Computer System Design - Final Project

Prototyping of an Audio Message Recorder

Team ISIS

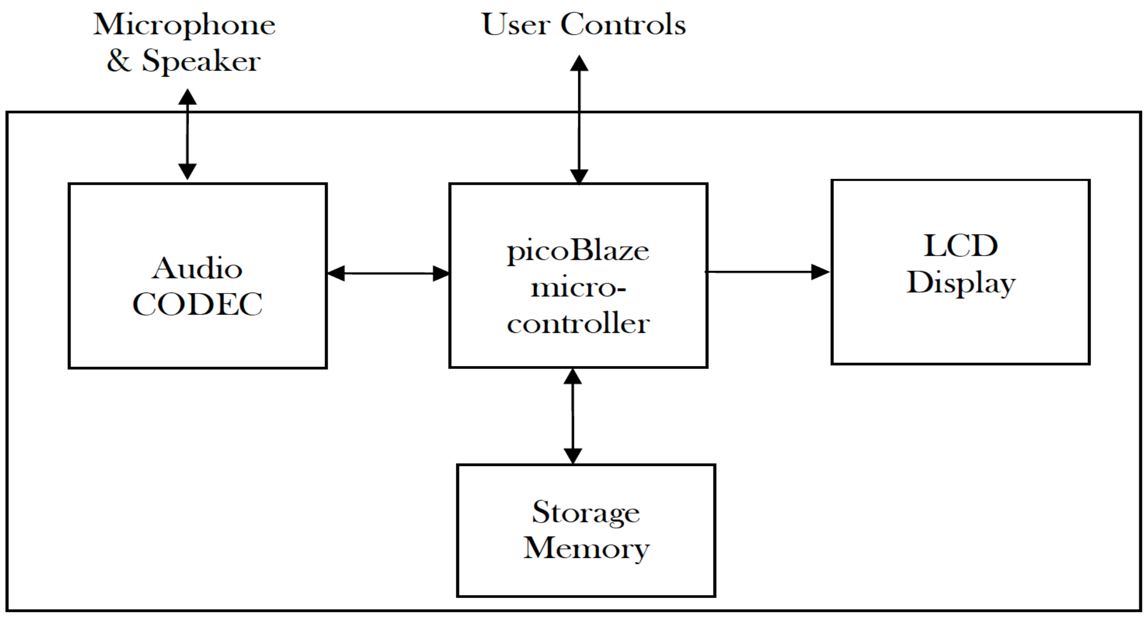
Team Member:	U#:	Work Distribution:
John Gangemi	6871-4612	33.3%
Chris Frazier	5883-1412	33.3%
Bassam Saed	5260-7130	33.3%

Introduction

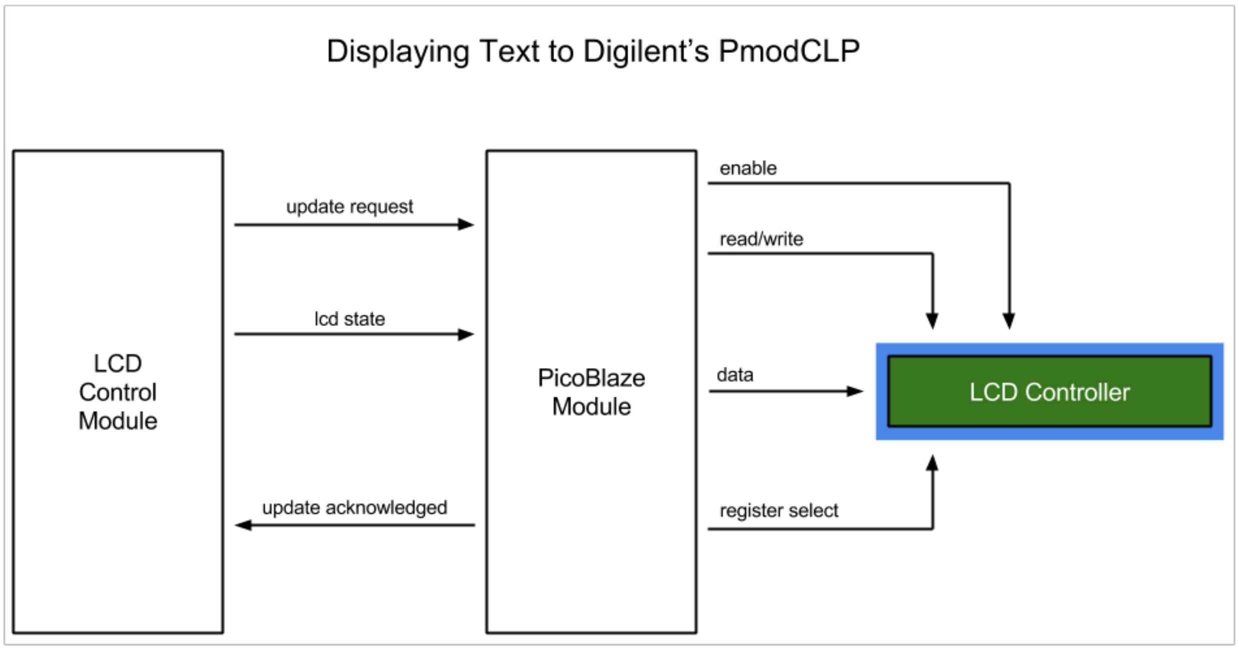
In this project we created an audio record/player using the ATLYS FPGA board. The player was implemented with PicoBlaze soft microcontroller. This design had features of an audio recorder, the ability to record an audio message, play/pause/delete audio. We were able to control/ display by the LCD display push buttons and display switches.

Design

There were four main parts Audio Message Recorder and they were the LCD, AC97, RAM, and the interacting them with each other. LCD would show a play message, record message, delete message, delete all message and volume control. When we select the play and delete messages the system display the audio library and from there we was able to scroll thru the list. when mem_full the LCD displayed memory full. We was able to play or resume and interact when music is playing. majority of the code was run on FSM and had different states for each individual task that needed to be run. RAM was created by an FSM for writing and reading. on the first state it would set the address, data_in and write_en. Then on the next states we set address to the address we want to read, then pulse read_request for on clock, then we read the data from data_out and pulsed read_all



LCD Control Module



In order to display text to the LCD component our group decided to use a "handshake" protocol as discussed in lecture. The basic premise for the design exists as follows…

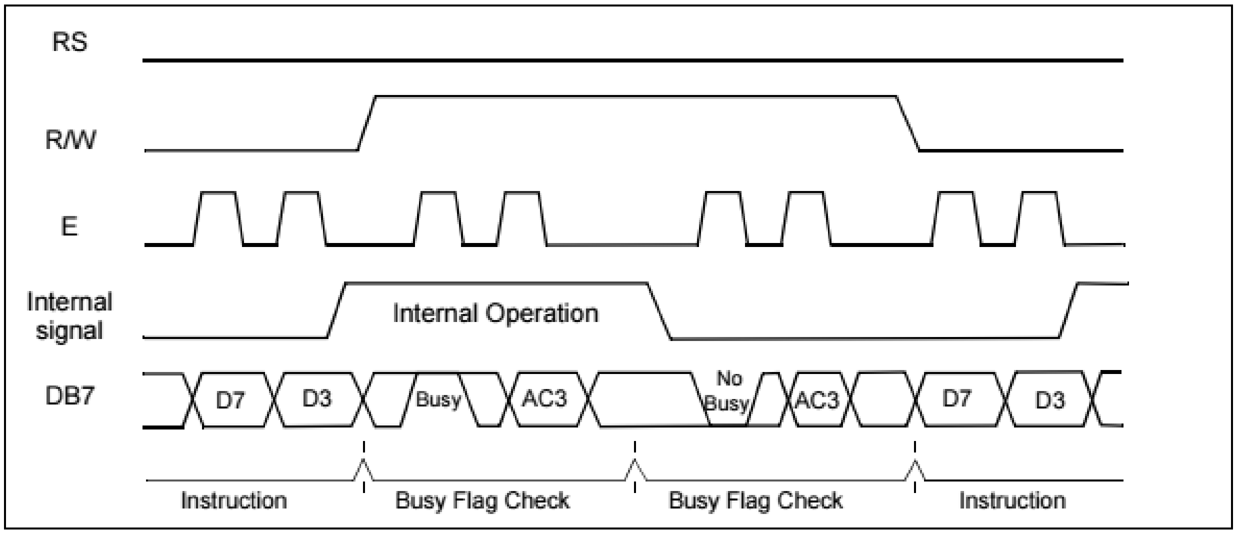
The LCD Control Module implemented in verilog will assert an 'update request' given appropriate forms of user interaction such as a push-button press. The exact push-button pressed will define the next state of the LCD or rather the next message to be displayed. This is accomplished by sending an encoded byte across the 'lcd state' wire to the PicoBlaze module. After one clock cycle of sending a request and encoded data, PicoBlaze will accept the request and decode the lcd's next state, if not already processing a previous update request. Once the message is decoded PicoBlaze will jump to the correct sub-routine and begin to send character-by-character to the LCD's controller. After completing the necessary number of write transactions to the LCD, PicoBlaze will assert 'update acknowledged' before jumping back to the main process loop. Only now can the verilog LCD Control Module return to the actual state requested through user interaction.

PicoBlaze

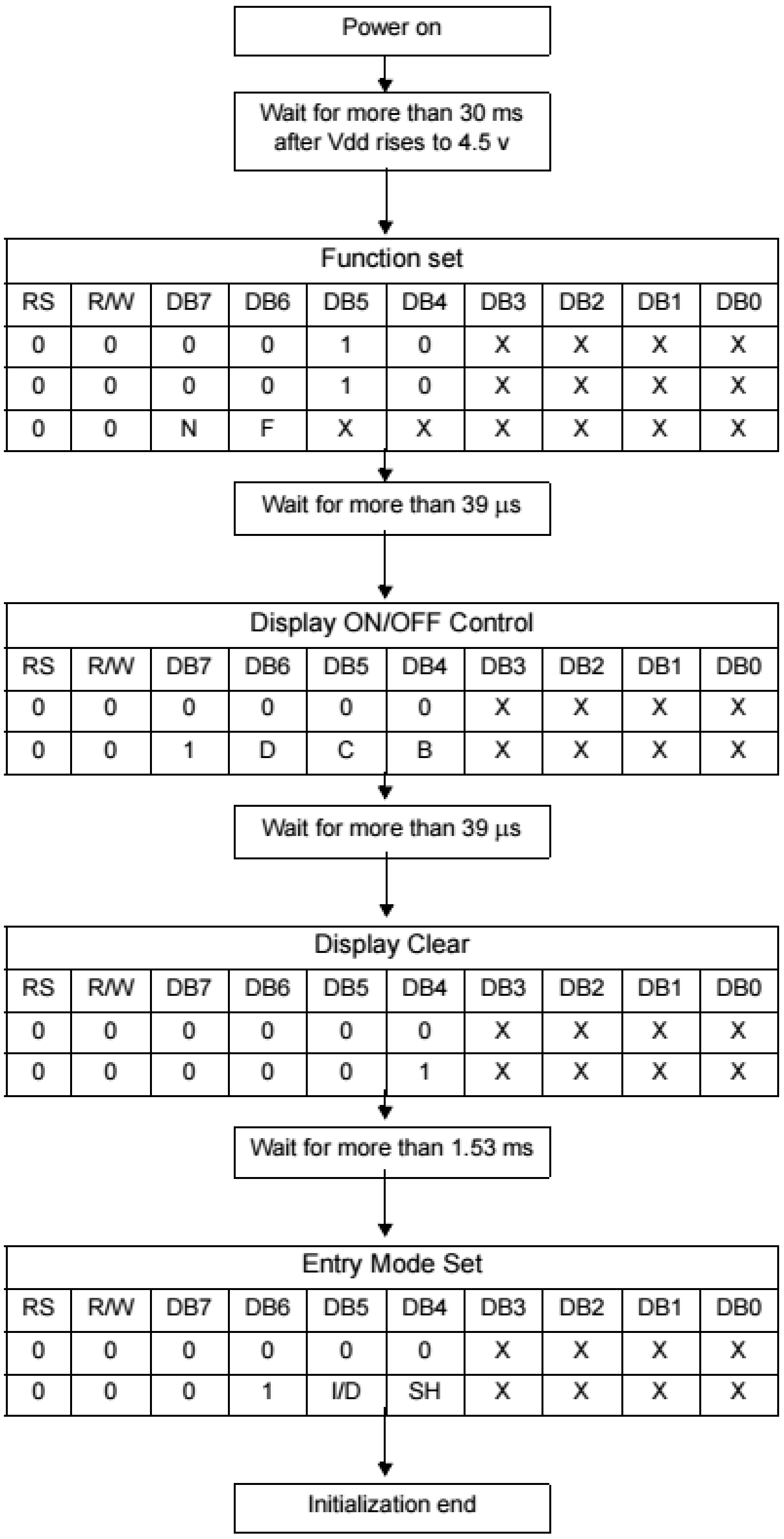
Implementing communication to the LCD controller is done through precise timing via sub-routines coded in assembly language. By far the most critical subroutine involves initializing the LCD controller in which a number of character codes must be written in a sequentially-timed manner. If successful the welcome message will be displayed and the soft-microprocessor will be put into its main loop.

PicoBlaze's main loop subroutine constantly polls for an 'update request' from another verilog module. Upon comparing the input to a value of 01 hex will the main loop jump to another subroutine that handles decoding of the 'lcd state'. If the state exists then another jump to the correct subroutine will occur else a jump back to the main loop.

Every character written to the LCD controller must happen in timed intervals of 40 microseconds. Having only 4-bits to represent a character requires that the byte of data be split into high-order and low-order nibbles. Each nibble sent must be separated by 1 microsecond. To handle these timing constraints subroutines were created solely for the purpose of delaying character and nibble writes to the LCD controller. Shown below are timing diagrams sourced from the Samsung KS0066U datasheet.



*Source from Samsung KS0066U Datasheet



Power on

Wait for more than 30 ms after Vdd rises to 4.5 v

Function set

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	1	0	X	X	X	X
0	0	0	0	1	0	X	X	X	X
0	0	N	F	X	X	X	X	X	X

Wait for more than 39 µs

Display ON/OFF Control

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	X	X	X	X
0	0	1	D	C	B	X	X	X	X

Wait for more than 39 µs

Display Clear

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	X	X	X	X
0	0	0	0	0	1	X	X	X	X

Wait for more than 1.53 ms

Entry Mode Set

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	X	X	X	X
0	0	0	1	I/D	SH	X	X	X	X

Initialization end

*Source from Samsung KS0066U Datasheet

Audio Codec

The AC97 Codec on the FPGA board interfaces with the verilog module using five pins:

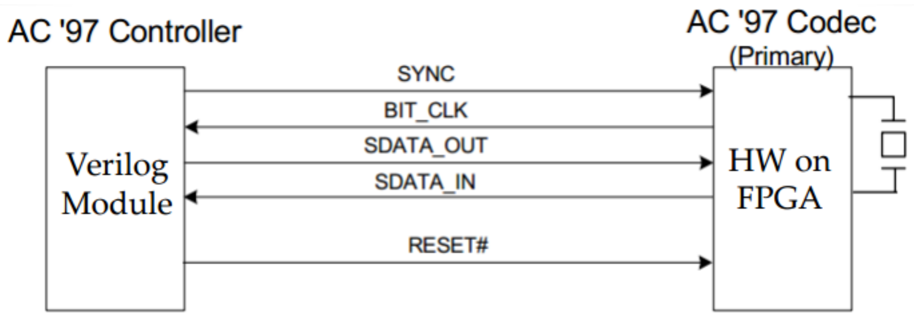


Figure 6. Controller to Codec connections

- BIT_CLK is a 12.288 MHz clock signal the controller will use to move data bits
- SYNC synchronizes each 256 bit frame which will be split up into 13 slots. Each SYNC cycle is 256 bit_clk cycles long. This makes the SYNC signal a 48 KHz oscillating signal. The SYNC signal is high for the first 16 bits of each frame. This is the first slot and is called the “tag” slot. The SYNC signal should be low for the remaining 12 slots called the “data” slots.
- SDATA_OUT and SDATA_IN transfer the data serially between the controller and the codec.
- RESET does just that. It performs a cold reset to the codec which means all data in the registers will be lost.

The following timing diagram shows how the 13 slots are assigned to specific data which is communicated on the SDATA lines:

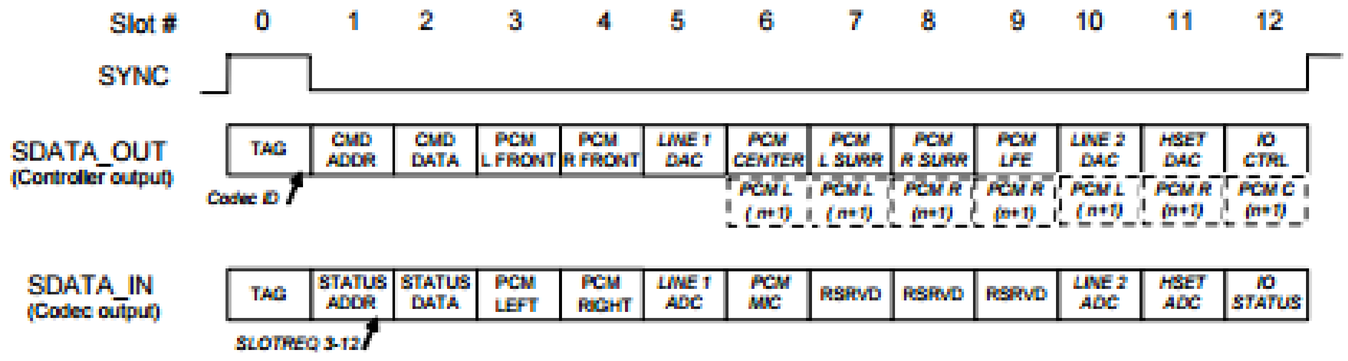
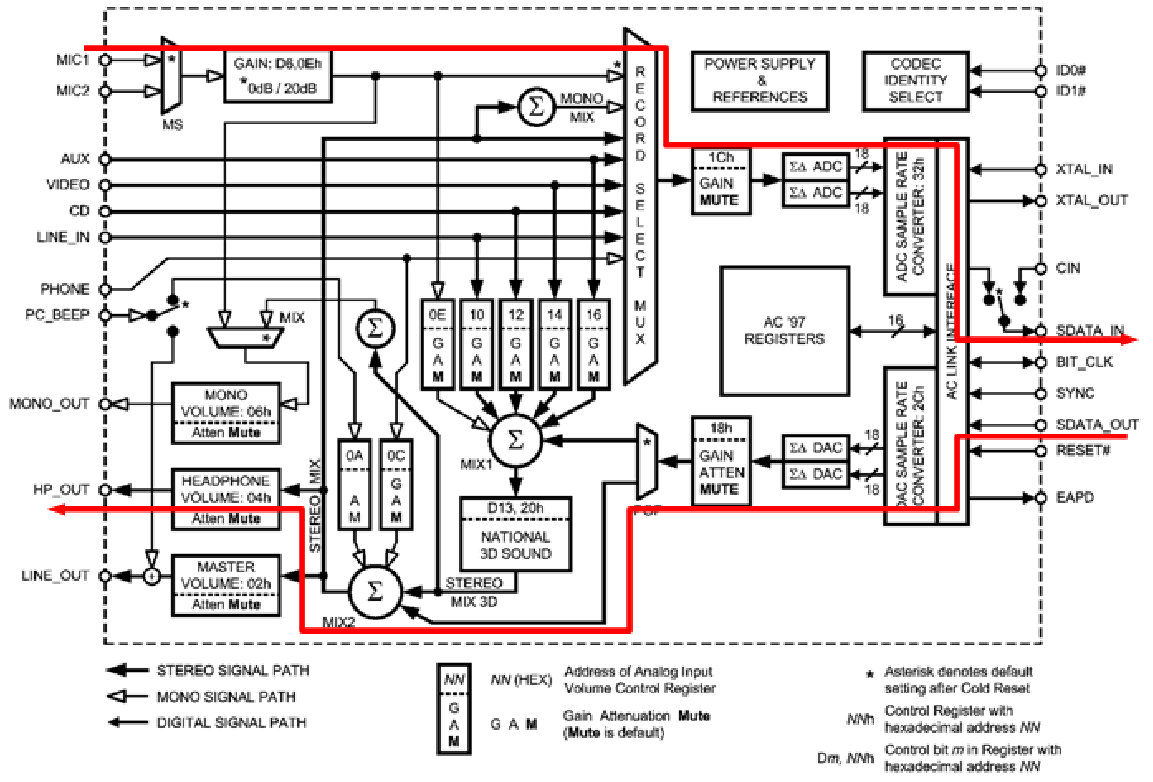


Figure 10. Bi-directional AC-link Frame with Slot assignments

The following diagram shows how the serial audio data flows through the codec:

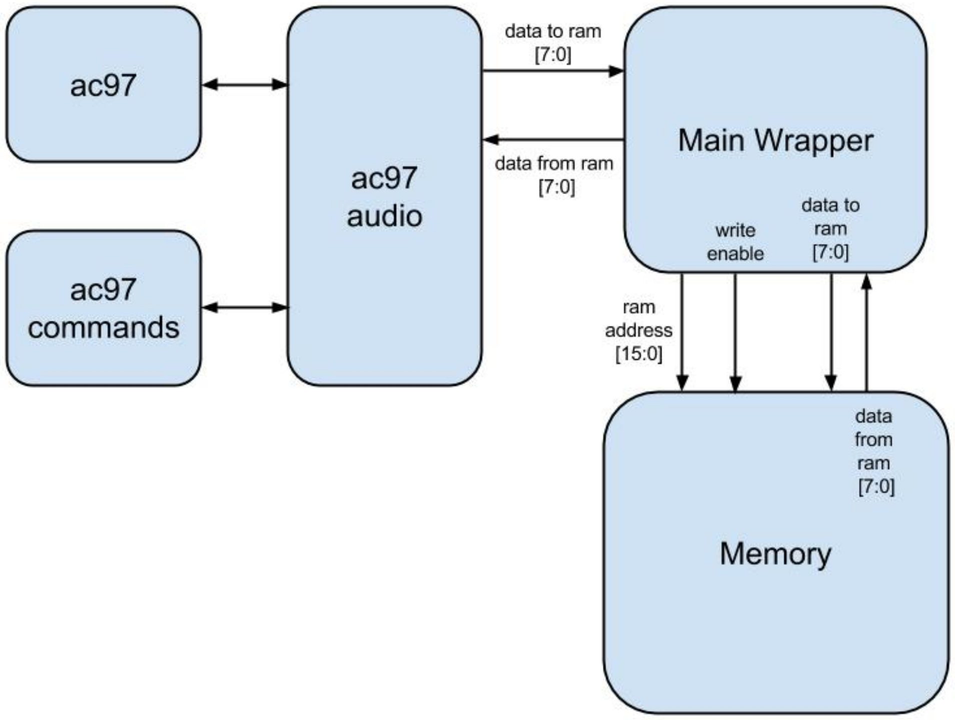

MIC1
MIC2
MS
GAIN: D6,0Eh
*0dB / 20dB
MONO MIX
RECORD SELECT MUX
POWER SUPPLY & REFERENCES
CODEC IDENTITY SELECT
ID0#
ID1#
AUX
VIDEO
CD
LINE_IN
PHONE
PC_BEEP
1Ch GAIN MUTE
ΣΔ ADC
ΣΔ ADC
ADC SAMPLE RATE CONVERTER: 32h
XTAL_IN
XTAL_OUT
CIN
AC '97 REGISTERS
AC LINK INTERFACE
SDATA_IN
BIT_CLK
SYNC
SDATA_OUT
RESET#
EAPD
MIX
0E GAM
10 GAM
12 GAM
14 GAM
16 GAM
MONO_OUT
MONO VOLUME: 06h Atten Mute
HP_OUT
HEADPHONE VOLUME: 04h Atten Mute
LINE_OUT
MASTER VOLUME: 02h Atten Mute
STEREO MIX
0A AM
0C GAM
MIX1
MIX2
D13, 20h NATIONAL 3D SOUND
STEREO MIX 3D
18h GAIN ATTEN MUTE
ΣΔ DAC
ΣΔ DAC
DAC SAMPLE RATE CONVERTER: 2Ch
POP
STEREO SIGNAL PATH
MONO SIGNAL PATH
DIGITAL SIGNAL PATH
NN (HEX) Address of Analog Input Volume Control Register
G A M Gain Attenuation Mute (Mute is default)
* Asterisk denotes default setting after Cold Reset
NNh Control Register with hexadecimal address NN
Dm, NNh Control bit m in Register with hexadecimal address NN

We coded the ac97 controller in Verilog and synthesized it onto the board. The controller was split up into three files:

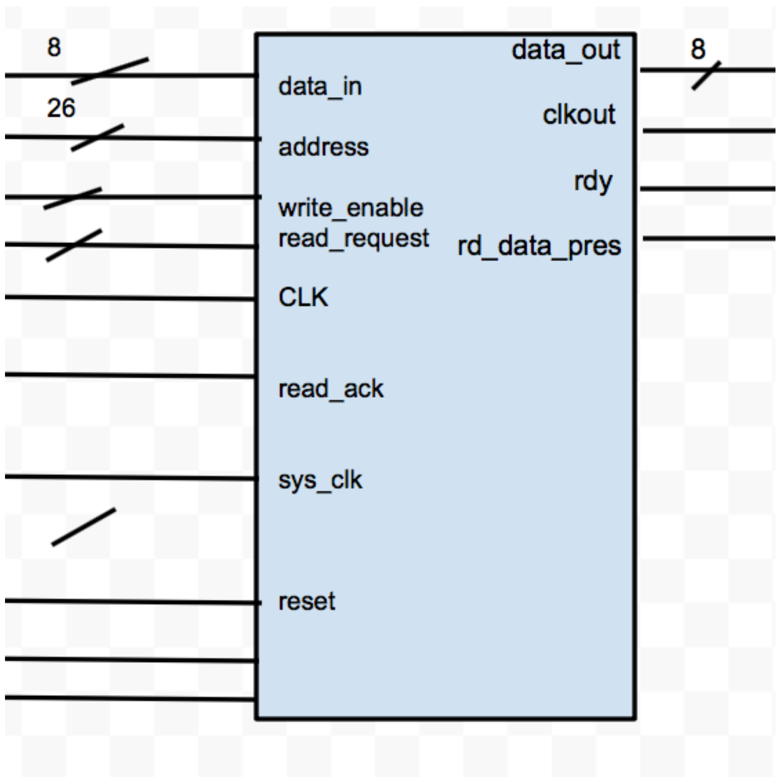
- ac97.v: We generated the sync signal, separated each slot and assigned the relevant data to each slot. We also generated a “ready” signal that would be an output and would let the rest of our system when audio data is ready to be used.

- ac97commands.v: We assigned values to the relevant registers to achieve optimal sound, and selected which settings we needed.

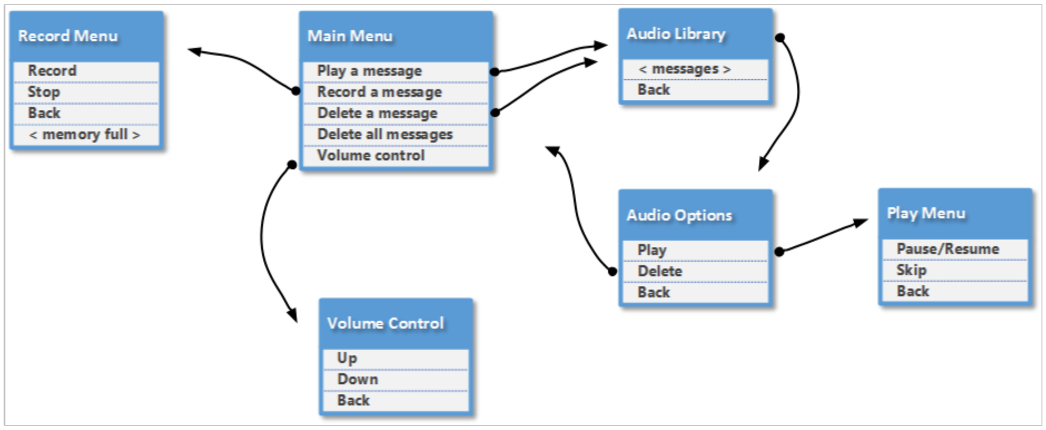
- ac97audio.v: In this file we assigned the left and right 20 bit audio channels to one 8 bit data out that we would use to store in memory. This module wraps around the other two modules and prepares the data to be input and output to the codec serially and to RAM in 8 bit parallel form.



Storage Memory



For RAM we created an FSM for writing and reading. On the first state it would set the address, data_in and write_en. Then on the next states we set address to the address we want to read, then pulse read_request for on clock, then we read the data from data_out and pulsed read_all. having the right clk and the sys_clk running smoothly together was one of the most difficult parts for the RAM. having an input clk at 100MHz and system clock at 37.5 MHZ. For writing to the design we simply just assigning what input goes to the correct spots. For reading we had read_request for one clock cycle. rd_data_pres indicated an ‘1’ when a read was performed and the data is ready. the data that was to be outputted came from data_out and the high end of read_ack.



Testing and Results

Our initial design suffered from incorrect communication with the memory module. therefore we designed another project to test reading and writing of the memory module against a top module wrapper in which an 8 bit counter wrote the 8-bit binary value 0-255 to the first 255 addresses of 128 MB system memory. after writing to the memory we then read back the first 255 address locations and displayed the values to all 8 onboard LEDs. Each read occurred every 1 second or every 37 millions clock cycles. All data gathered from this test helped up implement the memory module with our main project.

To test the audio controller we set the data_in signals equal to the data_out signals in the ac97audio file. This enabled the data to pass through the codec and play back without using the memory. This gave us an idea of the audio quality to expect and allowed us to test our register values. We made many attempts to improve the audio quality including reducing the mic gain. We also tried to increase the rate at which we generated the ready signal, however this took up too much memory when we made the connections to RAM.

Synthesis

creating multiple FSM’s in verilog to create an audio/record player on our FPGA. Working with RAM AC97 and the LCD we were able to interact the three components to record, skip, delete, etc with certain dip switches, buttons and LCD. Also being able to go from our home screen to to the main menu and from there we had the option to go to many different option like volume control or record menu. once in those different option we were able to do the command needed.

Conclusion

In conclusion, we successfully met a majority of the requirements except for a skip function and navigation while recording/playing audio. Our mp3 player was able to record and store five three minute messages in RAM. We could then play and pause these messages back through the headphone port. Our volume control consisted of three settings: low, medium, and high. The volume was persistent and when we navigated away from the volume menu, it stayed at the same level. Overall, we learned a lot about working with the ac97 audio codec and communication protocols in relation to hardware design and FPGA synthesis. In the future, we could use the knowledge we gained from this project to interface with other audio components.