memory access FSM's lay at the bottom of the hierarchy. The sampling FSM produces one metapixel for every five samples of the camera input. The memory access FSM, given a column number, reads the previously frame's metapixel at a specific location and subsequently stores the new metapixel value.

The column FSM sits atop the memory access and sampling FSM's. It directs the memory access FSM when and where to read and write data. Additionally, given a metapixel from the previous frame and a metapixel from the current frame it computes their difference. As the FSM scans down a column, it

determines the maximum difference and its location within the column. The difference must meet a certain threshold to qualify as a maximum.

This location information is passed up to a frame FSM. The frame FSM tabulates pixel location information. It sums the locations of the brightest metapixels in each column. Additionally, it sums the column numbers of the columns that contain valid metapixels. After the camera has transmitted an entire frame, the FSM divides these two numbers by the number of columns that contained valid metapixels. These quotients represent the volume and pitch information respectively.

The volume and pitch information are taken by the top level FSM and serially transmitted to a digitally controlled potentiometer (MCP41010) and the pitch shifting subsystem respectively. The digitally controlled potentiometer is connected to an OP-AMP to control the gain of the output from the pitch shifting subsystem. The top level FSM is also responsible for turning the laser on and off with each new frame.

Proof of Concept Images

Figure 6: First Frame

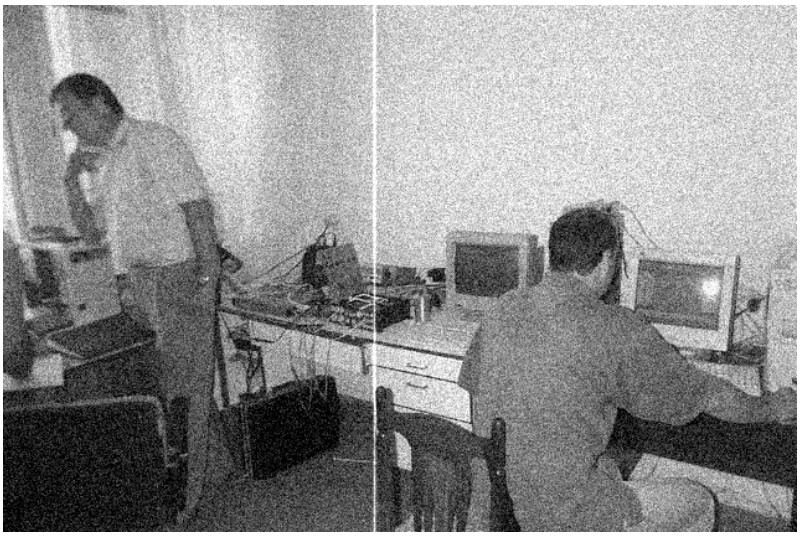


Figure 7: Second Frame -- Noise and Laser

Figure 8: First and Second Frames Filtered and Subtracted

Figure 9: First and Second Frames Filtered, Subtracted, and Threshold Applied

Fundamental Frequency Estimator

Overview

This section describes the implementation of an autocorrelation function in hardware which computes the fundamental frequency of the input signal. Fundamental frequency detection of a rapidly changing sound source requires more complex algorithms and hence it has been assumed that the source is not changing too rapidly in frequency.

The design can be split into two sections: storing the analog data into the 6264 RAM and the autocorrelation module. The analog signal is sampled at a rate of 40 KHz and the data is stored into the first 1024 addresses in RAM. During the sampling process, the autocorrelation module remains idle. After 1024 samples have been collected, the autocorrelation module is set to active. The sampled signal is multiplied by itself for a range of values for a range of lags. This "lag" is simulated in hardware by accessing different addresses. Instead of storing all the values of the autocorrelation function in RAM, the adjacent values are compared dynamically after the computation for each lag is finished. This ensures a faster estimation of the fundamental frequency instead of storing all the data-points for the autocorrelation function in RAM which would slow down the rate of fundamental frequency estimation.

Description

The design of the system was done using a major-minor FSM architecture. Since the storing of the data into RAM and implementing the autocorrelation module was done in two distinct phases requiring no overlaps, the design is fairly simple. Each of the individual modules is described below starting off with the major FSM itself. When relevant, a few words have been included about the chips being used in the design process and the control signals required by them. It goes without saying that the datasheets for the chips should be consulted for a more detailed description.

Major FSM

The major FSM consists of five states. The first state initiates the analog to digital conversion and transitions to the second state where a signal from the minor FSM is awaited indicating a completion of the analog to digital conversion process. Once the data is ready on the bus of the ADC12441 chip, a signal is sent to the minor FSM implementing the writing to the 6264 RAM chip. It must be noted that the **read_bar** and the **cs_bar** signals to the ADC12441 chip are pulled low before the minor FSM executing the writing to RAM is called and kept low throughout until the minor FSM indicates the writing to RAM has been completed. This is done to ensure that the digital data remains on the ADC12441 chip. The chip provides for the data to remain on the data bus as long as the **read_bar** signal is held low after the conversion process. The conversion of the analog signal to digital bits is carried out 1024 times following which the **flag_correlation** signal is set high which initiates the minor FSM carrying out the autocorrelation module. The bulk of the processing is carried out within the module and after a done signal is sent by this minor FSM, the fundamental frequency of the input signal is output. See Figure 10 for the state transition diagram of the major FSM.

Analog-To-Digital Conversion

Once the **start_ad** signal is received from the major FSM, the analog to digital conversion is initialed. The **write_bar** signal is sent to the ADC12441 for three clock cycles, or for a period of 300ns, which is more than the minimum of 200ns required to initiate an analog to digital conversion. Once the conversion process has started, the **int_bar** signal from the chip is awaited which indicates that the conversion process has been completed. The **read_bar** signal is pulled low at this point which outputs the converted result to the data-bus. This is also the point where the **done_ad** signal is set to high signaling the major FSM to begin writing the digital data to RAM. Once the write to RAM has been completed, the **read_bar** signal is pulled high. The **cs_bar** signal is held low throughout this process ensuring that the ADC12441 chip is enabled. See Figure 11 for the state transition diagram of the analog-to-digital minor FSM.

Writing-To-Ram

After the digital to analog conversion has been completed, indicated by the **done_ad** signal, the major FSM sends a **start_write_to_ram** signal to this minor FSM. An internal counter establishes the address where this digital data is supposed to be written. After the address has been registered to ensure that there are no glitches in the address, the **w_bar** and the **e_bar** signals to the 6264 SRAM are held low for a period of 100ns. This is far above the 8ns minimum limit specified in the 6264 datasheet to ensure a clean write. After the data has been written to the particular address on RAM, the **done_write_to_ram** signal is set high and sent to the major FSM. See Figure 12 for the state transition diagram of the writing-to-ram minor FSM.

Autocorrelation

Once the **done_write_to_ram** signal is received by the major FSM, the start_correlation signal is set to high and the autocorrelation minor FSM is entered. This module has a total of twenty-two states, of which five are redundant, and are added to ensure that the operations are not being carried out too aggressively.

This was done because at the time of writing, the system is not yet functional and it was hoped that the addition of these states and the implementation of a more conservative design would help remedy the problem. After the **start_correlation** signal is received, the **read_address** is set to a counter value. The **read_address value** rotates between two counters called read1_address and read2_address. While both the counters increment at the same rate, read1_address also takes the lag into account when initiating itself. For example, after the signal has been multiplied with itself for lag 0, the signal has to be shifted one sample and then multiplied by the original unshifted signal. In this scenario, **read2_address** would start at 0, but read1_address would start at address 1. Once **read1_address** has been registered to ensure that the signal is glitch-free, **read_address** is assigned to **read1_address**. Once the address is stable, the **g_bar** and **e_bar** signals to the 6264 RAM are set high and the read from the SRAM is completed and the result stored in a register. The same process is followed with the **read2_address** and the data stored in a separate register. Once both the reads have been completed, the **start_mult** signal is sent to the major FSM which starts the multiplication portion of the multiply-and-accumulate module. The **start_accumulate** signal is set to high at the next clock edge which starts the accumulate portion of the multiply-and-accumulate module. After **read1_address** has reached the last address of 1024, the accumulated value is stored in another internal register called **reg0**. There are three internal registers called **reg0**, **reg1** and **reg2**. While **reg0** always takes in the accumulated value, reg1 takes the value of **reg0** while **reg2** takes in the value of reg1 after the completion of each autocorrelation cycle. The values in these registers are compared and if the value in **reg1** is greater than the values in reg0 and **reg2**, it indicates the presence of a local peak. There is another internal register called **marker_address** which stores the initial value of **read1_address** at the beginning of each autocorrelation cycle. Once the local peak has been found, a **start_divide** signal is sent to the major FSM which initializes the divider module. The **done_divide** signal from the divider module is awaited after which the fundamental frequency is updated and a **done_correlation** signal sent to the major FSM. This indicates the end of the autocorrelation module and the write to RAM cycle is started again.

Divider module

Altera does not support division. Hence, a divider module needed to be coded so that the fundamental frequency could be calculated. Once the marker address corresponding to the local maxima has been found, the **start_divide** signal is sent to the divider module along with the **marker_address**. A parameter of 40,000 which happens to be the sampling rate is divided by the marker address by multiplying the **marker_address** with incrementing an integer i until the product is greater than 40,000. The number i – 1 corresponds to the estimated fundamental frequency.

Multiplier and Accumulator Module

In order to calculate the correlation function, a running sum of all the multiplications corresponding to every lag is performed. It is this result which is stored in the individual registers referred to as **reg0**, **reg1** and **reg2** in the correlation module. The multiplication of two's complement numbers was done by taking the magnitude of the two numbers and using the asterisk (*) operator provided by Verilog. In addition, the most significant bits of the two numbers was analyzed and if they were different, then the number was made negative by flipping all the bits and adding one to the result.

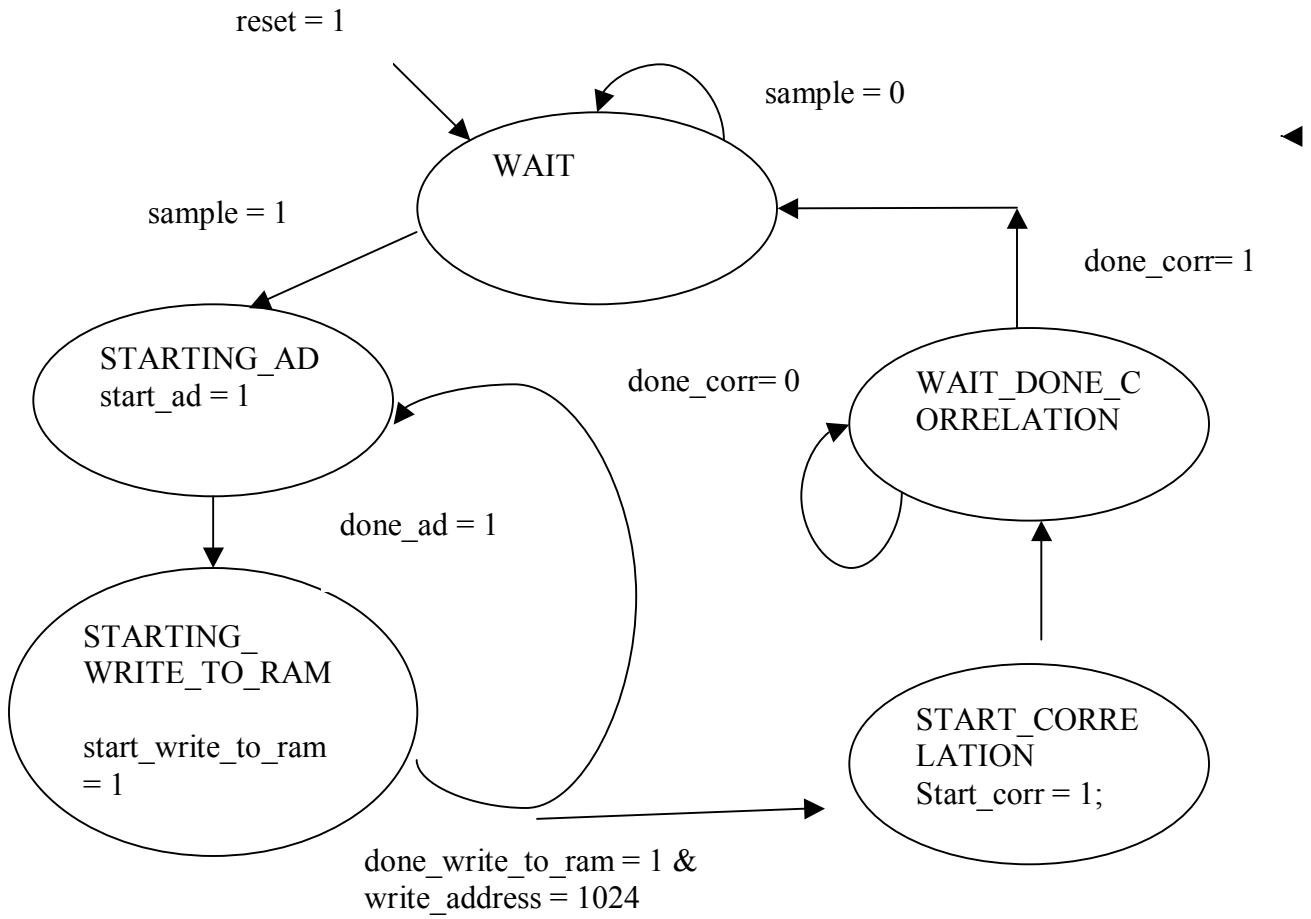


Figure 10: State transition diagram for major FSM controlling the individual minor FSMs

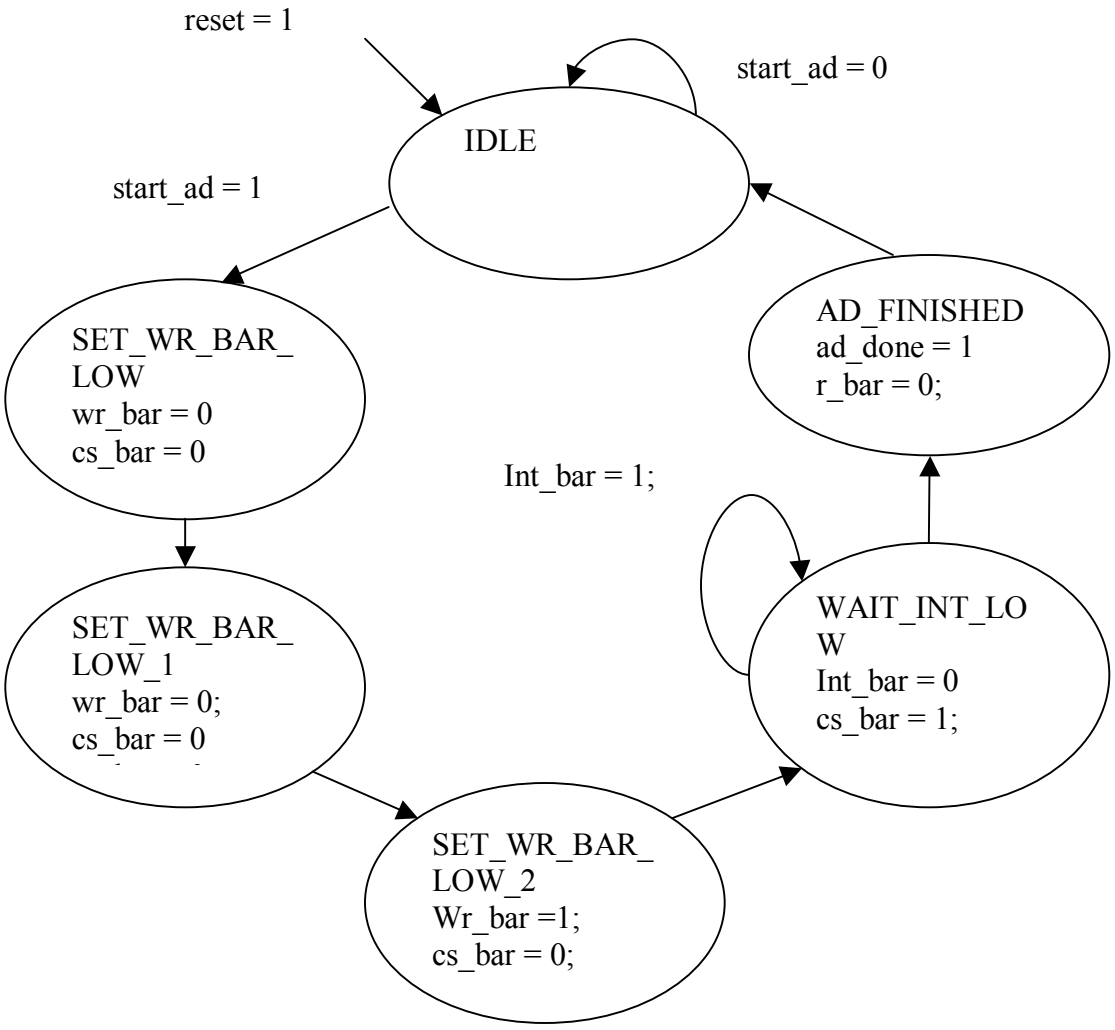


Figure 11: State transition diagram for Analog to Digital conversion

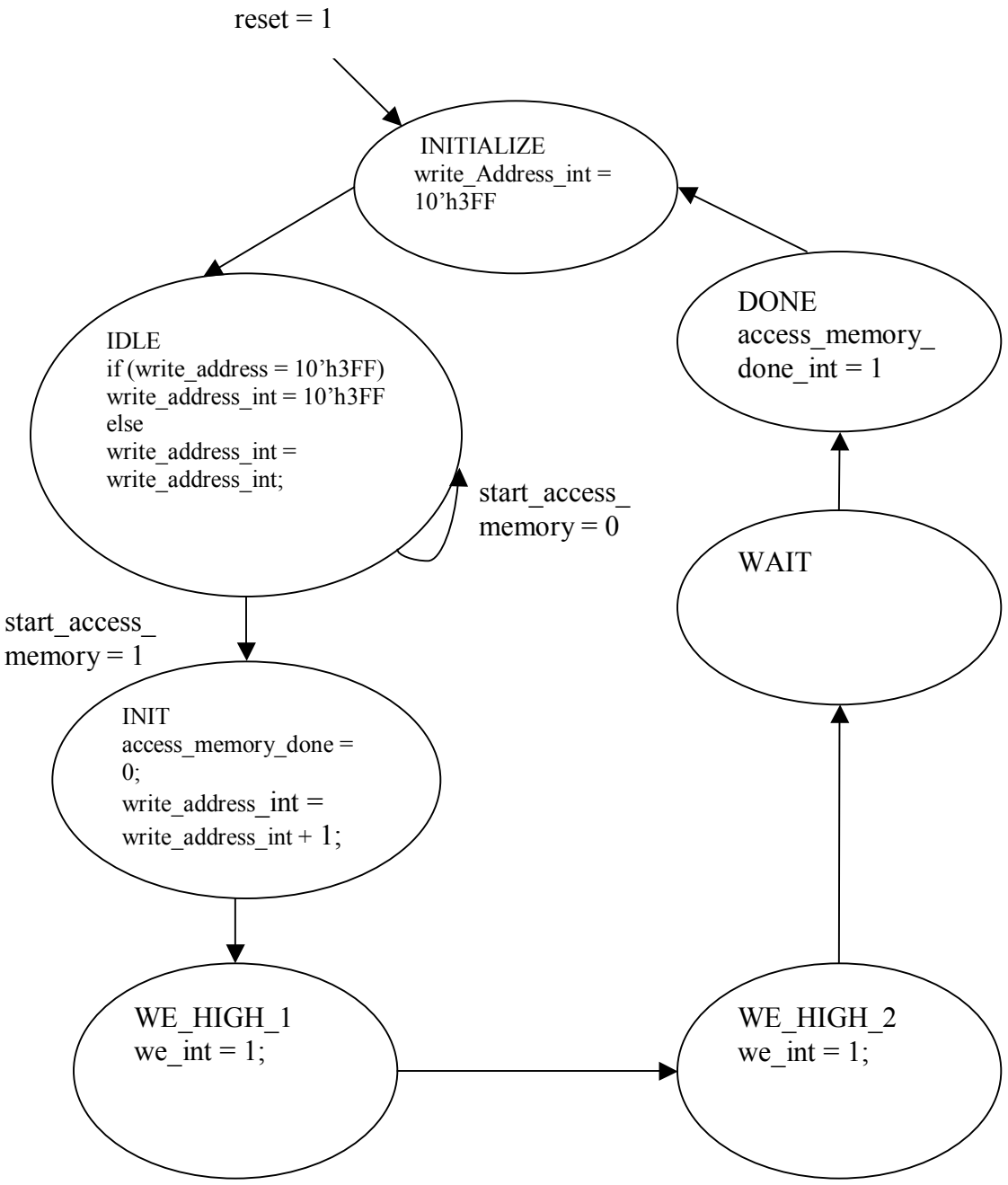


Figure 12: State transition diagram for storing the registered analog to digital data into the SRAM

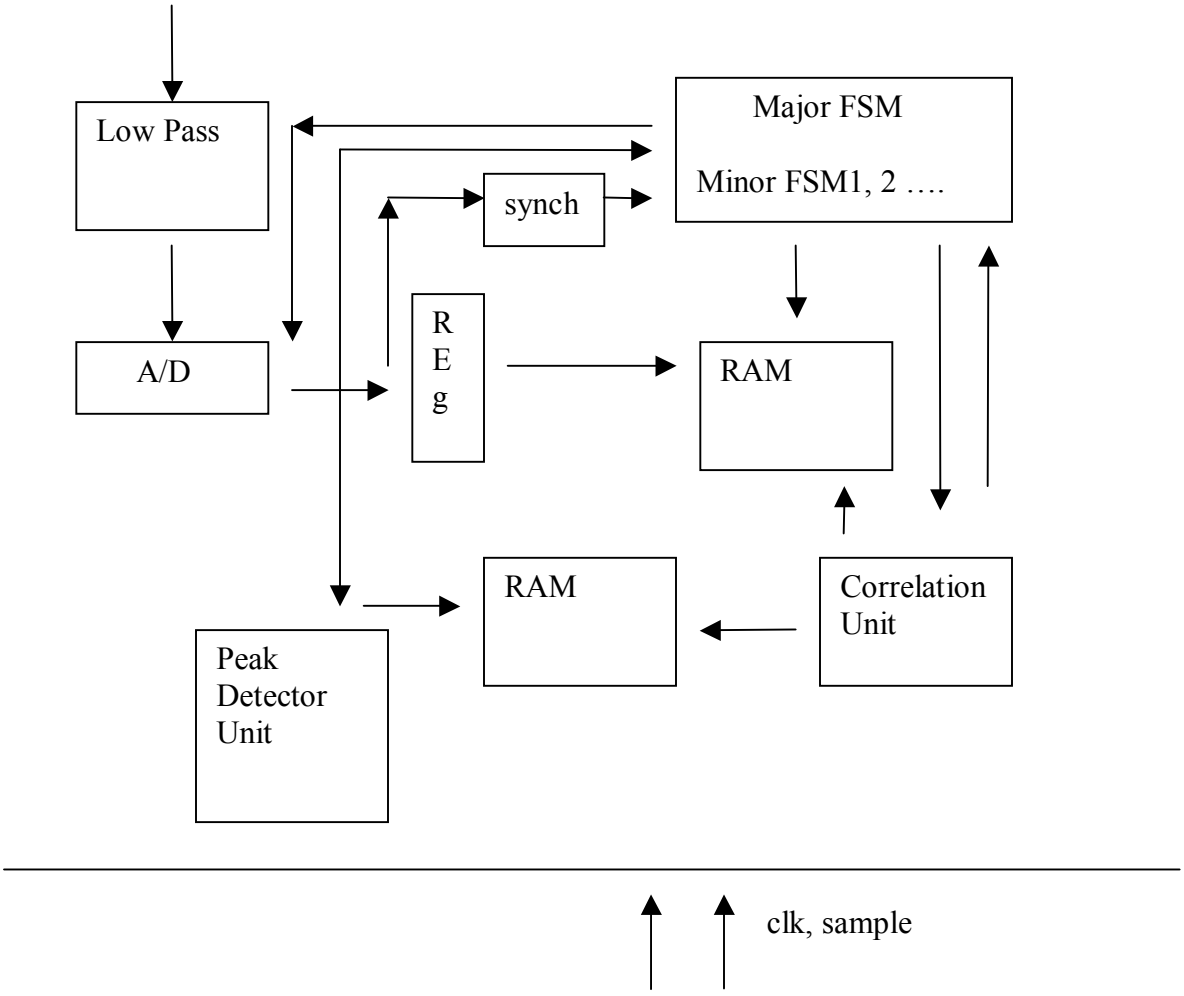


Figure 13: Overall block diagram of the system

Pitch Shifting

Introduction:

This part of the project was to implement a pitch shifter that could take in the fundamental frequency, a target pitch frequency, and shift the input to that new frequency. This technique is used commonly in synthesizers, audio software, and guitar effects pedals. Converting the fundamental frequency of an input can be performed in the time or frequency domain. Given the limitations of the hardware, time domain manipulation was applied to the design. In order to raise or lower the pitch, the individual samples of the input are played back at a higher or lower "rates," respectively. The actual output rate remains the same at 40 kHz, but samples are chosen at different intervals. For example, doubling the frequency would be equivalent to outputting every other sample.

Pitch shifting a finite sound sample is simple since the length of the output sample is not very important. For a real-time system, however, the output should be the same length as the input. Therefore, a time-expansion technique must be applied. The majority of this project was spent developing a suitable time-expansion, time-compression technique that could operate fast enough in hardware. The technique developed was based on the SOLAF (Synchronous Overlap-Add Algorithm), but substantially faster. The hardware implementation failed, however, and a much simpler method of pitch shifting was chosen. The author believes the design developed for the SOLAF-based algorithm is much more valuable and informative, and therefore will describe the design in detail. A brief design description of the simple pitch shifter is also described below.

Overview:

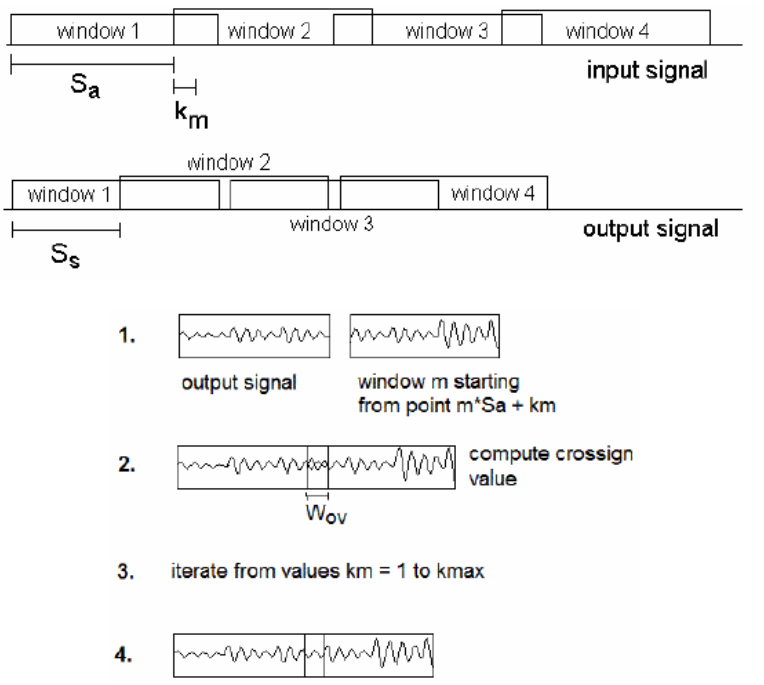
The SOLAF algorithm is a robust algorithm used to time compress or expand periodic signals. The basic concept is to split a sound sample into small overlapping windows with a specified overlap Sa. The windows are then stitched together with another overlap size Ss. The ratio of Sa to Ss determines the final size of the sound. The beauty of the algorithm, however, lies in the variation in overlap Sa. For each window, the best overlap with the output is calculated such that the window is an additional km samples away from the default position. These small variations allow the windows to stitch together to form a continuous waveform. The newly expanded or compressed waveform can then be resampled to the desired frequency and original time length.

The original SOLAF algorithm requires a calculation of the starting position of each window. This proved to be almost impossible for the hardware implementation. Instead, a simpler version of this algorithm was developed, with surprisingly good results. It is assumed that the input sound source will fill up a dual buffer, each of 4000 samples long. Therefore, while the input is filling up one buffer of 4000 samples at 40 kHz with a 10MHz clock, the system has 1 million clock cycles to write a new 4000-length buffer for the output. The algorithm is described as follows:

1. Resample the 4000 input buffer S1 to a second buffer S2 of size 4000, with the interval determined by the frequency desired.
2. Begin calculating the optimal overlap of the last 300 samples of S1 with S2. Instead of using a cross-correlation for this, the sign bits of each sample were compared and not(xor)'d. If the samples were correlated in sign, a score of 1 would be received. If they were anticorrelated, a 0. The scores are then summed for each lag and the best lag Kmbest is chosen. A maximum possible lag was chosen such that the calculation never had to run more than 400 lags.
3. Write to the output buffer S3 from S2, starting from the lag Kmbest. This ensures connectivity between the individual buffers. Writing continues until the last sample of S2 has been written.
4. Step 2 is performed again, this time to determine the best overlap between S2 and S3. A second Kmbest2 is generated.

5. Write to the output buffer S3 from S2, continuing from where it stopped before, using samples starting from Kmbest2 until the end of S2, or if S3 has 4000 samples.
6. Check to see if S3 is filled up. If so, the process is complete, otherwise iterate 5 with the same Kmbest2 until complete.

In this algorithm, the most time-consuming step is to determine the best overlap. This was projected to take around 300,000 clock cycles. By limiting the algorithm to two calculations of this overlap, the system takes at most 600,000 cycles to perform the dirty work, and leaves 400,00 cycles to write to resample S1 and write to S3.



Windowing and Cross-Sign computation.

Block diagram of system.

The key feature of this system is the shared bus of 13 bits for the A/D converter, D/A, converter and 3 SRAMS (6 in reality because each stores only 8 bits). A bi-directional block receives requests by the minor FSMs to either drive the bus or set it to high impedance. All the SRAMs are connected to the same 13 bit address line. An additional 3-bits are tagged to addresses, one for the enable of each SRAM. This is okay since only SRAM is operated at a time. Therefore, as far as the major FSM is concerned, there is only one SRAM with 16 bits of addresses. For control, a Major FSM acts as traffic controller between the A/D minor FSM, which continuously writes to the S1 buffer, and the Cross-sign, Output-writer, and Resampler minor FSMs.

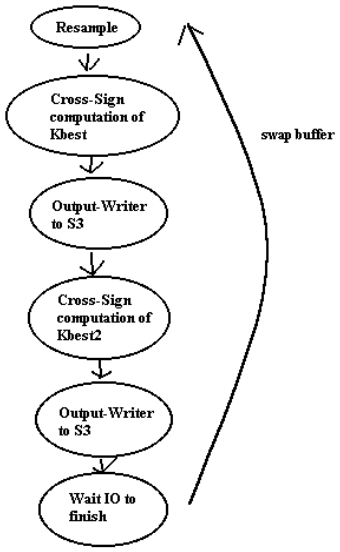
Major FSM:

The Major FSM controls the flow of the operation of minor FSMs. It controls which buffer is being written to and sends requests to the Bidirectional block, Address Manager, and SRAM R/W manager. Since the processing of the S3 buffer must be done fast, while the IO block writes to the S1 block, a series of pause/unpause commands must given to the minor FSMs. When the Major FSM receives a request from the IO block, it sends a pause to the minor FSM in operation at the time. This minor fsm will work continue its loop until it gets backa to the beginning and then send a paused signal to the Major FSM and go into the

paused state. When the IO block is finished writing and outputting to the D/A, the Major FSM unpauses the minor FSM and it continues its cycle. Each minor FSM has its own pause/unpause interface, which is possible because only one minor FSM operates at a time.

I/O block

This minor FSM receives a 40 kHz sampling pulse from the Divider. Upon each sample, it activates the enable signals for the A/D chip. The chip, an AD12441 has internal latching, and therefore does not interfere with the shared bus until called to release the data. This is convenient because of the 13us conversion time. Once the conversion is complete, the chip sends a signal to the fsm, and it tells the major FSM that data is waiting. It waits for a confirmed read command from the Major FSM, before driving the bus (from the AD chip) with the newly acquired data. The IO block also has control over S1, and activates the memory at the same time in write mode to write to the appropriate buffer. After it finishes writing to S1, it then activates S3 in read mode to drive the bus with data to output to the D/A. Therefore at every sample, the IO block inputs and outputs one sample from opposite buffers.



Control flow of System.

Resampler

This minor fsm requests a rate from the Rate Converter every time it is activated by the Major FSM Its responsibility is to write from the input buffer to S2. In order to do this, it has control over the Bi-directional block. It sets the bus to high impedance, addresses the S1 SRAM in read mode, then stores the data in an input register. It then drives the bus, enables the S2 SRAM in write mode, and loads the new data on to the bus. The details of address are controlled in this fsm as well. It has an internal counter that increments the address by the rate until either a) the input buffer S1 is used up, or b) S2 is filled with 4000 samples. Note that this means S2 could have less than 4000 samples. The addresses are routed through the AddressManager, which controls which address is being outputted at any given time.

Cross-Sign:

This block computes the best lag. It has two operation modes, a) finding the best lag between S2 and the output buffer of S3 and b) between S2 and the written buffer of S3. The fsm reads in the sign bits of both buffers sequentially. It first sets the bus to high impedence, then loads the address for S2. It loads this data into the register Sign1. It then changes the address to S3 and loads the data to register Sign2. A not(xor)

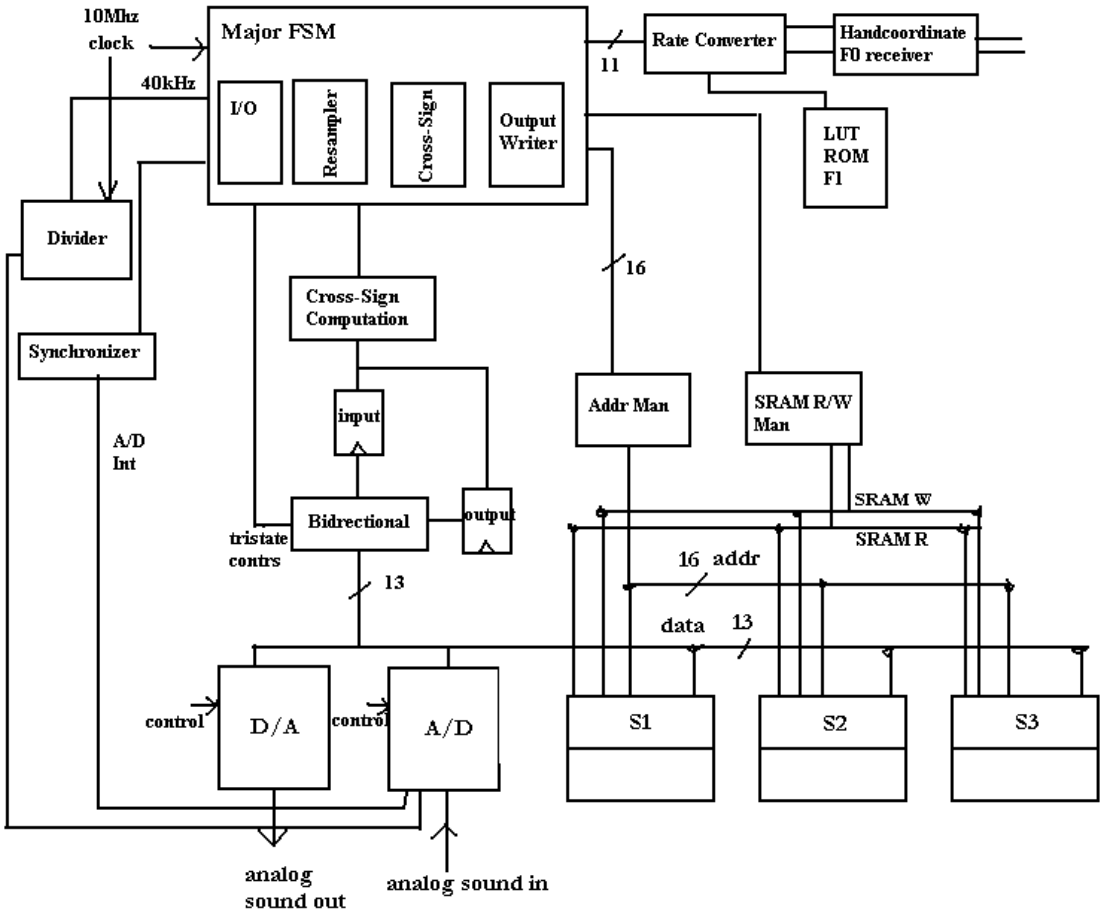
combinational block receives the outputs of both registers and is connected to an accumulator. The fsm pulses the accumulator at the end of each iteration until the window size of 300 has been computed. It then checks if this sum is larger than the previously stored sum. If it is it replaces it and also updates what should be the optimal lag Km.

Output Writer:

This fsm receives the best lag Kmbest from the Cross-Sign block and begins writing to the S3 write buffer from S2. It keeps track of the last address written for S3 and stops when S3 is full. This fsm is called twice, after calculations of Kmbest and Kmbest2. When it is called the second time, it also does not stop until S3 is full.

Rate Converter:

This block loads in F0 and the looks up F1 from the LUT Rom given the 8-bit handcoordinate. F0 and F1 were limited to range from 40-500 Hz. 20-bit division was performed to find the rate to increment the address for the resampler.



Block Diagram of System.

Simple Pitch Shifter

The method of the simple pitch shifter is to preserve the length of the input by looping around the window length until the desired output length is generated. This technique is fast and efficient, but fails to truly output the input at the desired frequency because of the sharp discontinuities that arise from sampling a point at the end of the window and subsequently at the beginning of the window.
The simple pitch shifter uses two buffers—one for filling up samples, and the other for outputting samples at a given rate. The shifter samples from the buffer with different address intervals, given the chosen frequency output. This method requires no calculation for time-expansion or time-compression, but rather continuously outputs samples from the same buffer as long as the input source continues to fill up the other buffer. When the input buffer is full, the output and input buffers are swapped and the process continues. Operation of the pitch shifter is dependent on three external signals: and 9-bit fundamental frequency F0, 8-bit hand coordinate, and a switch selecting whether to play both the original and resampled source simultaneously.

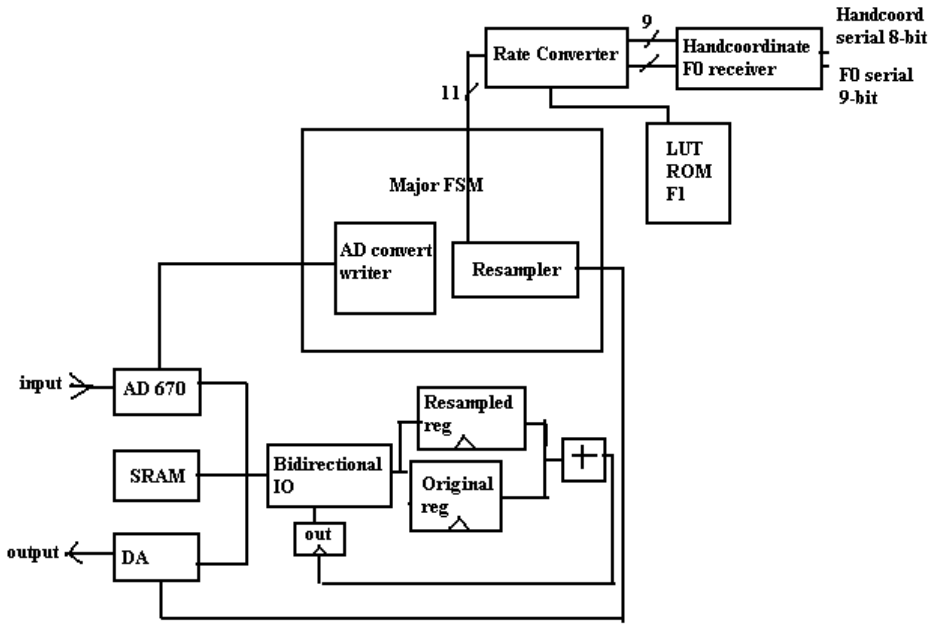
Modules:
The system is comprised of 5 modules: Major FSM, F0/Coordinate Receiver Block, A/D conversion/ writer block, Resampling block, Address Manager. A Synchronizer and Divider block were also created to synchronize external inputs and divide the clock frequency (10MHz) to the sampling frequency (40kHz), and also specify when the sampling rate should be calculated, which was performed the Rate Converter module.

F0/Coordinate receivers:
These modules were developed from Kyle Gilpin's code, such that all three portions of this project were ensured to be compatible. It uses 8-bit transmission for hand coordinates and 9-bit transmission for the fundamental frequency. A synchronized clock signal and select switch tells the system when a new set of bits is ready for transfer.

A/D converter/writer:
This module converts the incoming audio signal via the AD670. It then writes the 8-bit data to an external 8K x 8 6264 SRAM. This module performs all functions with the shared bus at high impedence with respect to the FPGA.

Resampling module:
This block reads from the external SRAM, and loads the chosen sample to the FPGA. If needed, it does this again to get the sample of the original signal. The two inputs are summed together and written back to the shared bus. An AD558 D/A converter then outputs the data to analog form.

Address Manager:
This module receives requests from both the A/D converter/writer module and Resampling module to change the address for the SRAM. The manager controls a 16-bit address output. The top three bits are the enables of three different SRAMs (only one SRAM is used in this design, while the SOLAF-based design required six). The manager calculates the addresses needed by each module, given the current buffer and count supplied by the major FSM.



Simplified Pitch Shifter Block Diagram

Testing And Debugging

Ultimately, the project was not realized in working form. Faulty lab kits and a lack of I/O pins were two obstacles that prevented its success. All three major subsystems of project required a large amount of RAM. To access this RAM, a large number of control and data lines were needed. Many I/O lines were needed to interface with the external world through switches, LED's and analog to digital converters. Several data lines had to be used to interface the three lab kits. Consequently, very few working I/O pins remained for testing the system. It was difficult to even determine the state of an FSM.

Additionally, it was difficult to simulate the external signals used to drive the algorithms. While Max+PlusII was used to simulate each module of every subsystem, the three subsystems all required external analog data to operate correctly. It was incredibly time consuming to input all test data by hand. A simple determination of fundamental frequency required 500,000 data points. A full test of the vision system required close to 330,000 data points. Without more advanced software, these simulations are not possible.

Finally, interfacing to the analog signals was not straightforward. The analog to digital converts proved difficult to work with. They were sensitive to noise and did not always meet timing specifications. For the vision system in particular, the NTSC video signal had to be buffered which proved daunting until a high frequency OP-AMP was used.

Overall, initial slow progress on the analog portions of the subsystems delayed the implementation and testing of the algorithms used to determine hand position, fundamental frequency, and the pitch shifted output sound.

Conclusion

This project proved to be a challenge in both algorithmic and hardware design. New ideas in vision processing and audio processing were designed to suit the hardware limitations. In the future, we hope to upgrade these algorithmic designs, perhaps a adding more pipelined architecture to increase the throughput of the input fundamental frequency and the output signal. Other features worth considering are to add different filters to preserve the "natural" sound of the output as the frequency is increased. Such technology is readily available in music synthesizers today. The Singing River Project still has a long way to go before musicians can easily use it and outputs sound pleasant for listeners, but the foundation has been laid with this initial project.

Verilog Code

Vision subsystem

```
module frame_fsm (
          clk, _reset, enable, busy,
          vsync, hsync,
          AD_data,
          RAM_data,
          RAM_addr, RAM__WE, RAM__OE,
          volume, pitch);

//IO
input clk, _reset, enable;
output busy;
input vsync, hsync;

input [7:0] AD_data;

inout [7:0] RAM_data;
output [15:0] RAM_addr;
output RAM__WE, RAM__OE;

output [7:0] volume, pitch;

//Wires
wire row_fsm_busy;
wire hsync;
wire [7:0] z_coordinate;
wire no_return;
wire [7:0] AD_data;
wire [7:0] RAM_data;
wire [15:0] RAM_addr;
wire RAM__OE, RAM__WE;

wire volume_division_busy, pitch_division_busy;
wire [15:0] volume_quotient, pitch_quotient;

//Registers

reg start_row_fsm, start_row_fsm_int;
reg [8:0] current_row, current_row_int;

reg start_pitch_division, start_pitch_division_int;
reg start_volume_division, start_volume_division_int;

reg [7:0] volume, volume_int, pitch, pitch_int;

reg busy, busy_int;
reg [15:0] x_total, x_total_int;
reg [15:0] z_total, z_total_int;
reg [8:0] number_of_reflections, number_of_reflections_int;
reg [8:0] hsync_count, hsync_count_int;

reg [3:0] state, next_state;

parameter init = 0;
parameter discard_blank_rows_1 = 1;
parameter discard_blank_rows_2 = 2;
parameter wait_for_row_start_1 = 3;
parameter wait_for_row_start_2 = 4;
parameter wait_for_row_fsm_start = 5;
parameter wait_for_row_fsm_complete = 6;
parameter begin_stats = 7;
parameter wait_for_begin_stats = 8;
parameter wait_for_complete_stats = 9;

//Instantiations
```

```
row_fsm row_fsm (
          .clk(clk),
          ._reset(_reset),
          .start(start_row_fsm),
          .busy(row_fsm_busy),
          .hsync(hsync),
          .row(current_row),
          .vol_coordinate(z_coordinate),
          .no_return(no_return),
          .AD_data(AD_data),
          .RAM_data(RAM_data),
          .RAM_addr(RAM_addr),
          .RAM__WE(RAM__WE),
          .RAM__OE(RAM__OE));

divider_fsm volume_divider (
          .clk(clk),
          ._reset(_reset),
          .start(start_volume_division),
          .busy(volume_division_busy),
          .dividend(z_total),
          .divisor(number_of_reflections),
          .quotient(volume_quotient));

divider_fsm pitch_divider (
          .clk(clk),
          ._reset(_reset),
          .start(start_pitch_division),
          .busy(pitch_division_busy),
          .dividend(x_total),
          .divisor(number_of_reflections),
          .quotient(pitch_quotient));


always @ (posedge clk or negedge _reset) begin
          if (!_reset)
                    state <= init;
          else
                    state <= next_state;

          start_row_fsm <= start_row_fsm_int;
          current_row <= current_row_int;
          busy <= busy_int;
          x_total <= x_total_int;
          z_total <= z_total_int;
          number_of_reflections <= number_of_reflections_int;
          hsync_count <= hsync_count_int;

          pitch <= pitch_int;
          volume <= volume_int;

          start_pitch_division <= start_pitch_division_int;
          start_volume_division <= start_volume_division_int;

end

always @ (state or vsync or hsync) begin
          start_row_fsm_int = start_row_fsm;
          current_row_int = current_row;
          busy_int = busy;
          x_total_int = x_total;
          z_total_int = z_total;
          number_of_reflections_int = number_of_reflections;
          hsync_count_int = hsync_count;

          start_pitch_division_int = start_pitch_division;
          start_volume_division_int = start_volume_division;

          case(state)
                    init: begin
                              if (enable) begin
```

```
                    next_state = discard_blank_rows_1;

                    busy_int = 1;
                    x_total_int = 0;
                    z_total_int = 0;
                    number_of_reflections_int = 0;
                    hsync_count_int = 0;
                    current_row_int = 0;
          end
          else begin
                    next_state = init;

                    busy_int = 0;
          end

          start_row_fsm_int = 0;

end

discard_blank_rows_1: begin
          if (hsync_count > 9) begin
                    next_state = wait_for_row_start_1;
          end
          else if (!hsync)
                    next_state = discard_blank_rows_2;
          else
                    next_state = discard_blank_rows_1;
end

discard_blank_rows_2: begin
          if (hsync) begin
                    next_state = discard_blank_rows_1;

                    hsync_count_int = hsync_count + 1;
          end
          else
                    next_state = discard_blank_rows_2;
end

wait_for_row_start_1: begin
          if (!hsync)
                    next_state = wait_for_row_start_2;
          else
                    next_state = wait_for_row_start_1;
end

wait_for_row_start_2: begin
          if (hsync) begin
                    next_state = wait_for_row_fsm_start;

                    start_row_fsm_int = 1;
          end
          else
                    next_state = wait_for_row_start_2;
end

wait_for_row_fsm_start: begin
          if (row_fsm_busy) begin
                    next_state = wait_for_row_fsm_complete;

                    start_row_fsm_int = 0;
          end
          else
                    next_state = wait_for_row_fsm_start;
end

wait_for_row_fsm_complete: begin
          if (!row_fsm_busy) begin

                    z_total_int = z_total + z_coordinate;
                    if (!no_return) begin
```

```
                                                x_total_int = x_total + current_row;
                                                number_of_reflections_int =
number_of_reflections + 1;
                                        end


                                        if ((current_row > 225) || (!vsync))
                                                next_state = begin_stats;
                                        else begin
                                                next_state = wait_for_row_start_1;

                                                current_row_int = current_row + 1;
                                        end
                                end
                                else
                                        next_state = wait_for_row_fsm_complete;
                        end

                        begin_stats: begin
                                next_state = wait_for_begin_stats;

                                start_volume_division_int = 1;
                                start_pitch_division_int = 1;
                        end

                        wait_for_begin_stats: begin
                                if ((volume_division_busy) && (pitch_division_busy))
                                        next_state = wait_for_complete_stats;
                                else
                                        next_state = wait_for_begin_stats;
                        end

                        wait_for_complete_stats: begin
                                if ((!volume_division_busy) && (!pitch_division_busy)) begin
                                        next_state = init;

                                        volume_int = volume_quotient[8:1];
                                        pitch_int = pitch_quotient[8:1];
                                end
                                else
                                        next_state = wait_for_complete_stats;
                        end

                endcase

end

endmodule
```

Fundamental Frequency Estimator

```
// Kushan Surana
// Module which computes the correlation function and outputs the result to the Ram (called write_ram from now on)

// 04/31/04

// Reads addresses from single RAM where the digital data is stored
// Computes the multiplication and performs the accumulation
// Compares the value of the autocorrelation function with the value before and the value after to find the local maxima
// Once the maxima is found, the fundmantal_freq is output using the (fs / formula write_address - 1)
// *** NEED TO WRITE MY OWN DIVISION MODULE -- ALTERA DOES NOT SUPPORT DIVISION
```

```
module
correlation_peak(clk,reset,start_correlation,accumulated_value,done_correlation,g_bar,enable_ram2,read_address,start_mult,start_acc
um,reset_accum,start_divide,done_divide,divisor,le_ram1,le_ram2,state,marker_address,reg0,reg1,reg2);


// System clock of 10 MHz....this is the input clock to the system
input clk;
// Global reset signal, assumes its synchronized
input reset;

// User interface
input start_correlation;

input [35:0] accumulated_value;

// Reading from the two RAM's storing the analog to digital converted data

output done_correlation;
reg done_correlation;


output g_bar;
reg g_bar;
reg g_bar_int;

output enable_ram2;
reg enable_ram2;
reg enable_ram2_int;


reg flag;

// Read addresses

output [9:0] read_address;

reg [9:0] read_address_int;
reg [9:0] read_address;

reg [9:0] read2_address;
reg [9:0] read2_address_int;

reg [9:0] read1_address;
reg [9:0] read1_address_int;

output start_mult;
output start_accum;
output reset_accum;

reg start_mult;
reg start_accum;

reg reset_accum;

output start_divide;
reg start_divide;
reg start_divide_int;
input done_divide;

output [9:0] divisor;
reg [9:0] divisor;
reg [9:0] divisor_int;

output le_ram1;
reg le_ram1;
reg le_ram1_int;

output le_ram2;
reg le_ram2;
reg le_ram2_int;
```

```
output [4:0] state;

// internal state

reg [4:0] state;
reg [4:0] nextstate;


// Marker address

output [9:0] marker_address;
reg [9:0] marker_address;
reg [9:0] marker_address_int;

output [35:0] reg0;
output [35:0] reg1;
output [35:0] reg2;

reg [35:0] reg0;
reg [35:0] reg1;
reg [35:0] reg2;

reg [35:0] reg0_int;
reg [35:0] reg1_int;
reg [35:0] reg2_int;

// State declarations

parameter INITIALIZE = 0;
parameter IDLE = 1;
parameter SET_READ1_ADDRESS_INITIAL = 2;
parameter SET_READ1_ADDRESS = 3;
parameter SETTLE_READ1_ADDRESS = 4;
parameter SET_GBAR_LOW = 5;
parameter WAIT_REG1_LOAD = 6;
parameter WAIT_REG1_LOAD_2 = 7;
parameter SET_READ2_ADDRESS = 8;
parameter SETTLE_READ2_ADDRESS = 9;
parameter SET_GBAR_LOW_2 = 10;
parameter WAIT_DATA_LOAD = 11;
parameter WAIT_DATA_LOAD_1 = 12;
parameter START_MULTIPLY = 13;
parameter START_ACCUMULATE = 14;
parameter CONTINUE_MULTIPLY = 15;
parameter WAIT_FIND_F0 = 16;
parameter FIND_F0 = 17;
parameter STARTING_DIVIDE = 18;
parameter WAIT_DONE_DIVIDE = 19;
parameter CHECK_WRITE_ADDRESS = 20;
parameter DONE = 21;


always @(posedge clk or posedge reset)
        begin
                if (reset) state <= INITIALIZE;
                else state <= nextstate;

                if (flag == 1)
                        begin
                                reg0[35] <= !accumulated_value[35];
                                reg0[34:0] <= accumulated_value[34:0];
                                reg1 <= reg0_int;
                                reg2 <= reg1_int;
                        end
                        else
                        begin
                                reg0 <= reg0_int;
                                reg1 <= reg1_int;
                                reg2 <= reg2_int;
```

```
                end

        divisor <= divisor_int;
        le_ram1 <= le_ram1_int;
        le_ram2 <= le_ram2_int;
        read_address <= read_address_int;
        read1_address <= read1_address_int;
        read2_address <= read2_address_int;
        marker_address <= marker_address_int;
        g_bar <= g_bar_int;
        enable_ram2 <= enable_ram2_int;
        start_divide <= start_divide_int;


    end

always @(state or start_correlation) begin

// defaults
        flag = 0;
        le_ram1_int = 0;
        le_ram2_int = 0;
        divisor_int = divisor;
        //divisor = 10'b1111111110;
        done_correlation = 0;
        read_address_int = read_address;
        read1_address_int = read1_address;
        read2_address_int = read2_address;
        marker_address_int = marker_address;
        enable_ram2_int = 1;
        g_bar_int = 1;
        start_mult = 0;
        start_accum = 0;
        reset_accum = 0;
        start_divide_int = 0;
        reg0_int = reg0;
        reg1_int = reg1;
        reg2_int = reg2;

case(state)

    INITIALIZE:
        begin
            divisor_int = 10'b0000000001;
            read_address_int = 10'b0000000000;
            read1_address_int = 10'b0000000000;
            read2_address_int = 10'b0000000000;
            marker_address_int = 10'b0000000000;

            reg0_int[35:0] = 36'h000000000;
            reg1_int[35:0] = 36'h000000000;
            reg2_int[35:0] = 36'h000000000;
            nextstate = IDLE;
        end

    IDLE:
        begin
            if (start_correlation)
            begin
            nextstate = SET_READ1_ADDRESS_INITIAL;
            end
            else nextstate = IDLE;
        end

    SET_READ1_ADDRESS_INITIAL:
        begin
            read1_address_int = marker_address;
            nextstate = SET_READ1_ADDRESS;
        end
```

```
SET_READ1_ADDRESS:
        begin
                read_address_int = read1_address;
                nextstate = SETTLE_READ1_ADDRESS;
        end

SETTLE_READ1_ADDRESS:
        begin
                nextstate = SET_GBAR_LOW;
        end

SET_GBAR_LOW:
                begin
                        enable_ram2_int = 0;
                        g_bar_int = 0;
                        nextstate = WAIT_REG1_LOAD;
                end

WAIT_REG1_LOAD:         begin
                        enable_ram2_int = 0;
                        g_bar_int = 0;
                        le_ram1_int = 1;
                        nextstate = WAIT_REG1_LOAD_2;
                end
WAIT_REG1_LOAD_2:       begin
                        le_ram1_int = 0;
                        nextstate = SET_READ2_ADDRESS;
                end

SET_READ2_ADDRESS:
        begin
                read_address_int = read2_address;
                nextstate = SETTLE_READ2_ADDRESS;
        end

SETTLE_READ2_ADDRESS:
        begin
        nextstate = SET_GBAR_LOW_2;
        end

SET_GBAR_LOW_2:
                begin
                enable_ram2_int = 0;
                g_bar_int = 0;
                nextstate = WAIT_DATA_LOAD;
                end

WAIT_DATA_LOAD:
                begin
                enable_ram2_int = 0;
                g_bar_int = 0;
                le_ram2_int = 1;
                nextstate = WAIT_DATA_LOAD_1;
                end

WAIT_DATA_LOAD_1:       begin
                        le_ram2_int = 0;
                        nextstate = START_MULTIPLY;
                 end

START_MULTIPLY:         begin
                start_mult = 1;
                nextstate = START_ACCUMULATE;
            end

START_ACCUMULATE:               begin
                        start_accum = 1;
                        nextstate = CONTINUE_MULTIPLY;
                  end
```

```
CONTINUE_MULTIPLY:
                        begin
                                if (read1_address == 10'b1111111111)
                                //if (read1_address == 10'h005)
                                        begin
                                        flag = 1;
                                        nextstate = WAIT_FIND_F0;
                                        end
                                else
                                        begin
                                        read1_address_int = read1_address + 1;
                                        read2_address_int = read2_address + 1;
                                        nextstate = SET_READ1_ADDRESS;
                                        end
                        end

WAIT_FIND_F0:   begin
                        nextstate = FIND_F0;
                end

FIND_F0:
                        begin
                                if (marker_address < 10'b0000000100)
                                // if (marker_address < 10'h011)
                                        nextstate = CHECK_WRITE_ADDRESS;
                                else
                                        begin


//                                                      if ((reg1[34] == 0) & (reg0[34] == 1) & (reg2[34] == 1))
//                                                              begin
//                                                              divisor = marker_address - 1;
//                                                              nextstate = STARTING_DIVIDE;
//
//                                                              end
//                                                      else if ((
//                                                              ((reg1[34] == 0) & (reg0[34] == 0) & (reg2[34] == 0)) |
((reg1[34] == 1) & (reg0[34] == 1) & (reg2[34] == 1))) & ((reg1[33:0] > reg0[33:0]) & (reg1[33:0] > reg2[33:0])))
//

                                                        if ((reg1[35:0] > reg0[35:0]) & (reg1[35:0] > reg2[35:0]))
                                                                begin
                                                                divisor_int = marker_address - 1;
                                                                nextstate = STARTING_DIVIDE;
                                                                end
                                                        else

                                                                nextstate = CHECK_WRITE_ADDRESS;
                                        end
                        end

STARTING_DIVIDE:                        begin
                                        start_divide_int = 1;
                                        nextstate = WAIT_DONE_DIVIDE;
                        end

WAIT_DONE_DIVIDE:                       begin
                                        if (done_divide)
                                                nextstate = DONE;
                                        else nextstate = WAIT_DONE_DIVIDE;
                         end

CHECK_WRITE_ADDRESS:
                        begin
                                if (marker_address == 10'b1111111110)
                                //if (marker_address == 10'h005)
                                        begin
                                        divisor_int = marker_address;
                                        //divisor = 10'b1111111110;
                                        nextstate = STARTING_DIVIDE;
```

```
                                        end
                                else
                                        begin
                                        marker_address_int = marker_address + 1;
                                        read2_address_int = 10'b0000000000;
                                        reset_accum = 1;
                                        nextstate = SET_READ1_ADDRESS_INITIAL;
                                        end
                        end


        DONE:           begin
                        reset_accum = 1;
                        read2_address_int = 10'b0000000000;
                        marker_address_int = 10'b0000000000;
                        reg0_int[35:0] = 36'h000000000;
                        reg1_int[35:0] = 36'h000000000;
                        reg2_int[35:0] = 36'h000000000;
                        divisor_int = 10'b0000000001;
                        done_correlation = 1;
                        nextstate = IDLE;
                        end

        default: nextstate = IDLE;

endcase

end

endmodule
```

Pitch Shifter

```
module majorfsm
  …

parameter IDLE = 0;
parameter WAITSAMPLE = 1;
parameter STARTRESAMPLE = 2;
parameter WAITRESAMPLE =3;
parameter STARTAD = 4;
parameter WAITADFINISH = 5;
parameter CHECKCOUNT = 6;
parameter INCREMENTADDR = 7;

adwrite myadwrite(….)
resampler myresampler …);

//tristate buffer
assign ext_data = data_oen ? out_reg : 8'hz;

always @ (posedge clk) begin

   if (reset) begin state <= IDLE;
      buffer <= 0;
      count <= 0;
      data_oen <= 0;  //begin tristated
     end
   else begin
   if (ad_tristate | resampler_tristate)
      data_oen <= 0;
   else if (resampler_drive)
      data_oen <= 1;
   //keeps whatever value

      state <= next;
```

```
      buffer <= buffer_int;
      count <= count_int;

   if (get_resampled)
      resampled_reg <= ext_data;
   if (get_original)
      original_reg <= ext_data;
   if (out_reg_enable)
      out_reg <= sample_out;
   status_sync <= status;

   end

end

always @ (state or start or sample or resampler_state or ad_state or count) begin
ad_start = 0;
resampler_start = 0;
data_oen_int = 0; //always low for now, keeps bus high impedence.
increment_addr = 0;
buffer_int = buffer;
count_int = count;

case (state)
  IDLE: begin
    if (start) begin
      next = WAITSAMPLE;
      increment_addr = 1; // after a reset, this initializes all the address.
      end
    else
    next=IDLE;
   end
  //0
  WAITSAMPLE: begin
     if (sample) begin
        resampler_start = 1;
        next = STARTRESAMPLE;
     end
     else next = WAITSAMPLE;
  end
  //1
  STARTRESAMPLE: begin
     next = WAITRESAMPLE;
  end
  //2
  WAITRESAMPLE: begin
     if (resampler_state == IDLE) begin
        ad_start = 1;
        next = STARTAD;
      end
     else next = WAITRESAMPLE;
  end
  //3
  STARTAD: begin
    next = WAITADFINISH;
  end
  //4
  WAITADFINISH: begin
     if (ad_state == IDLE)
        next = CHECKCOUNT;
     else next = WAITADFINISH;
  end
  //5
  CHECKCOUNT: begin
     if (count == BUFFERSIZE) begin
        buffer_int = ~buffer;
        count_int = 0;
        next = INCREMENTADDR;
      end
     else begin
        count_int = count + 1;
```

```
        next = INCREMENTADDR;
      end
  end
  //6
  INCREMENTADDR: begin
     increment_addr = 1;
     next = WAITSAMPLE;
  end
  //7
  endcase
end //always

endmodule


module AddrMan(…);

// this module receives requests to output the address for SRAM. AD670 has one address
register, resampler has two.
// both can turn off the address, which means disabling the sram.
// increment_addr comes from the major fsm, signalling AddrMan to update minor fsm
addresses with  buffer, count
// and input_rate.

input clk;
input reset;
output [15:0] addr; //addresses are {enable line, 8 bits}
reg [15:0] addr;

//for address calculation
input increment_addr, buffer;
input [11:0] count;   //should be size of buffer
input [10:0] input_rate;  //from rate_converter is 11-bits wide 5 integer, 6 fractional
reg [17:0] rate;    //rate is 18-bits: 12 integer, 6 fractional

//interface to AD
input AD_addr_req, AD_addr_off;
reg [12:0] AD_addr;

//interface to resampler
input resampler_addr_reqr, resampler_addr_reqo, resampler_addr_off;
reg [17:0] resampler_addr_r;
reg [12:0] resampler_addr_o;
reg [17:0] resampler_sum_r;
parameter OFF = 16'b1110000000000000;  //default is disabled sram, address 0
parameter S1 = 3'b011;

always @ (resampler_addr_r or rate) begin
  resampler_sum_r = resampler_addr_r + rate;
end

always @ (posedge clk) begin
   if (reset)
      addr <= OFF;  //default is disabled sram, address 0
   else if (AD_addr_req)
      addr <= {S1, AD_addr};
   else if (AD_addr_off)
      addr <= OFF;
   else if (resampler_addr_reqr)
      addr <= {S1, ~buffer, resampler_addr_r[17:6]};
   else if (resampler_addr_reqo)
      addr <= {S1, resampler_addr_o};
   else if (resampler_addr_off)
      addr <= OFF;
   else if (increment_addr) begin
      AD_addr <= {buffer, count};
      resampler_addr_o <= {~buffer, count};
      resampler_addr_r <= resampler_sum_r;
      end
   else begin
      addr <= addr;
```

```
      rate <= input_rate;
    end
end

endmodule

*************
module rateconverter(clk, reset, calc_pulse, rate, f1, f0,
                     bigf1,
                     bigf0,
                     state);
…
reg output_enable, load_reg;
parameter WAITPULSE = 0;
parameter WAITLOAD = 1;
parameter BEGIN = 2;
parameter INCREMENT = 3;
parameter OUTPUTDATA = 4;
always @ (posedge clk) begin

   if (reset) begin
      state <= WAITPULSE;
      count <= 0;
       //add 7 bits of fractional part
       //put 7 bits in front
      end
   else begin
     state <= next;
     bigf1 <= bigf1_int;
     count <= count_int;
   if (output_enable)
      rate <= count;
   if (load_reg) begin
      bigf1 <= {f1, 6'b0};
      bigf0 <= {6'b0, f0};
    end
    end

end


always @ (state) begin
count_int = count;
output_enable = 0;
bigf1_int = bigf1;
load_reg = 0;

case(state)
    WAITPULSE: begin
       if (calc_pulse) begin
         count_int = 0;
         load_reg = 1;
         next = WAITLOAD;
       end
       else next = WAITPULSE;
    end
    //0

    WAITLOAD: begin
       next = BEGIN;
    end
    //1
    BEGIN: begin
       if (bigf0 >= bigf1-bigf0)
          next = OUTPUTDATA;
       else begin
          bigf1_int = bigf1 - bigf0;
          count_int = count + 1;
          next = INCREMENT;
        end
    end
```

```
    //2
    INCREMENT: begin
       next = BEGIN;
    end
    //3
    OUTPUTDATA: begin
       output_enable = 1;
       next = WAITPULSE;
    end
//4
  endcase

end
endmodule

States from Resampler FSM
case (state)
  IDLE: begin
    if (start) begin  //if received start signal from mfsm
      resampler_tristate = 1;
      next = OPENBUS;
      end
    else
      next = IDLE;
  end
  //0
  OPENBUS: begin
     SRAM_R = 1;
     resampler_addr_reqr = 1;
     next = LOADADDR;
  end
  //1
  LOADADDR: begin
     get_resampled = 1;
     next = READDATA1;
  end
  //2
  READDATA1: begin
      if (summer_sync) begin
         resampler_addr_reqo = 1;
         next = LOADORIGINALADDR;
      end
      else begin
         resampler_addr_off =1 ;
         load_resampled = 1;
         next = WAITSUM;
      end
  end
  //3
  LOADORIGINALADDR: begin
      next = WAITSTABLE;
  end
  //4
  WAITSTABLE: begin
     get_original = 1;
     next = READDATA2;
  end
  //5
  READDATA2: begin
     resampler_addr_off = 1;
     load_sum = 1;
     next = WAITSUM;
  end
  //6
  WAITSUM: begin
     resampler_drive = 1;
     next = DRIVEDATA;
  end
  //7
  DRIVEDATA: begin
     ad558_start = 1;
```

```
   next = WAIT;
end
//8
WAIT: begin
   next = WAITFINISH;
end
//9
WAITFINISH: begin
   if (ad558_state == IDLE)
      next = IDLE;
   else
      next = WAITFINISH;
end
//10
```