ECE 532 Digital Hardware

Design Project

Guitar Sound Processing System

Students: Quan Gun 990049425

Brad Drehmer 990905455

Professor: Paul Chow

Date: April 10, 2004

Overview

Goals: The goal of this project was to implement digital effects on audio signals from an electric guitar. The guitar was plugged into an amplifier, which was then connect to the Xilinx multimedia board using the composite audio jacks. By changing the state of the user input switches different audio effects could be selected. The effects were:

1 – *clean mode* : no processing occurs, input is passed straight to the output

2 – *distortion mode*: the audio input is hard clipped such that its magnitude does not exceed a certain level. This produces a hard rock type of sound.

3 – *polynomial waveshaping mode*: the input is distorted using a technique referred to as polynomial waveshaping. This produces a more mellow sounding distortion.

4 – *echo mode*: the input is added to a short buffer and then played along with the oldest sample in the buffer, producing a reverberation or echo sound.

Our Guitar Audio Processing System can be described by the following diagram:

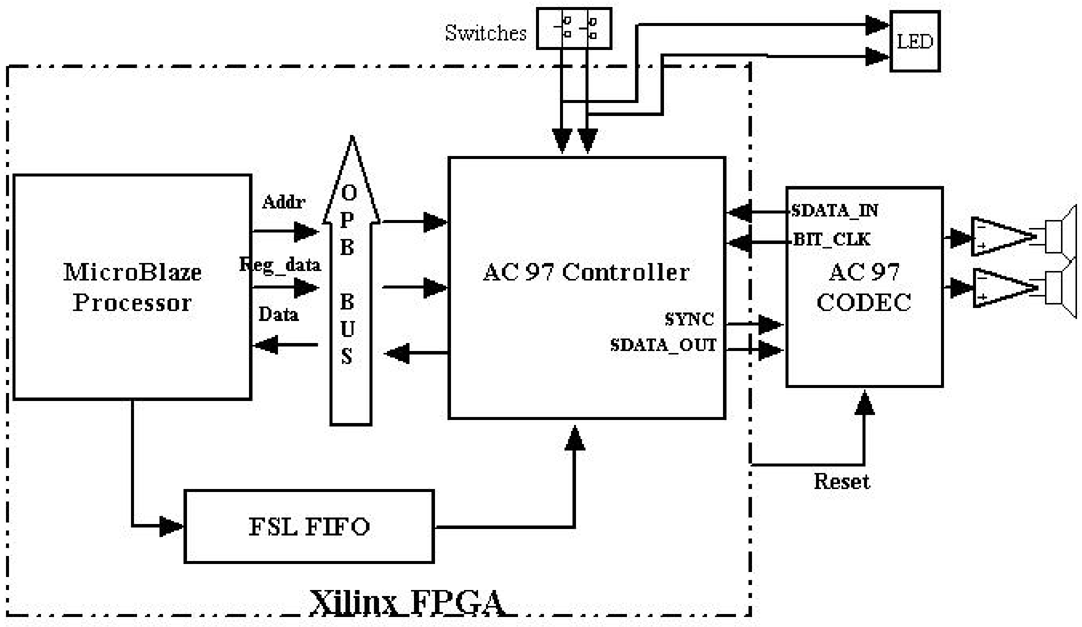


Figure 1: Top level block diagram

We use the two existing IPs from Xilinx’s MP3 project: AC 97 controller and FSL FIFO. We add two switches and two LEDs into the AC 97 controller so that MB processor can read the status of the switches from AC97 controller’s status register. The switches select one of the four digital audio signal processing operations and also turn on the appropriate LEDs. We also modified the core of the AC 97 controller in order to capture the audio sample correctly; more detailed information can be found in the Description of the IP block section. No modification has been made to the FSL FIFO block. Our audio processing unit is implemented in C Programming Language, which greatly reduce the amount of time spent on debugging.

Outcome

The biggest problem we had in this project was figuring out how to gather audio samples from the codec. Our project was working as expected in our proposal. Future improvements include a larger buffer and more efficient code for the echo buffer. With the addition of a memory controller, off chip memory could be used as a buffer. An extension of the echo buffer is to add recording and playback functionality.

Although there was a bug in recoding the audio sample in the AC 97 controller, we had found a workaround for this problem. We found that the AC 97 controller recorded extra zero samples into the recording FIFO, so we skipped the zero samples in our software program so that we could generate noise free playback. We suspected the problem might lie between the AC97 Clock and the OPB Clock domains when an audio sample was transferred to the recording fifo. We had run a functional simulation in Modelsim to simulate the AC97 controller, and we did not find any obvious problems. If we had time, we would have run a simulation on the overall system to verify correct operation and find out the cause of the problem.

Description of the Blocks

AC 97 Codec

The AC 97 Codec is a physical ASIC that connected to the FPGA, it provided the SData_in and Bit_CLK signals to the AC 97 controller and received SData_out and Sync signals from the AC 97 controller. The FPGA needed also to provide a reset signal to reset the codec into operational mode. When the Codec was reset, its internal registers were set to default values, many of the registers were set to mute state and electrical paths were set to open such that no output could be observed. The AC 97 controller could write to the registers of the Codec so that the appropriate path would be enabled and the signal could be send to the controller and out to the speakers. Please see Appendix for a detail block diagram of the Codec and its registers' description. The data sheet of the Codec could be found in http://www.national.com/pf/LM/LM4549A.html. We had enabled the following registers: 02h, 10h, 1Ah, 1Ch, 20h, 18h. The Codec was set to the following configuration:

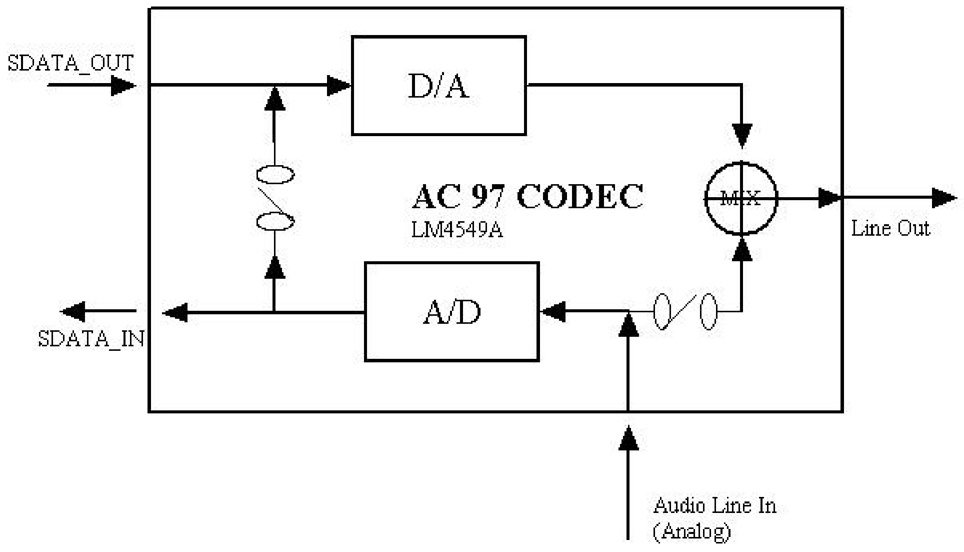


Figure 2: Our Codec configuration

AC 97 Controller

This IP block was taken out from Xilinx's MP3 project. This IP block (version 2) was a modified version of the OPB AC97 Sound Controller (version 1) that was provided with Xilinx's ML300 development board. The ML300 EDK Reference Design User Guide had very good description on the functionality and architecture of the controller. The user guide could be found from: http://www.xilinx.com/products/boards/ml300/docs/ml300_ref_des_ug.pdf

The controller IP managed three primary functions to control the AC97 Codec: the playback FIFO, the record FIFO and the Codec's control and status registers. We had added a few circuits and modifications to the AC 97 controller from the MP3 project as illustrated in the following figure 3 (version 3). In version 1, there was a playback FIFO inside the controller (see Appendix for original block diagram). The playback FIFO was replaced by an external FSL FIFO in version 2. We had added two switches and two LEDs to version 2 of the controller. The two switches could write their value to the AC 97 controller's FIFO status's register, and switched the LEDs on or off. The FIFO status register was memory mapped to the Microblaze's local memory and Microblaze could check the status of this register to execute different functions. The recording data, as well as Codec control/Register data were sent and received via the OPB bus. The OPB bus clocked at 27 Mhz, doubling the speed of the BIT_CLK enabled the processor to process the data properly.

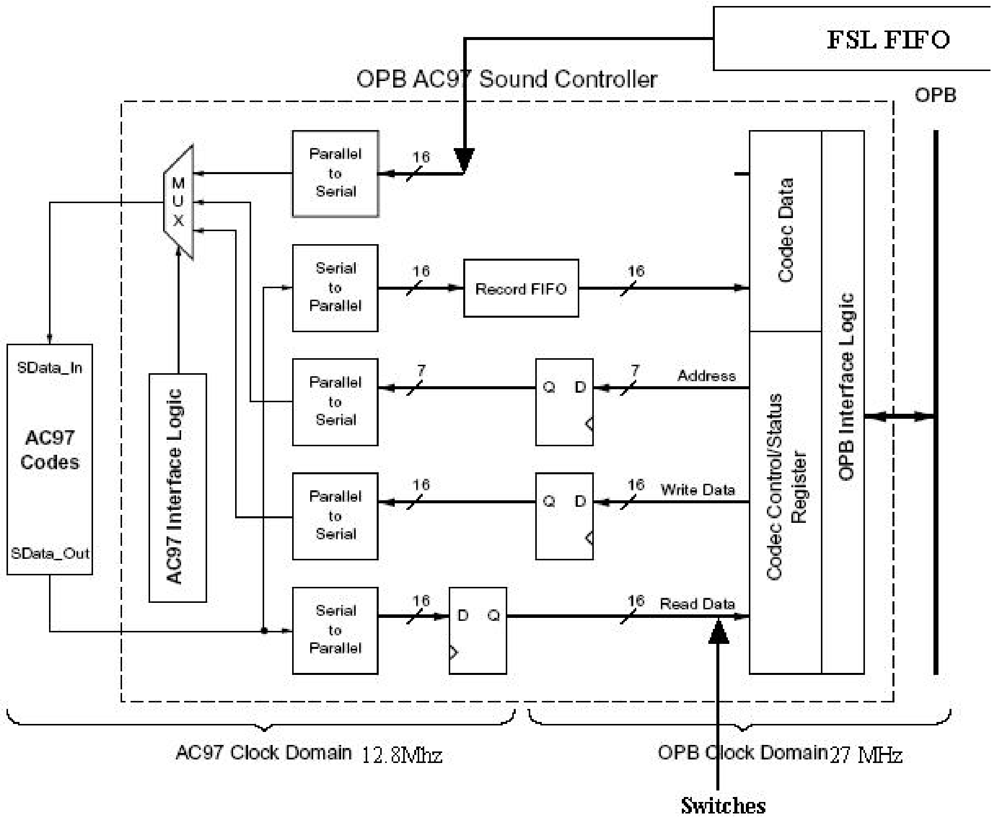


Figure 3: Our version of AC 97 sound controller

We had also modified the core of the controller to sample the audio data properly. We had to go through the core file and adjust the slots and bit numbers so as to match the LM4549 datasheet. In the process of “Get_Record_Data” of the opb_ac97_core.vhd file, it was using slot_no= 4 for left channel and 5 for right channel to sample the audio data, we changed to slot_no = 3 and 4 respectively. This modification enabled us to capture the audio data, but it captured an extra 0 sample in every 3 or 4 samples. This caused a great deal of noise to be present in the audio signal. Therefore, there should not be zero magnitude samples appearing in the recording FIFO. Our workaround was to skip the 0 value samples in the software program using a recursive getsample() function, and the result turned out acceptable.

The memory map would be updated as shown in the following table:

Register Address	Bits	Read/ Write	Description
Base Address + 4	[16:31]	R	Read 16 bit data sample from record FIFO. Data should be read two at a time to get data from the left channel followed by the right channel.
Base Address + 8	[24]	R	Record FIFO Overrun: 0 = FIFO has not overrun 1 = FIFO has overrun **Note:** Record FIFO must be reset to clear this bit. Once an overrun has occurred, the Record FIFO will not operate properly until it is reset.
	[25]	R	Play FIFO Underrun: 0 = FIFO has not underrun 1 = FIFO has underrun **Note:** Not used, play FIFO has been replaced by FSL FIFO
	[26]	R	Codec Ready: 0 = Codec is not ready to receive commands or data. (This may occur during initial power-on of immediately after reset.) 1 = Codec ready to run
	[27]	R	Register Access Finish: 0 = AC97 Controller waiting for access to control/status register in Codec to complete. 1 = AC97 Controller is finished accessing the control/status register in Codec. **Note:** This bit is cleared when there is a write to the"AC97 Control Address Register" (described below).
	[28]	R	Out Data Exists: 0 = Out Data doesn’t Exist 1 = Out Data Exists
	[29]	R	Switch 1: 0 = Switch 1 off 1 = Switch 1 on
	[30]	R	Switch 2: 0 = Switch 2 off 1 = Switch 2 on
	[31] (LSB)	R	Playback FIFO Full: 0 = Playback FIFO not Full 1 = Playback FIFO Full
Base Address +12	[30]	W	Clear/Reset Record FIFO: 0 = Do not Reset Record FIFO 1 = Reset Record FIFO. Resetting the record FIFO also clears the "Record FIFO Overrun" status bit.

FSL FIFO

The FSL FIFO was taken from the Xilinx MP3 project. The FSL bus configuration of the Microblaze can be used in conjunction with any of the other bus configurations. Microblaze had specified FSL related instructions defined in “mb_interface.h” to access the data in the FSL FIFO.

The following instructions made the FSL FIFO a handy block:

• **get, put** : Blocking Read and Blocking Write of data to the FSL. The control signal is set to ’0’.

• **nget, nput** : Non-blocking Read and Non-blocking Write of data to the FSL. The control signal is set to ’0’.

• **cget, cput** : Blocking Read and Blocking Write of data to the FSL. The control signal is set to ’1’.

• **ncget, ncput** : Non-blocking Read and Non-blocking Write of words to the FSL.

“mb_interface.h” can be found under the microblaze_0\include directory. We did not make any modifications to the FSL FIFO. We used 2024 depth for the FSL FIFO so that there were enough buffers for playback. The implementation of the FSL FIFO could be described by the following diagram:

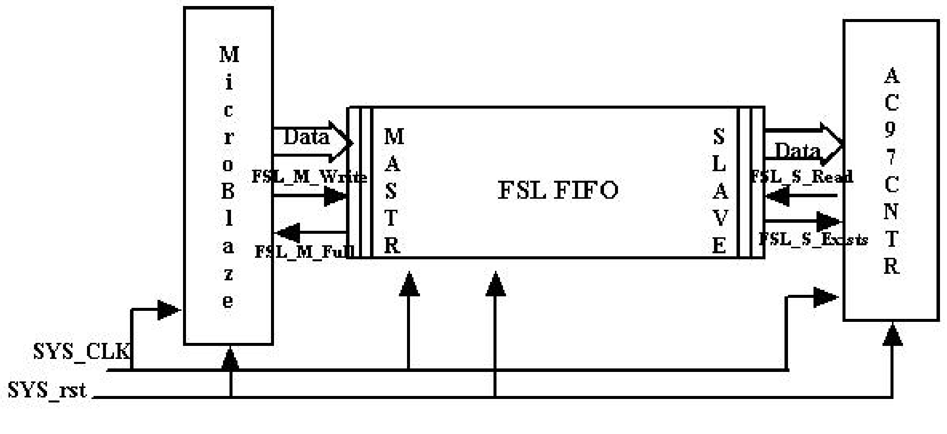


Figure 4: FSL FIFO Implementation

soundcontroller.c Description

Firstly, all required include files and parameters were defined. mb_interface.h contains the microblaze_bwrite_datafsl() function that is used to play data via the fls bus. This is how the mp3 player example played audio also. Most of the parameters and variables defined at the start of the program were also used in the mp3 player example and just needed to be copied over. The effect parameters that were defined are specific to this project.

The functions used to read and write to the codec registers were also included in the mp3 player example. By writing to various registers on the codec different data paths can be established and adjusted. The LM4549 datasheet gives the address of each register, along with its default value. The register write functions were used in the initializing functions. The recording source was selected, recording parameters were set such as sample rate, volume levels were set, and unwanted audio sources were muted. A table provided in the LM4549 datasheet gives the format of each individual register, telling you what values should be written for the desired outcome. To write to a register on the codec use `WriteAC97Reg(0x`*`addr`*`,0x`*`value`*`);` where addr is the address of the register according to the datasheet and value is the data to be written to the register.

The user input switches were used to select which audio effect should be applied. At first the switches and leds were on the OBP bus. There was some problem with the GPIO however that caused the bus to be held and, since the OPB bus was used to read audio samples, the audio samples were corrupted. A simple solution was to tie the switches directly to the leds and also to some bits in the fifo status register that were unused. So, to read the state of the switches the fifo status register was masked appropriately. Note that the value read using the mask had to be bit shifted one place to the right because the switches were in bits 1 and 2 of the status register instead of bits 0 and 1.

As described earlier, a recursive function was used to gather audio samples because of a problem that caused too many zero valued samples. The function simply calls itself until a non-zero sample is found. Samples are played by writing to the fsl bus.

Several audio effects were written:

a) Clean mode ~ no effect is applied to the audio, it is played clean. Several methods were developed for clean mode.

i) clean() – an analog path was created on the codec for the audio.

ii) loopback() – the audio is digitized on the codec but then played directly without leaving the codec in what is referred to as 'loopback mode'

iii) ourloopback() – mostly used for debugging, the audio is digitized, passed to the uBlaze, and then played directly without any audio effects.

b) Distortion ~ a sample is read from the codec and then hard clipped using a mask. This produces a hard rock type distortion effect. By printing out the audio samples, we realized that they were in 2s compliment format. Specifically, when there was no input, the sample values were very small or very large hex numbers, which corresponds to small positive and negative samples in 2s compliment. Therefore, the sign of the sample had to be determined using the signbit mask and the appropriate mask used for negative and positive samples in order to obtain the desired clipping.

c) Polynomial Waveshaping ~ this effect was taken from www.musicdsp.org (click archive -> effects -> waveshaper (simple description)). A 2s compliment audio sample was gathered and its sign determined. The sample has to be normalized before the effect can be applied. The effect did not work properly for negative samples and, since we only figured out how to read data samples one day before the demo, we didn't have time to figure out why. The simple solution was to play negative samples clean for this effect. In any case a distorted sound can be heard, but it is a softer sounding distortion than hard clipping.

d) Echo ~ a buffer was set up to hold previous sample values. The current audio sample was played along with the oldest sample in the buffer. Note that the buffer must be cleared each time the echo effect is first selected. Also, due to the slow speed of the uBlaze, this implementation of the echo buffer is not very practical. Since every sample in the buffer is shifted for each sample, a small buffer must be used. A longer buffer could be used if pointers were used to select the location of the new and oldest audio samples in the buffer. Also, if off chip memory was used a much bigger buffer could be used. One obvious extension is to modify the echo effect into a record function.

main() is a fairly simple function. The codec is initialized, and then an infinite loop is entered. In the loop, the value of the user switches is read and then the appropriate effect is selected. As mentioned in the program comments, the oldswitches variable is used to avoid unnecessary register writes. This was particularly necessary when clean() was used instead of loopback() because the analog source had to be muted before any of the effects that required digital samples were used. clean() was used instead of loopback() up until the last minute, but it was decided that loopback() should be used so that the gain of each effect could be controlled easily by writing to the same register (0x18).

Description of Design Tree

A zip file of the clean project is included with this report. The structure of the project was described as below:

system.mhs: A higher level description of the hardware modules in the system

system.mss: A higher level description of the software modules in the system

../_xps/: Not being used.

./code/: software source code run on processor

    sound_controller.c: It include following functions:

        Read, write to Code registers

        Read audio sample from processor local memory

        Read the status of the switches

        Initialize the AC 97 Codec

        Distortion audio processing

        Clean playback

        Echo audio processing

        Waveshape Audio processing

        The Main function

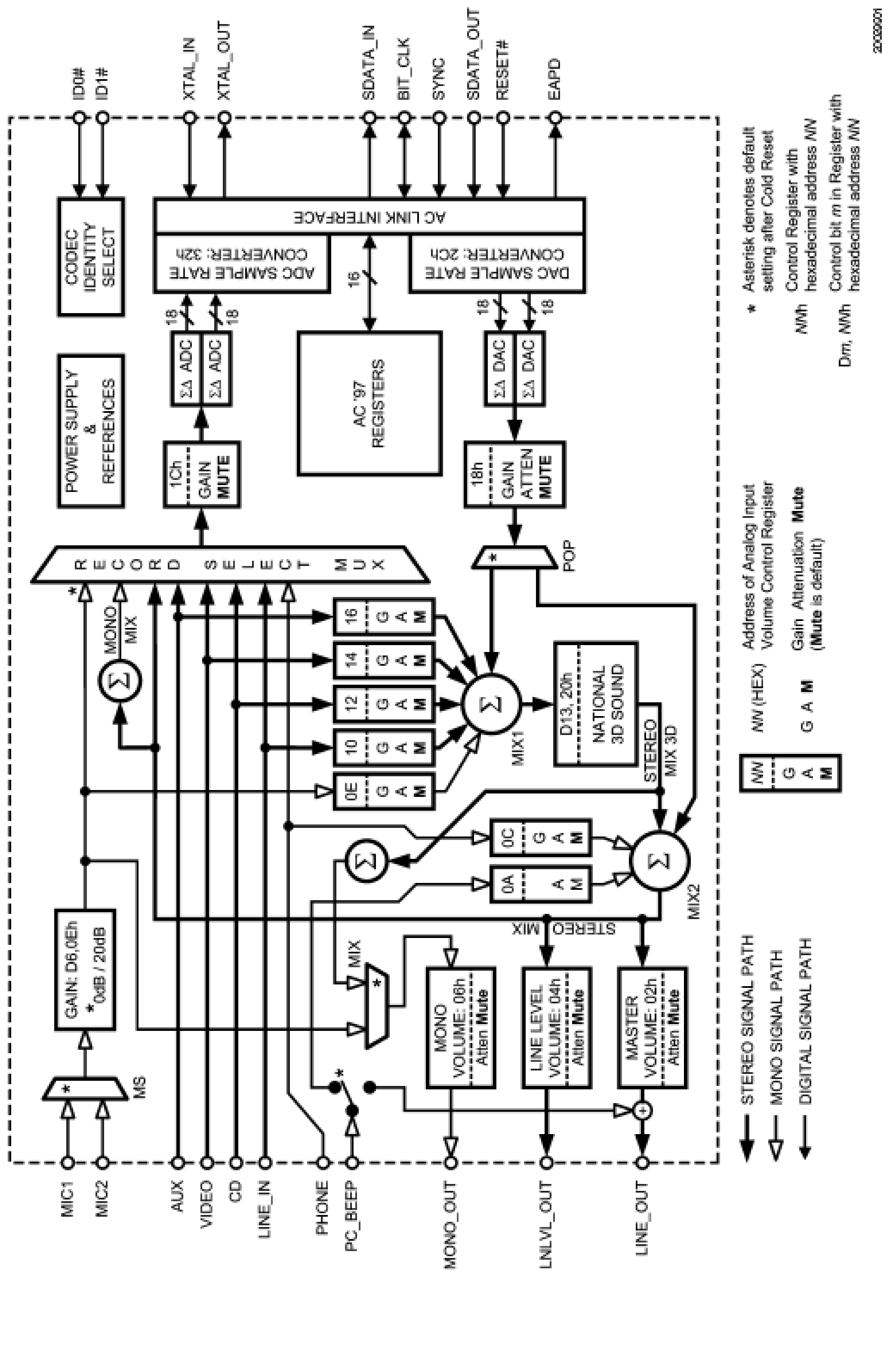
./data/: Contains the user constraint file (.ucf) which assigns external pins to ports, sets clock speed, etc

./pcores/: User designed peripherals

    FSL FIFO IP

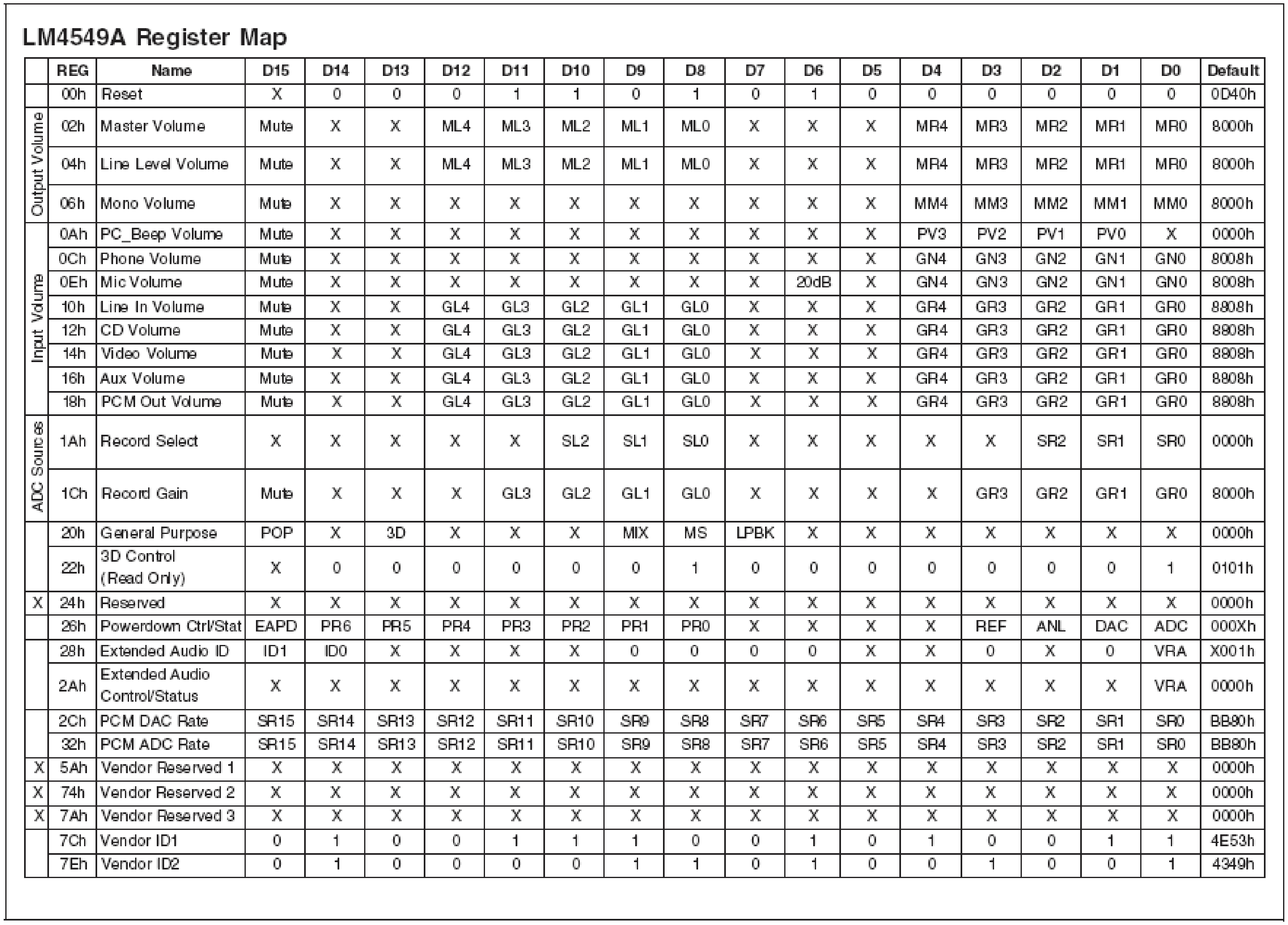
    AC 97 Controller IP

Appendix A: AC 97 Codec diagram


ID0#
ID1#
XTAL_IN
XTAL_OUT
SDATA_IN
BIT_CLK
SYNC
SDATA_OUT
RESET#
EAPD
CODEC IDENTITY SELECT
AC LINK INTERFACE
ADC SAMPLE RATE CONVERTER: 32h
DAC SAMPLE RATE CONVERTER: 2Ch
POWER SUPPLY & REFERENCES
ΣΔ ADC
ΣΔ DAC
AC '97 REGISTERS
1Ch GAIN MUTE
18h GAIN ATTEN MUTE
RECORD SELECT MUX
POP
MONO MIX
MIX1
MIX2
D13, 20h NATIONAL 3D SOUND
STEREO MIX 3D
STEREO MIX
GAIN: D6,0Eh *0dB / 20dB
MS
MIX
MONO VOLUME: 06h Atten Mute
LINE LEVEL VOLUME: 04h Atten Mute
MASTER VOLUME: 02h Atten Mute
MIC1
MIC2
AUX
VIDEO
CD
LINE_IN
PHONE
PC_BEEP
MONO_OUT
LNLVL_OUT
LINE_OUT
STEREO SIGNAL PATH
MONO SIGNAL PATH
DIGITAL SIGNAL PATH
* 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
NN (HEX) Address of Analog Input Volume Control Register
G A M Gain Attenuation Mute (Mute is default)
20080001

Block Diagram

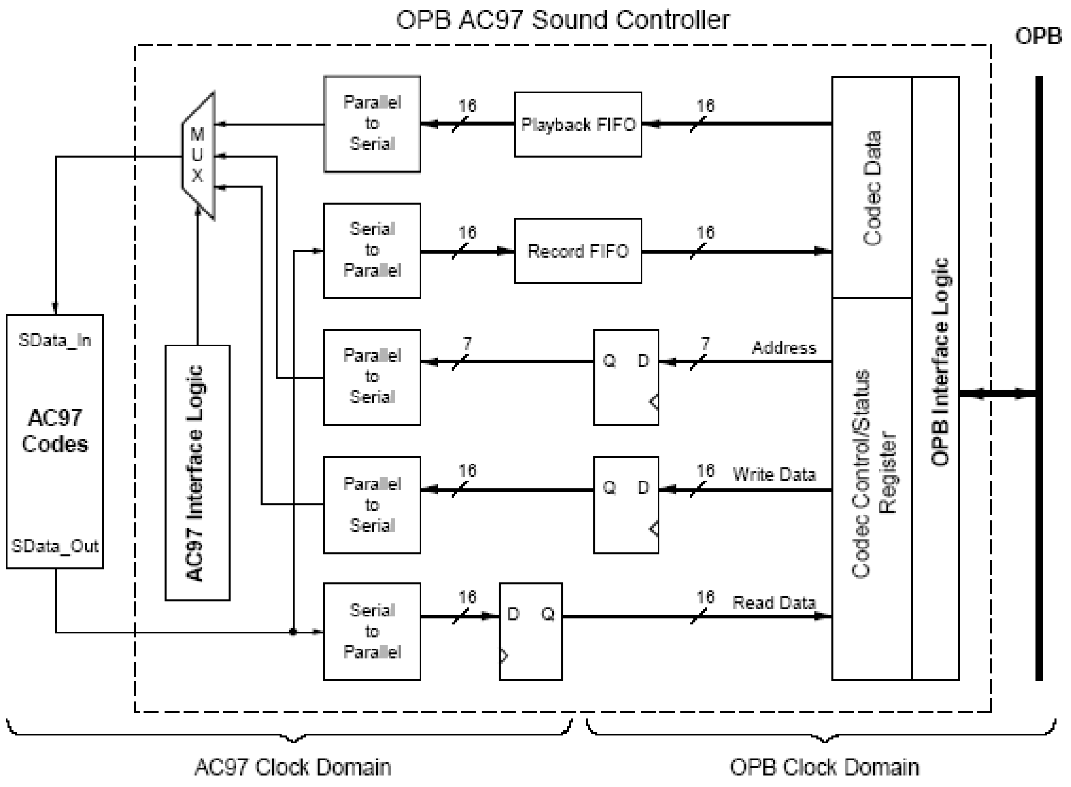
AC 97 Codec Registers Descriptions



LM4549A Register Map

	REG	Name	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	Default
	00h	Reset	X	0	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0D40h
Output Volume	02h	Master Volume	Mute	X	X	ML4	ML3	ML2	ML1	ML0	X	X	X	MR4	MR3	MR2	MR1	MR0	8000h
	04h	Line Level Volume	Mute	X	X	ML4	ML3	ML2	ML1	ML0	X	X	X	MR4	MR3	MR2	MR1	MR0	8000h
	06h	Mono Volume	Mute	X	X	X	X	X	X	X	X	X	X	MM4	MM3	MM2	MM1	MM0	8000h
Input Volume	0Ah	PC_Beep Volume	Mute	X	X	X	X	X	X	X	X	X	X	PV3	PV2	PV1	PV0	X	0000h
	0Ch	Phone Volume	Mute	X	X	X	X	X	X	X	X	X	X	GN4	GN3	GN2	GN1	GN0	8008h
	0Eh	Mic Volume	Mute	X	X	X	X	X	X	X	X	20dB	X	GN4	GN3	GN2	GN1	GN0	8008h
	10h	Line In Volume	Mute	X	X	GL4	GL3	GL2	GL1	GL0	X	X	X	GR4	GR3	GR2	GR1	GR0	8808h
	12h	CD Volume	Mute	X	X	GL4	GL3	GL2	GL1	GL0	X	X	X	GR4	GR3	GR2	GR1	GR0	8808h
	14h	Video Volume	Mute	X	X	GL4	GL3	GL2	GL1	GL0	X	X	X	GR4	GR3	GR2	GR1	GR0	8808h
	16h	Aux Volume	Mute	X	X	GL4	GL3	GL2	GL1	GL0	X	X	X	GR4	GR3	GR2	GR1	GR0	8808h
	18h	PCM Out Volume	Mute	X	X	GL4	GL3	GL2	GL1	GL0	X	X	X	GR4	GR3	GR2	GR1	GR0	8808h
ADC Sources	1Ah	Record Select	X	X	X	X	X	SL2	SL1	SL0	X	X	X	X	X	SR2	SR1	SR0	0000h
	1Ch	Record Gain	Mute	X	X	X	GL3	GL2	GL1	GL0	X	X	X	X	GR3	GR2	GR1	GR0	8000h
	20h	General Purpose	POP	X	3D	X	X	X	MIX	MS	LPBK	X	X	X	X	X	X	X	0000h
	22h	3D Control (Read Only)	X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0101h
X	24h	Reserved	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0000h
	26h	Powerdown Ctrl/Stat	EAPD	PR6	PR5	PR4	PR3	PR2	PR1	PR0	X	X	X	X	REF	ANL	DAC	ADC	000Xh
	28h	Extended Audio ID	ID1	ID0	X	X	X	X	0	0	0	0	X	X	0	X	0	VRA	X001h
	2Ah	Extended Audio Control/Status	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	VRA	0000h
	2Ch	PCM DAC Rate	SR15	SR14	SR13	SR12	SR11	SR10	SR9	SR8	SR7	SR6	SR5	SR4	SR3	SR2	SR1	SR0	BB80h
	32h	PCM ADC Rate	SR15	SR14	SR13	SR12	SR11	SR10	SR9	SR8	SR7	SR6	SR5	SR4	SR3	SR2	SR1	SR0	BB80h
X	5Ah	Vendor Reserved 1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0000h
X	74h	Vendor Reserved 2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0000h
X	7Ah	Vendor Reserved 3	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0000h
	7Ch	Vendor ID1	0	1	0	0	1	1	1	0	0	1	0	1	0	0	1	1	4E53h
	7Eh	Vendor ID2	0	1	0	0	0	0	1	1	0	1	0	0	1	0	0	1	4349h

Appendix B: Original AC 97 controller diagram


OPB AC97 Sound Controller
OPB
M U X
Parallel to Serial
16
Playback FIFO
16
Codec Data
Serial to Parallel
16
Record FIFO
16
SData_In
AC97 Codes
SData_Out
AC97 Interface Logic
Parallel to Serial
7
Q D
7
Address
Codec Control/Status Register
OPB Interface Logic
Parallel to Serial
16
Q D
16
Write Data
Serial to Parallel
16
D Q
16
Read Data
AC97 Clock Domain
OPB Clock Domain