mp3 player

Alma Rico and Amy Wibowo

May 13, 2005

TA Charlie Kehoe

Class 6.111

Abstract Have you wondered exactly how your mp3 player accomplishes the tasks of downloading music onto your mp3 player and the playing of the music itself? We did, and that's why for our 6.11 final project, we wanted to make our own. It's a cool device, it's useful, and additionally, it makes for a good project because it modularizes itself quite nicely into well-defined subtask units. We wanted to implement the basic functionalities of an MP3 player: stop, play, pause, select song, and display playlist. An MP3 decoder was used to translate songs from MP3 format into digital sound information and a D/A converter to change this information into an analog signal to be fed into speakers. The results were not as we anticipated, but in this report, we discuss our design, test attempts, and the difficulties we had.

1 Overview of System

Mp3, which stands for MPEG audio layer 3, is a compression scheme for audio files. It is is a common format to store songs in, since it takes up little memory. And after decoding the file back into audio signals, the song can be played through speakers with relatively high quality. For our final project we wanted to create a device that stored mp3 files and decoded them back into audio files. The user interface would include ability to view song list and select song for play, along with play, pause, and stop buttons.

Mp3 decoding is a complicated process, so we chose to facilitate the decoding process and instead focus on the user interface by using an STA013 MP3 decoding chip. The audio file is stored as the contents of a rom. The output of the mp3 chip is fed to speakers, so the user can hear the song. The song list is displayed on a 640 x 480 vga monitor, or a 1 line, 16 character led display. The song information can be found in the last 128 bytes the mp3 file. The title information specifically is retrieved from the 4th through 33rd of these bytes. The ascii codes are stored into an SRAM, which is used by a video controller module to paint the picture on the display screen. Up and down buttons change the song that is selected (indicated by reverse video in the case of the 640 x 480 vga monitor, or by being in the viewing frame of the 1-line led display), and the song which is playing is indicated by a playing arrow next to its title.

2 Description of System

The system was divided into an audio component and a video component.

2.1 Audio Component (by Alma Rico)

MP3 Audio Compression MP3 formatting reduces the amount of bytes that make up a song through compression. A perceptual noise shaping scheme is used to convert audio data into MP3 formatted music to prevent hurting the quality of the music. By taking into account what the human ear can hear, the compression algorithm can reduce the size of a CD-quality song by a factor of 10. For example, MP3 formatting eliminates sounds that the human ear cant detect. This type of music compression allows you to download music faster and store more songs in your hard disk without taking up too much space (Brain, Marshall. How MP3 Files Work.).

The MP3 audio format is composed of various parts called frames, each containing a header and audio information. The header information contains the MPEG audio version ID, the layer description, the bit rate index, and sampling rate frequency index, and other data relating to the application. Meanwhile, the frame contains the actual audio data. At the end of the audio data is an ID tag that contains information describing the audio file (Supurovic, Predrag. MPEG Audio Compression Basics.)

STA013 MP3 Decoder Chip By extracting the bit rate information from audio data, the 3V chip can decode an MP3 data stream into digital audio data. Also, through careful configuration the decoder can send the control signals to a digital to analog converter.

IC Communication Protocol Since the STA013 has a wide range of functionalities, the chip must be configured for a specific application using Philips Semiconductor IC communication

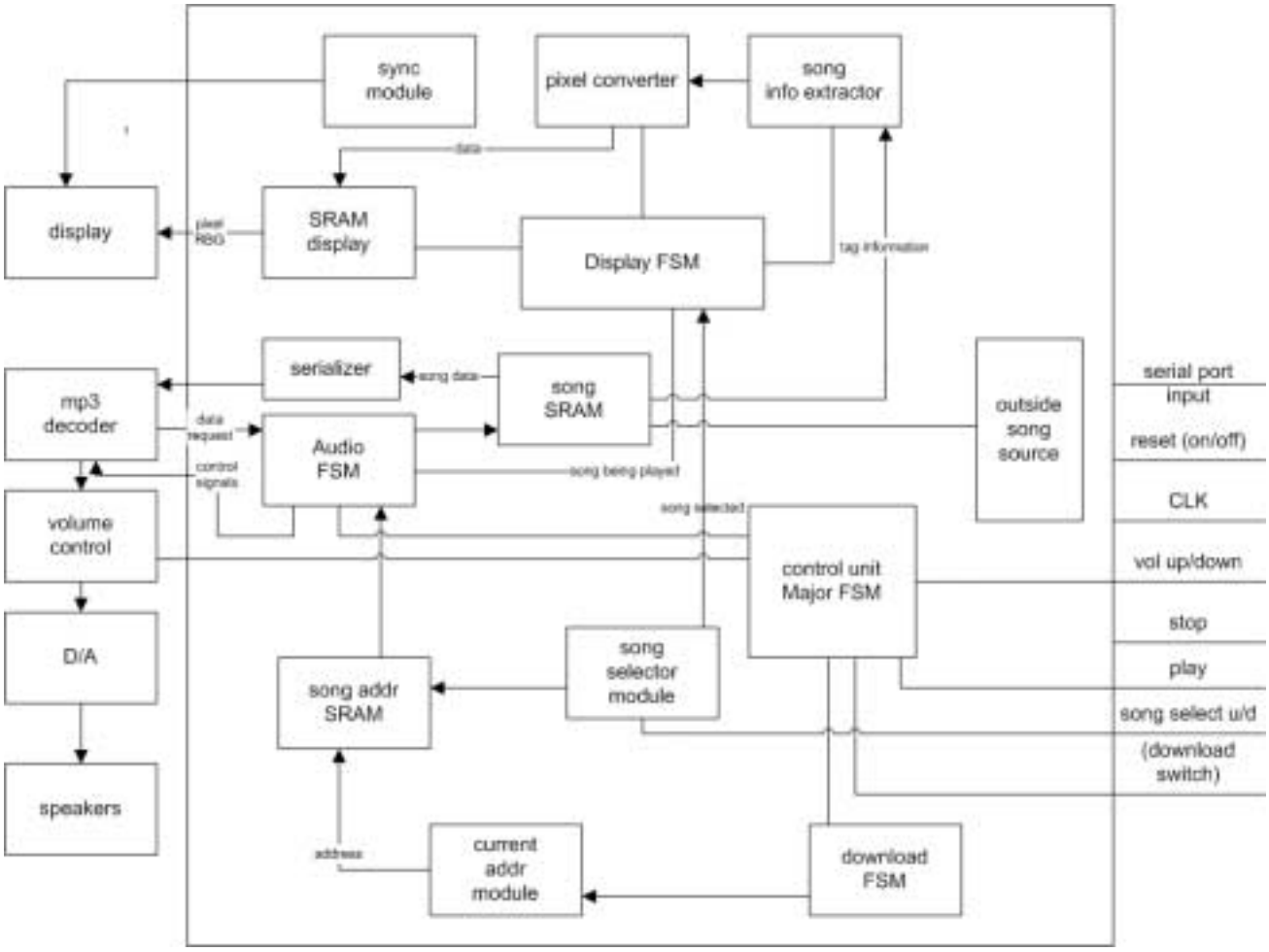

Figure 1: General block diagram of system.

protocol. In addition, ST requires that the decoder chip be initialized prior to use. They provide a file with 2007 internal register addresses and data. The need for this initial configuration is never explained. The communication bus protocol consists of two lines: SDA (data) and SCL (clock). IC is designed for multiple master-slave interactions, but in this case only the FPGA acts as the master and the chip as the slave.

The main operations used to communicate with the MP3 decoder are: start condition, stop condition, send a byte, and receive a byte. These four steps compose the read and write functions. Both the SDA and SCL lines should be idle prior to communication, where both are set to high. A master initializes communication with a start condition and ends with a stop condition. A start condition consists of setting the SDA line low while SCL is high, while a stop condition is a change from low to high of the SDA line while SCL is high. Other than the start and stop conditions, SDA should never change during the high period of the SCL. Since the SCL line behaves as a clock during the master-slave communication, it dictates the time at which bits are sent. Whether the byte is sent from master to slave or slave to master, the SDA line sends one bit every low clock period for eight clock periods. During a ninth cycle, the device receiving the data sends an acknowledge signal on the SDA line by setting the line low. The figure below shows the behavior

of the four basic operations.

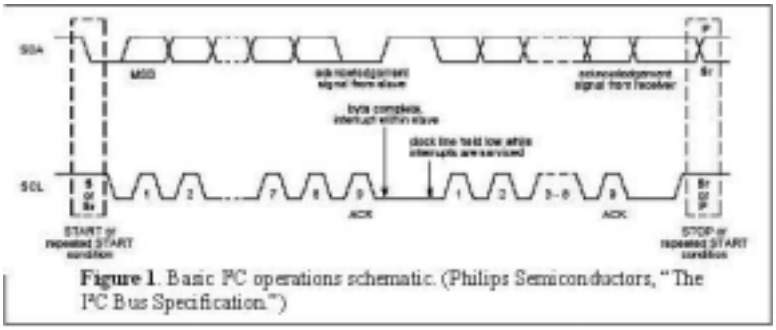

Figure 1. Basic I²C operations schematic. (Philips Semiconductors, "The I²C Bus Specification.")

Figure 2: Basic IC operations schematic. (Philips Semiconductors, "The IC Bus Specification.")

Tri-state buffer and Mux

The operations described above are combined to create read and write functions. The FPGA can read from the MP3 decoder by following these steps:

(1) Start Condition

(2) Send 7-bit Device Address plus R/Wbar bit set to 0

(3) Send Sub-Address Byte

(4) Stop Condition

(5) Start Condition

(6) Send 7-bit Device Address plus R/Wbar bit set to 1

(7) Receive a Byte

(8) Stop Condition

The STA013 device address assigned by Philip'ss Semiconductors is 0x86. The least significant bit during a send a byte operation is used to indicate a read or a write. A logical 1 indicates a read while a 0 indicates a write. Likewise, the write function consists of these steps:

(1) Start Condition

(2) Send 7-bit Device Address plus R/Wbar bit set to 0

(3) Send value of Address to write byte

(4) Send value of Data to write byte

(5) Stop Condition

All devices are required to acknowledge after receiving a byte, so the ACK bit should be low after every byte sent. Table 1 and Figure 2 below show the timing requirements for Standard-Mode IC communication. Although the specifications were not included in the STA013 data sheet, the timing requirements must be considered to achieve the desired behavior.

Table 1. I2C timing requirements

PARAMETER	SYMBOL	STANDARD-MODE	
		MIN.	MAX.
SCL clock frequency	f_{SCL}	0	100
Hold time (repeated) START condition. After this period, the first clock pulse is generated	$t_{HD;STA}$	4.0	–
LOW period of the SCL clock	t_{LOW}	4.7	–
HIGH period of the SCL clock	t_{HIGH}	4.0	–
Set-up time for a repeated START condition	$t_{SU;STA}$	4.7	–
Data hold time: for CBUS compatible masters (see NOTE, Section 10.1.3) for I²C-bus devices	$t_{HD;DAT}$	 5.0 (02)	 – 3.45(3)
Data set-up time	$t_{SU;DAT}$	250	–
Rise time of both SDA and SCL signals	t_r	–	1000
Fall time of both SDA and SCL signals	t_f	–	300
Set-up time for STOP condition	$t_{SU;STO}$	4.0	–
Bus free time between a STOP and START condition	t_{BUF}	4.7	–
Capacitive load for each bus line	C_b	–	400
Noise margin at the LOW level for each connected device (including hysteresis)	V_{nL}	$0.1V_{DD}$	–
Noise margin at the HIGH level for each connected device (including hysteresis)	V_{nH}	$0.2V_{DD}$	–

(Philips Semiconductors, "The I²C Bus Specification.")

Figure 3: IC timing requirements. (Philips Semiconductors, "The IC Bus Specification")

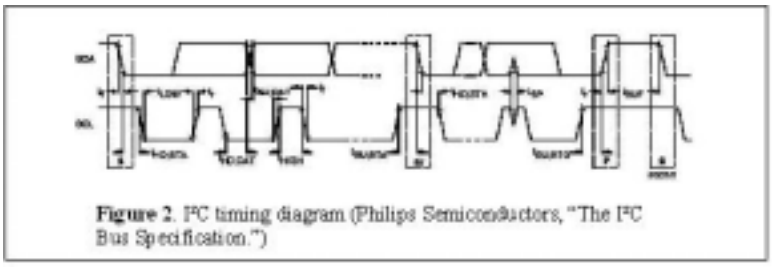

Figure 2. I²C timing diagram (Philips Semiconductors, "The I²C Bus Specification.")

Figure 4: IC timing diagram (Philips Semiconductors, "The IC Bus Specification")

Data Transfer The decoder chip uses three lines for MP3 input: SDI (data), SCKR (clock), and DATA-REQ (ready). Only when DATA-REQ is high, can the data source send MP3 files, MSB first, through the Serial Data Input (SDI) line. The SCKR (Receiver Serial Clock) dictates when each bit of the data is sent, and its limited to a frequency of less than 20Mbit/sec. A BIT-EN pin can also be used to control when data is ignored, but it was simply wired to high for this project.

Digital to Analog Converter Interface The STA013 can be configured to drive a DAC using serial PCM format. The signals Serial Data Output (SDO), and Serial Clock Output (SCKT), and Left/Right Channel Selection Clock (LRCLK) can be connected directly to the DAC. Since the CS4334 DAC is one of the chips that can be easily configured into the decoder chip, we decided to eliminate any complexity that might arise by using the AD558 by using the recommended DAC. The CS4334 chip also eliminates the need for any filtering and accepts a 24-bit digital input. Figure 3 below shows the circuit diagram for the STA013 and CS4334 chip, the only addition made was a pull-up resistor on the SDA line as will be explained later.

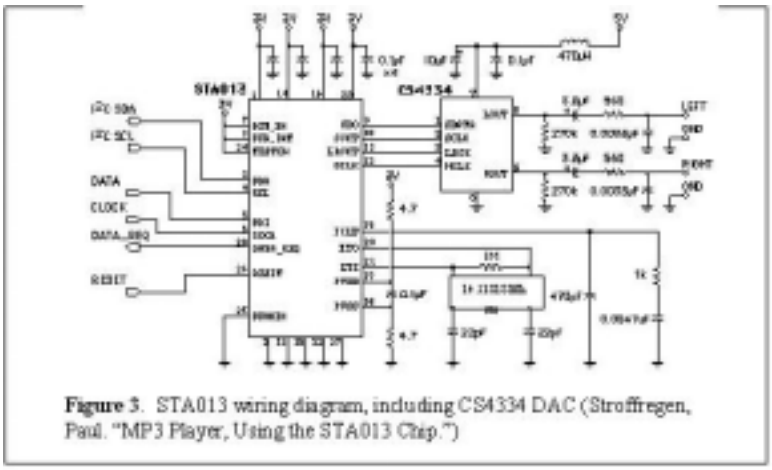

Figure 3. STA013 wiring diagram, including CS4334 DAC (Stroffregen, Paul. "MP3 Player, Using the STA013 Chip.")

Figure 5: IC timing diagram (Philips Semiconductors, "The IC Bus Specification")

Crystal Oscillator and System Clock As shown in the figure, the chip uses a crystal oscillator in addition to the SCL and SCKR clocks. Many crystal frequencies can be used, but the suggested three are 14.31818, 14.7456, and 10 MHz. The first oscillator was chosen for this design. To clarify, the crystal oscillator is used to set the internal clock of the decoder, the SCL clock is used for the IC communication, while the SCKR clock is used for the data transfer and creation of the DAC clock. Since the data rate limit is 20Mbit/sec, the oscillator was also used for the SCKR clock. On the other hand, the SCL clock is limited to 100kbits/sec under the Standard protocol, so we chose to use a 80kbit/sec SCL. Since the FSMs are timed by a clock, the clock must be assigned accordingly. In our case, the SCL clock is four times slower, as will be explained shortly.

Checking and Configuration of STA013 Before initiating any data transfers, the MP3 decoder is first checked and then configured with the configuration file and the proper application-specific values for the DAC and chosen clock.

Check FSM The decoder chip can be checked and configured using the read and write IC functions. Certain registers within the chip are labeled as read only registers, such as the IDENT register containing the special chip identity. Verifying that the chip returns the correct value not only checks that the decoder is there, but it also shows that its functioning properly. For the STA013, the address of the register read is 0x01 and the identity value it should return is 0xAC.

In order to test the chip behavior, a checkFSM was instantiated to cycle through the read and write procedures. The FSM holds the device address parameter listen (0x86), the sub-address to be

read addr (0x01), and the data that the chip should return data (0xAC). There are about 40 states in the FSM, and there are about 5 for each of the basic operations necessary to read. For example, to write the device address the FSM cycles through the LISTEN-BYTE states to set the proper serial bit by shifting the listen parameter. To make sure that the byte is shifted precisely eight times, a counter is used to keep track of the eight SCL cycles. To ensure that the SDA line only writes while SCL is low, the states are written so that the SCL period is four times that of the input clock. The other states where bytes are sent or received have a similar behavior. In order to keep track of when the SDA line must be tri-stated to receive the acknowledge signal from the decoder, a check-cond parameter is assigned to 1 when the program expects a response. However, one must be careful in choosing the proper states. Since the SDA and SCL signals are control signals to the device, they are registered and are therefore a clock cycle behind when they are initially assigned. So the state following the last ACK state should also be tri-stated. The state transition diagram below shows the reading behavior as specified by the read call.

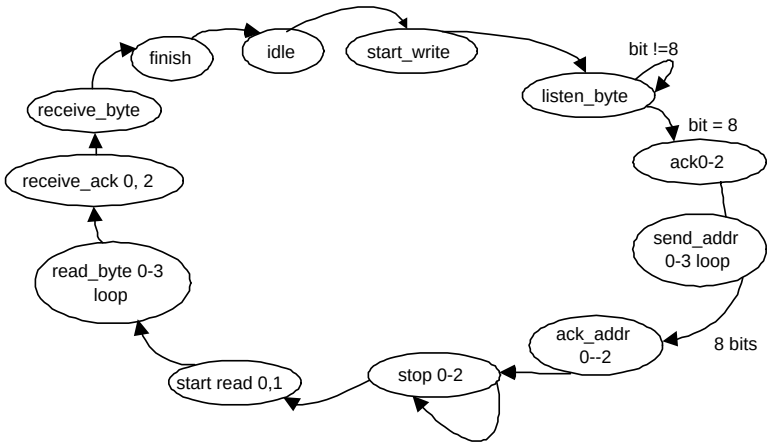

Figure 6: Check FSM state transition diagram

Configuration FSM After successfully reading from the identity register, one would proceed to write the total 2015 values to register addresses. In addition to the 2007 values in the configuration file, the values for the DAC and clock are set, and the chip is set to RUN and PLAY by writing to two more registers. The writing procedure is a little simpler than the reading, since it doesnt require a start or a stop midway through the call or multiple device writes. Similar to the check FSM the numerous states can be bunched into basic operations. The states can be grouped into LISTEN-BYTE, SEND-ADDR-BYTE, and SEND-DATA-BYTE, each followed with appropriate ACK states. The FSM cycles through the entire set of states for each value, until the count has reached the last PLAY write. Upon completing the configuration, the decoder should assert the DATA-REQ line. The state transition diagram depicts the FSM behavior and writing function. The configuration FSM also keeps track of a configuration condition that tri-states the SDA bus.

Address and Data ROMs The file provided by ST came in a bin format with two main rows, one for the address and the other for the corresponding write value. To simplify reading from memory, instead of saving the address and then the data right after, we chose to use two separate ROMs. The matched values are located at identical addresses on both ROMs. Using CoreGen, the configuration data was loaded in Excel and separated into separate columns and finally using the memory editing feature .coe files containing the information were created. The .coe file is used to

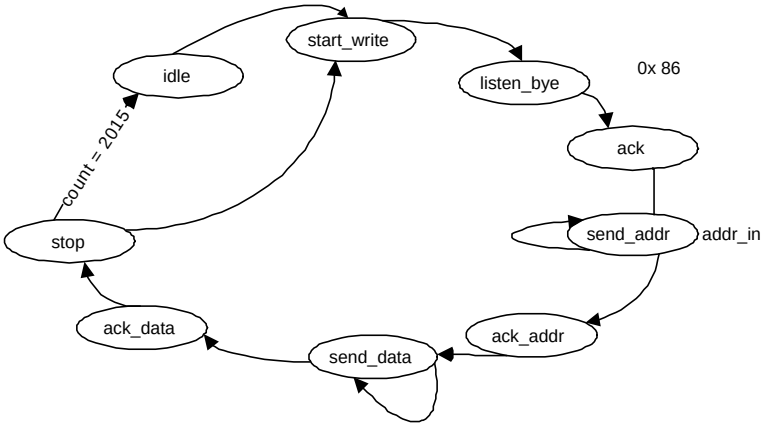

Figure 7: Configuration FSM state transition diagram

load the initial values of the memories. The configuration FSM can load both the address and data values at the same time during the LISTEN-BYTE4 state with enough time to allow the data to be stable.

Init FSM Dictating the actions of the check FSM and the configuration FSM is the init FSM. After a reset it enters a BEGIN state, which transitions into the CHECK state. Here, the FSM starts the minor check FSM and waits until the done signal is asserted. When checking is completed, the WAIT-CTRL state assures that the SDA and SCL lines are both high (once again since they are registered they are one clock cycle behind).The CONFIG state starts the configuration loop in the config FSM. After that, the initialization FSM goes into an IDLE state. The simple state transition diagram depicts the described behavior.

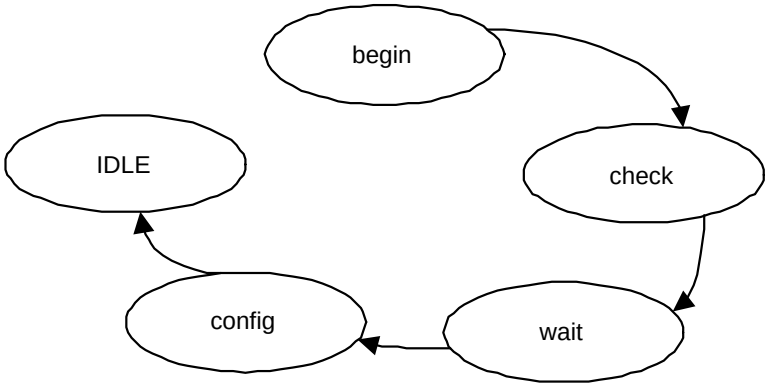

Figure 8: Initialize FSM state transition diagram

Tri-state buffer and Mux Since both the configuration and check FSMs use the SDA and SCL lines, the top level module uses a mux to decide which value to send. During the time that start-config is asserted, values from the configuration FSM are loaded, otherwise the check values are used. Moreover, the SCL line is not bidirectional and whatever value is returned by the mux can be sent to the decoder. However, the SDA line is carefully tri-stated. The output of the mux is sent to the tri-state buffer, which is only enabled when the check and config-cond signals are both

low.

Audio After completing the STA013 configuration, the chip should assert its DATA-REQ line allowing the program to send it MP3 data. The song that could be played could be sent in various ways, but we chose to instantiate a ROM. The data is sent serially to the decoder as dictated by the SCKR line.

Dido ROM Before playing Dido's "White Flag" the music information had to be stored in memory. As mentioned before, the initial file loaded onto a ROM must be in .coe format. As a result, the MP3 file had to be changed to hex, binary, or decimal. To perform this conversion, a ProXoft Binary Viewing tool was used to translate the data into columns of binary information including a corresponding address. The address served as a reference but was unnecessary for our application. Therefore, we had to eliminate it by deleting the column in Excel (since the file surpassed the maximum capacity for Excel the song had to be cut into several pieces). Since the .coe file has to be a long list of terms, we then wrote a short Perl program to change spaces into commas followed by newline characters. Then one could indicate the memory radix and replace the last comma by a semicolon. The perl script and a screen capture of the binary viewing tool can be found in the appendix.

Audio FSM To control when data is sent to the MP3 decoder, an audio FSM was used. Using play, stop, and DATA-REQ signals the FSM can transition through its states: IDLE, PLAY, STOP, PAUSE, and WAIT. During the IDLE state, no data is sent to the decoder and the system is awaiting a play signal. Once play is asserted, the next state PLAY can be cycled through to send data to the decoder unless play is set high again or stop is pushed. When a song is playing and play is hit, a pause will take effect, so that data is held until play or stop are asserted. Otherwise, when a song is playing and stop is chosen, then playback stops completely. Until play is high again will the chosen song be played from the beginning. The WAIT state is used when the DATA-REQ signal is low, and the MP3 decoder buffer is full. Since the play and stop signals are input through buttons, the user can push them down for arbitrary amounts of time. Therefore, a level to pulse converter was placed between he signals and the FSM to regulate the times they are asserted. The behavior of the audio system is shown in the state transition diagram.

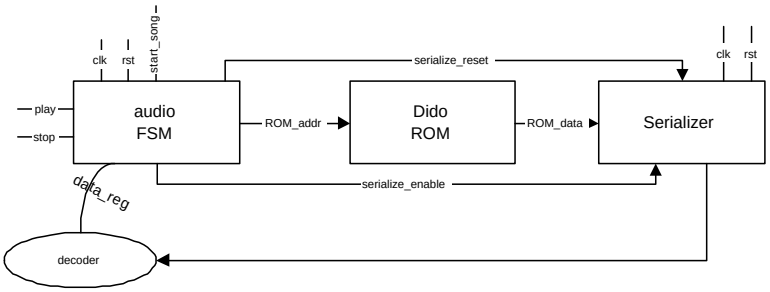

Figure 9: Audio FSM state transition diagram

Serializer To prevent the use of over abundant states as in the configuration and check FSMs the Audio FSM controlled a serializer when sending data to the chip. Using serialize-enable and serialize-reset, the FSM can control the information that is decoded during a pause, play, or stop. Resetting the serializer causes it to lose any registered values, which is useful during a stop. Without

the enable signal, the serialized data is held until the FSM leaves the WAIT or PAUSE states. The overall audio system interconnection is shown below.

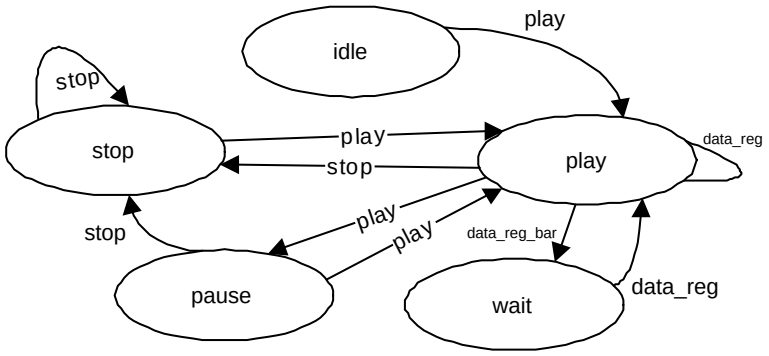

Figure 10: Block diagram of Audio FSM

2.2 Video Component (by Amy Wibowo)

Two attempts were made at generating a display for the mp3 player songlist. Both of them followed the same general flow from title data, to pixel information, to display, but were slightly different in details.

VGA Display The VGA display mode chosen was a 640 by 480 pixel mode with a pixel clock of 25.175 Mhz. This setting needed the following timing values to work:

Format	Pixel Clock (MHz)	Horizontal (in Pixels)				Vertical (in Lines)			
		Active Video	Front Porch	Sync Pulse	Back Porch	Active Video	Front Porch	Sync Pulse	Back Porch
640x480, 60Hz	25.175	640	16	96	48	480	11	2	31

Figure 11: Timing Parameters for a 640 by 480 VGA display at a 25.1 MHz pixel clock.

Pixel Clock The 25.175 Mhz pixel clock is generated by dividing the built in 25.175 Mhz clock by 30 and multiplying it by 28.

Sync Generator Module At the core of the VGA display is the Sync Generator Module. It produces the sync and blanking signals to be sent to the VGA display to tell it to repaint the screen. A horizontal sync signal occurs after each horizontal line of pixels has been completed. A vertical sync signal occurs after all horizontal lines of pixels have been painted. Blanking signals are sent during the front and back porch times. It has an internal counter that counts based on saved timing paramters the appropriate number of pixel clock ticks that each signal is supposed to last, and changes the signals when appropriate. The outputs are vga-out-sync-b, vga-out-blank-b, vga-out-hsync, vga-out-vsync. The labkit interfaces with a DAC and sends these signals to the VGA monitor. The sync generator also outputs the current horizontal pixel count and vertical line count, to be used to determine the current appropriate pixel color to be fed to the display.

Display SRAM The color information for the pixels painted in the sync-generator module is stored in an SRAM. It is 128 bits wide, and 13100 lines long. Each line contains 4 bits of information for 32 pixels. The four bits control the amount of red (which could easily be replaced by any or all colors) fed to the display (each bit controlling 1 bit of the 8 bit vga-out-red). It takes 25 lines of SRAM to display 1 row on the actual screen. This is a total of 800 pixels per line of display, because it includes the blanking and sync periods, where the color information ends up getting disregarded. Similarly, the SRAM holds enough information for 524 horizontal lines of pixels, the last 44 of which are ignored due to the sync and blanking signals but are included because the pixel count and vertical line count still increment during those times. The SRAM was generated by Coregen and initialized to 0.

Character Writer The chracter writer module takes in an ascii code, a display line position, and character position, and writes the appropriate pixel information in the SRAM for this character to be displayed on the screen. It does this by converting the ascii code into pixels with a rom, and shifting them so that they are at the appropriate position. Each character is 8 pixels wide and twelve lines high. For vertical spacing, however, characters are written so that the top row of pixels aligns with every 20th row of the display. So, the character module writer converts the line position and character position number into an SRAM address number, and the character position number into a shift amount within the SRAM line. 12 separate lines of SRAM must be written to, for the 12 lines that the character spans vertically. After shifting the pixel bits, they must be added to the existing contents of the SRAM at the same address, and this is the information that finally gets written.

Title Writer An mp3 title can only hold 30 characters of information, so it is safe to allow the Title Writer to write 30 characters to a given line of screen. This is done by having as input to the Charcter Writer module the 30 characters stored in a line of the title SRAM. The title writer takes in a write signal, and when high, signals the character writer to write each character to the SRAM, incrementing the horizontal count each time from the original line number. The orignal line number is fed into the title writer also, and is simply the number of the song in the order of the song list.

Title SRAM The title SRAM holds the title information. It was initialized pre-runtime, and had no connection to the audio portion, because the audio portion only held one song. The songs are indexed by their order in the play queue. Each line contains 1 character of the title as a 16 bit number which corresponds to an ascii code. It can store 10 titles, which contain 30 characters each, for a total of 300 lines in the SRAM.

LED Display The LED display is built into the new 6.111 labkit. It contains 640 leds, arranged into 16 groups of 40, which are in turn arranged into blocks of 5 leds horizontally and 8 (only the 7 top light up) leds. Each of the blocks can be used to represent one character. Whether each led is lit or unlit is determined by the 640 bit string named dots. The 40 most significant bits dots[639:600] represent the left most block. The first 8 most significant bits per group represent the leftmost vertical line of each block. The most significant bit per group of 8 pixels represents the lowermost led on that vertical line. Generating the sync signals for the display, plus the string dots, are sufficient to generate the LED display.

In the mp3 player specifically, the one line of LED display is used to display a single title of the playlist. The first block dislpays either a play symbol, a pause symbol, or neither if the song displayed is not the song in the play queue.

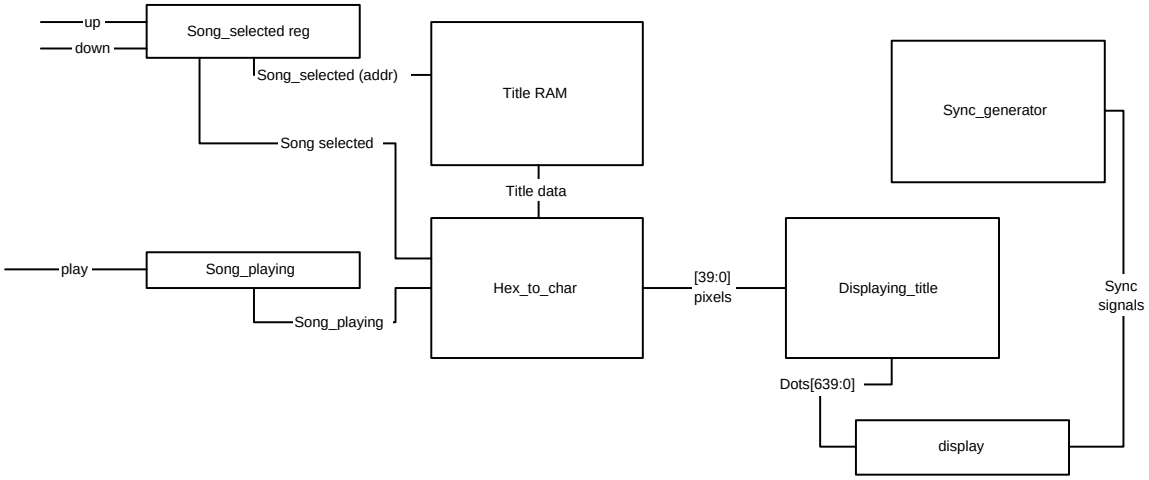

Figure 12: Block diagram of LED display module.

Sync Generator Module At the core of the VGA display is the Sync Generator Module. It produces the sync and blanking signals to be sent to the VGA display to tell it to repaint the screen. (Talk about the signals a bit). It takes the pixel information to be displayed on the LEDs as a 640 bit string called dots.

Title to LED's FSM Module The LED display can only show one line of text at a time. The Title to LED's Module takes in a number indicating the title that should be displayed. It is also connected to the memory module where the character information of the title is stored and the module that converts this information one character at a time from ascii hex digits into strings of 1's and 0's to be fed into the LED display. The first character on the LED display indicates if the current song is currently being played, is paused, or neither. There is a conversion state for each of the other 15 characters displayed, and a waiting state after each one, to make sure that the addresses and outputs of other modules accessed are valid. Once the module goes through all 15 states and has converted all the characters into pixels, an internal done signal is flagged and the 640 bits to be fed into the sync generator module are registered as outputs.

Ascii to Character module The ascii character module takes in 2 hex digits and converts them to a 40 bit string to be included in the 640 bit string LED input. These 40 bits represent the LEDs for the a single character. It is basically a look up table that matches the ascii code to the appropriate alphanumeric symbol. The module can recognize numbers, uppercase letters, and lowercase letters, but converts all lower case letters to uppercase letters, and replaces any other characters besides numbers and letters with spaces.

Title SRAM The title SRAM holds the title information. There was no difference between the VGA and the LED Title SRAM, except the size. The LED Title SRAM can store 10 titles, which contain 15 characters each, for a total of 150 lines in the SRAM.

Interface to Buttons The top level included the play, up, and down buttons, with a song-selected register, and a song-playing register. The up, down, and play buttons were connected to level to pulse converters. Up signals increased the value of song-selected, and down signals decreased the value. The value of song-selected was fed to the Title SRAM so that the up and down buttons controlled which song title was in display.

3 Testing and Debugging

3.1 Audio Module

Before programming the FPGA, all modules were first simulated on ModelSim to verify that the expected behavior was viewed. The only difficulty was that all the blocks depend on feedback from the decoder chip, so the complete behavior could only be viewed with the logic analyzer. Also, the entire configuration state cannot possibly all be checked for correctness. Instead, we made sure the first few initial and final values were right.

After a successful simulation, the FPGA was programmed including with logic analyzer parameters. The first hurdle when interacting with the chip was to receive proper responses. As mentioned earlier, the decoder will set the SDA line low to indicate it received the written information. At first, there were no ACKs received, and after analyzing the signal with an oscilloscope found that the high ACKs occurred every time the LSB of a byte was high. Therefore the ACK was a decaying signal due to the capacitance of the wire. This behavior indicated that the decoder was not responding. Furthermore, the contrary also happened. One could be fooled into thinking that the chip was responding when in fact the decoder was simply not setting the line. To fix this problem a 10k pull up resistor was connected from the SDA line to 3.3V.

Various people were asked to check the wiring of the chip, where some mistakes were found. After making sure the wiring was correct, the logic analyzer was triggered once again to recheck the responses. By using the tri-state conditions as outputs, one could easily detect where an acknowledge should be located. The waves on the analyzer showed glitchy but low responses during the 9th cycle of the clock. Unfortunately, the value read from 0x01 consisted of a bunch of glitches.

Once again, the STA013 and IC datasheets were referenced to make sure the protocol was being used correctly. Before receiving acknowledge signals, the IC datasheet revealed that the maximum bit rate for the standard mode is 100kbits, which was never mentioned in the device datasheet. Therefore, both resources were referenced. After making sure the behavior was correct, we consulted Nathan Ickes who is familiar with the protocol. He verified the behavior and suggested that the tri-state buffer be moved up to the labkit module. Although this change is better design practice, it did not solve the problem.

After asking all the lab aides and TAs, we searched the Internet for possible solutions. Unfortunately, no helpful information was found, other than a possible delay after one of the writes of the configuration file. We decided to attempt to configure the chip anyway, hoping that only the read function was somehow faulty. However, the chip never asserted the DATA-REQ signal at the end as mentioned in the datasheet. The delay after the "soft-reset" write was also inserted, but the chip remained inactive. The soldering was checked for inaccuracies using a voltmeter, but the

result showed no connected neighboring pins. As a last resort, we contacted the creator of a helpful website about the STA013 where the device was purchased, but we are still waiting to hear back.

3.2 Video Module

When testing the VGA video module, a very bottom up approach was used. The first module to be written was the Sync Signal generator module. Testing in simulation involved counting the number of pixel clock ticks per horizontal front porch, sync, and back porch, and the number of horizontal lines per vertical front porch, sync, and back porch. Once the video display showed a solid color, the next test was to see if the display SRAM corresponding to pixels in the intended way: 4 bits per pixel, representing a 16-bit gradient from white to black, with 32 pixels per line of SRAM. These tests progressed in complexity: first, programming the entire SRAM with a single color. Next, half of the SRAM was programmed with one color, and the second half with another. Next, a single character was written directly into the SRAM in the left hand corner, to make sure that calculations for horizontal and vertical placement were correct. These tests were time consuming because the SRAM had to be reinitialzed each test with an appropriate .coe file. Once the tests showed that the pixel SRAM corresponded to each pixel of the display, the character to pixel module was tested. The purpose of the module is to write a single character to a specific line and character position on the display by changing the correct pixel entries in the display SRAM. Though the simulation showed the correct data being written to the SRAM, and the correct pixel information being sent to the VGA display, The display did not change as expected.

LED Module A similar approach was used in designing and testing the LED display. The first module to be tested was the Sync Signal generator, with an input dots[639:0] that represented stripes. A word as hard-coded input to dotes[639:0] displayed a the word as expected. The ascii to pixel converter was tested in simulation and in hardware. The title-display module was simulated with the pre-initialzed contents of the title-list SRAM, and produced the correct dot[639:0] output based on the title. However, the title-dislpay module, when actually synthesized, was only successful in displaying the first letter from each song in the SRAM. The interface part of the LED module was tested by pressing various combinations of up, down, and play. Scrolling of the titles was successful, and the play symbol appeared when play was pressed, but the play signal did not display only by the appropriate song.

4 Conclusion

Alma Although the MP3 decoding chip was intended to alleviate some of the decoding load, it added complexity in other areas. Without successfully configuring the chip, the more important audio playback feature is put aside, which is imperative for this project. Needless to say, we exhausted every debugging approach, but the STA013 never responded. We approached the problem systematically, first by testing with the logic analyzer and requesting help from the lab aides. However, IC was not a protocol many people were familiar with. The lack of thorough resources, even after searching online made it very difficult to find a solution. If we had the opportunity to attempt this project once again, we would choose to use a chip with perhaps not as wide capabilities as the STA013, but which is easier to use.

Amy The product was a good introduction to display signal generation— in its simpler form with LEDs and its more complicated form as a 640 by 480 VGA display. Managing, updating, storing,

and processing that large quantity of information was difficult, and even after lots of design and planning, still easily done wrong. Moreover, the timing conditions for the display are very strict. I would predict that timing issues are what kept both displays from working properly, and am still interested in working on the project further to find out exactly what went wrong and correct it.

5 Acknowledgements/References

Brain, Marshall. "How MP3 Files Work." How Stuff Works Inc. "http://computer.howstuffworks.com/mp31.htm 2005.

Philips Semiconductors. "The IC Bus Specification." Version 2. http://www.semiconductors.philips.com/acroba 2000.

Stroffregen, Paul. "MP3 Player, Using the STA013 Chip.". http://pjrc.com/tech/mp3/sta013.html, Feb. 23 2005.

Supurovic, Predrag. "MPEG Audio Compression Basics." Data Voyage. http://mpgedit.org/mpgedit/mpeg(unc 1999.

We got a tremendous amount of help and encouragement from Jenny, Chris, Charlie, and Prof. Anantha. They are terrific.