ECE532 Digital Hardware

Group Report

Prepared by: Christopher Yip 990917941
Benjamin Lai (991016994)

Table of Contents

1. Overview

This project uses the Xilinx Multimedia board and the MicroBlaze embedded processor to function as a music player similar to Apple's iPod mp3 player. This project was built based on last year's Guitar Project and AC97 controller project to include playback control and storage of recorded audio. The audio format used is 16-bit PCM data.

The main storage medium is the Compact Flash (CF) slot, accessed through the Xilinx System ACE interface. Music can therefore be recorded to and/or retrieved from CF media. Additionally, it also features storage and playback using on-board ZBT RAM in case if compact flash is not present.

The playback control status of the system will be displayed using the RS232 serial CONPORT connection connected to a computer screen. It will tell the user if the system is playing back data, recording data and control audio processing options.

The main goal of this project is to show that it is possible to use the microprocessor to access the Compact Flash hardware to store and retrieve data. Access to the Compact Flash resource increases the number of uses for this board. For example, it can act as a network storage device by incorporating TCP/IP functionalities in the microblaze soft processor.

2. Outcome and Problems Encountered

2.1 Outcome

Overall, we are pleased with the outcome of the project. The audio record/playback function works as expected, thanks to the excellent documentations provided by last year's students as well as information found on the course website.

We were able to record and playback successfully onto the ZBT RAM and implement playback control using the keyboard. System status is displayed using the RS232 interface through a dummy terminal running on a PC. Recording and playback onto onboard RAM was to mitigate the event that the Compact Flash did not function.

We were able to read and write data to the Compact Flash card. However it was not as transparent as we had thought since the XilFatFS API call `sysace_fwrite()` was not functioning. As a result we resorted to writing one sector at a time using the commands `sector_read()`and `sector_write()`, which prevented the written data to be read by other devices such as a Windows PC. We were also not able to access the entire card, perhaps due to reserved sectors in the file system that are not accessible using the API calls we chose to use. As a result, we could only store about 1MB of data onto a 16MB Compact Flash card.

2.2 Problems Encountered

2.2.1 Compact Flash Issues

The Xilinx EDK includes software to enable access to the Compact Flash card through the System ACE interface. The software library is called the XilFatFS (Xilinx FAT File System) library. It contains several API calls which are 'equivalent' to standard C file access functions such as `sysace_fwrite()` and `sysace_fread ()` functions. We were able to succesfully create a file name on the compact flash and see the file appear in a windows machine. However the `sysace_fwrite()` functions did work on the hardware, constantly freezing during operation. We were not able to write data into the file on the compact flash. There was not enough time to debug the problem further. The problem was traced down to a subroutine called by the `sysace_fwrite()` function that updates the directory listing. This could be a suggested future work topic for students next year to get the FAT FS commands to function.

Additionally, the System ACE chip requires very particular formatting of the compact flash. The following answer record was found in the Xilinx wbsite which talks about formatting CF cards for use on the board: http://support.xilinx.com/xlnx/xil_ans_display.jsp?iLanguageID=1&iCountryID=1&getPagePath=14456

This method did not work for our board. However, we had Canon digital cameras that uses compact Flash cards and that allowed us to properly format the cards.

Instead we were able to demonstrate that we were still able to read and write to the compact flash on a sector-by-sector basis. So the compact flash was more like the ZBT RAM where it is a device with a mapped memory space that we can write to. This function works well, except for the fact that the data written was not readable with a PC connected via a USB adaptor. The original idea was to use the PC to copy audio to the Compact Flash card and use the Multimedia board to play it back. Alternatively, the user could record their voice or anything else they wanted to onto the compact flash, and then copy that onto their PC for archiving.

We also did not expect to the System ACE controller to transfer data to the Compact Flash so slowly. Instead of directly reading/writing to the card, all the data had to be buffered in advance.

2.2.2 User Defined Core Issues

Initially we intended to incorporate several audio-processing features, implemented on hardware, such as bass enhancement and vocal enhancement. These audio filters were to be implemented as FIR (Finite Impulse Response) filters generated by CoreGen. CoreGen is an IP (intellectual property) customization tool that contains many commonly used components such as adders, memory blocks and DSP blocks. The coefficients for the filters were derived using simulations from MatLab (more on this later). The cores worked very well in simulation within Modelsim. However they didn't respond when implemented in hardware in the FPGA. Due to time limitations we were unable to

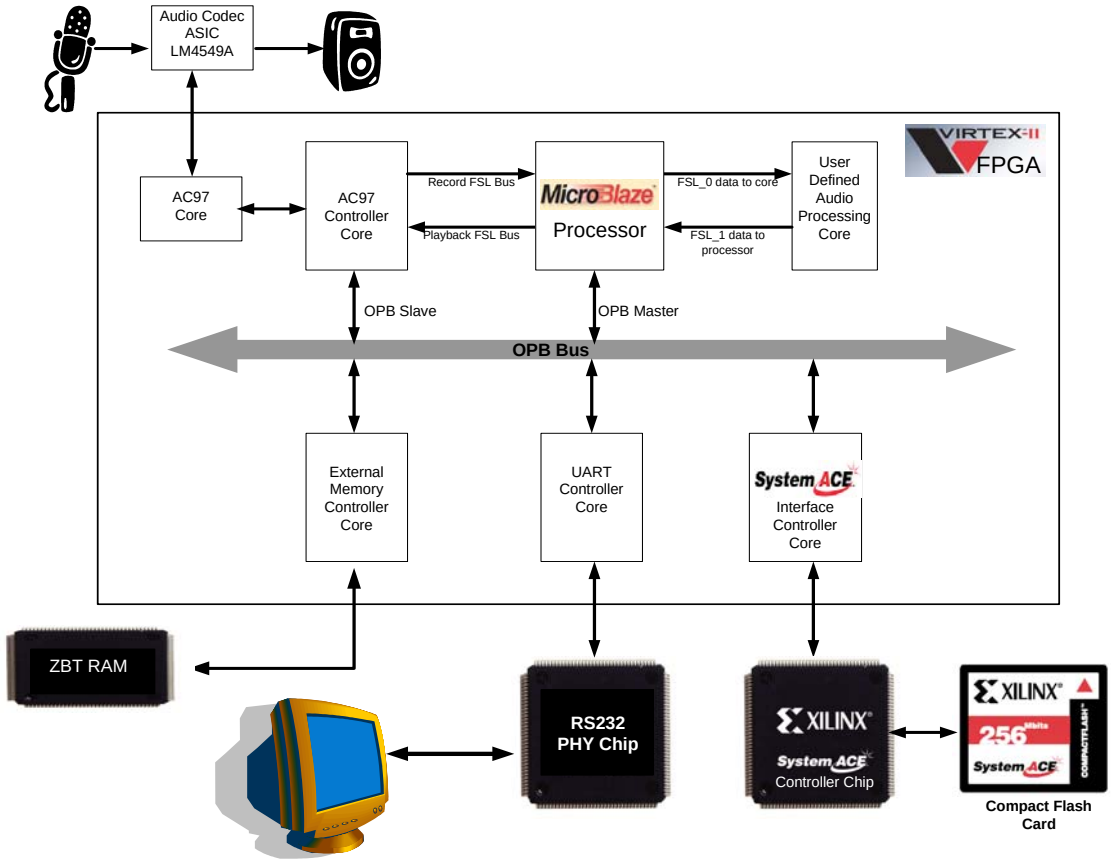
incorporate the FIR core into the system. A full description of this core and the simulation results are described in section 5.

To show that we knew how to integrate a user-defined core to our microprocessor through the FSL, we made a simpler core which performs a phase delay effect. Essentially it is a fifo that delays the audio samples, thus adding very small echo effect.

3. Description of the Blocks

3.1 Hardware Blocks and System Block Diagram

The figure below shows the overall system block diagram of the Xpod music player. The main playback control is achieved in software, which calls the appropriate hardware drivers to playback audio, record audio or stop.



3.1.1 AC97 CODEC and Audio Controller (v3.01a):

The AC97 controller allows the Microblaze soft processor to interface with the AC97 audio CODEC chip LM4549A. We had initially tried using the audio controller from last year's MP3 lab to record. However it did not seem to work properly. As a result controller was taken from last year's AC97 project and used it as the foundation to implement the additional hardware and software to make the music player.

For more information regarding how to setup audio using this enhanced controller, please refer to the document entitled "AC97 Sound Controller with Device Driver" located at http://www.eecg.toronto.edu/~pc/courses/432/2004/projects/ac97controller.doc.

The AC97 CODEC is an ASIC manufactured by National Semiconductors. Its datasheet can be found here: http://www.national.com/pf/LM/LM4549A.html. It converts analog audio to PCM coded data and vice versa.

3.1.2 UART Core (v1.00b):

This core is used to interface with the RS232 physical layer (PHY) chip to enable status reporting to a computer screen and user interaction. For example, it will display the words "Recording" when data is being recorded. User input is collected via the RS232 interface from the keyboard. Depending on the key pressed the software ware will behave differently. More details about the software user interface can be found in the software section of this report.

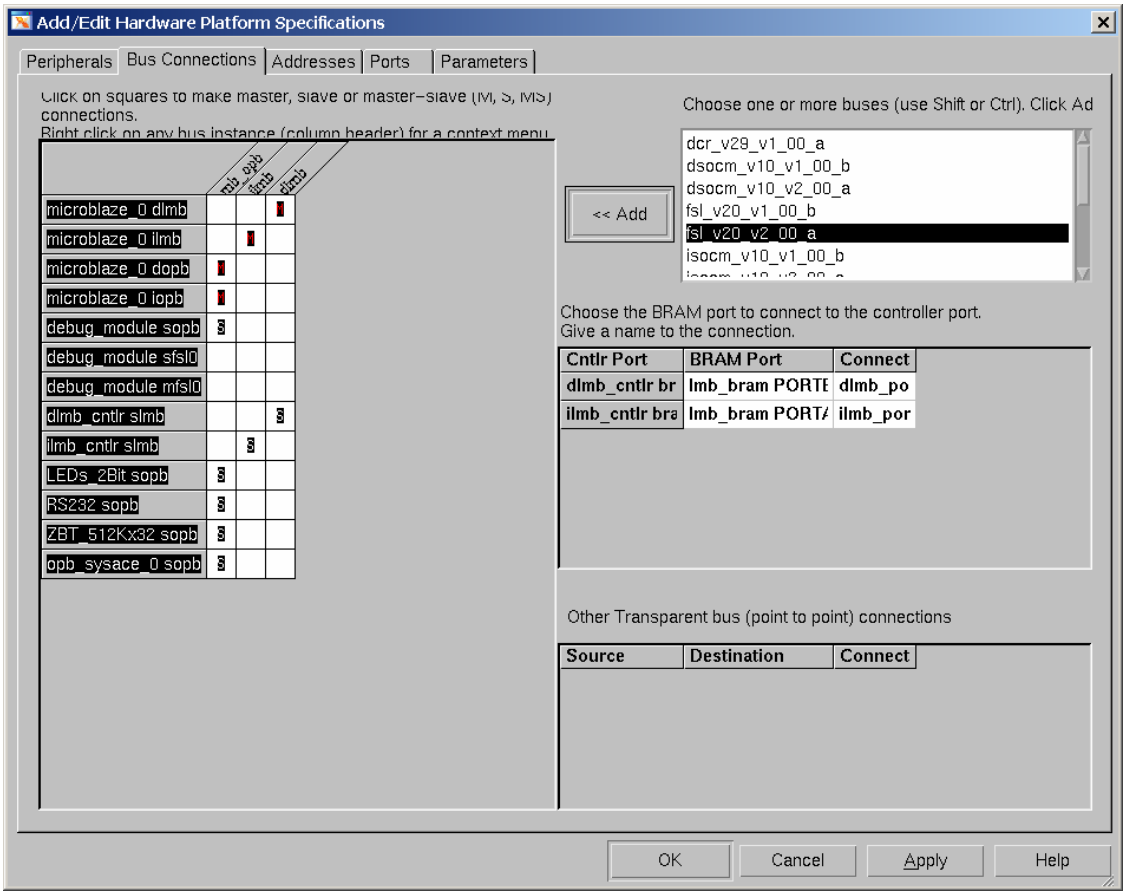
Incorporating RS232 functionality can be included in the design during the initial system within the BSB (Base System Builder) wizard or add it in later on within the "Add/Remove Cores" dialog in XPS. The stdio.h library will be copied to the appropriated directories to allow standard commands such as `xil_printf()`, which functions the same as the standard C function `printf()`.

3.1.3 FSL 'BUS' (v2.00a):

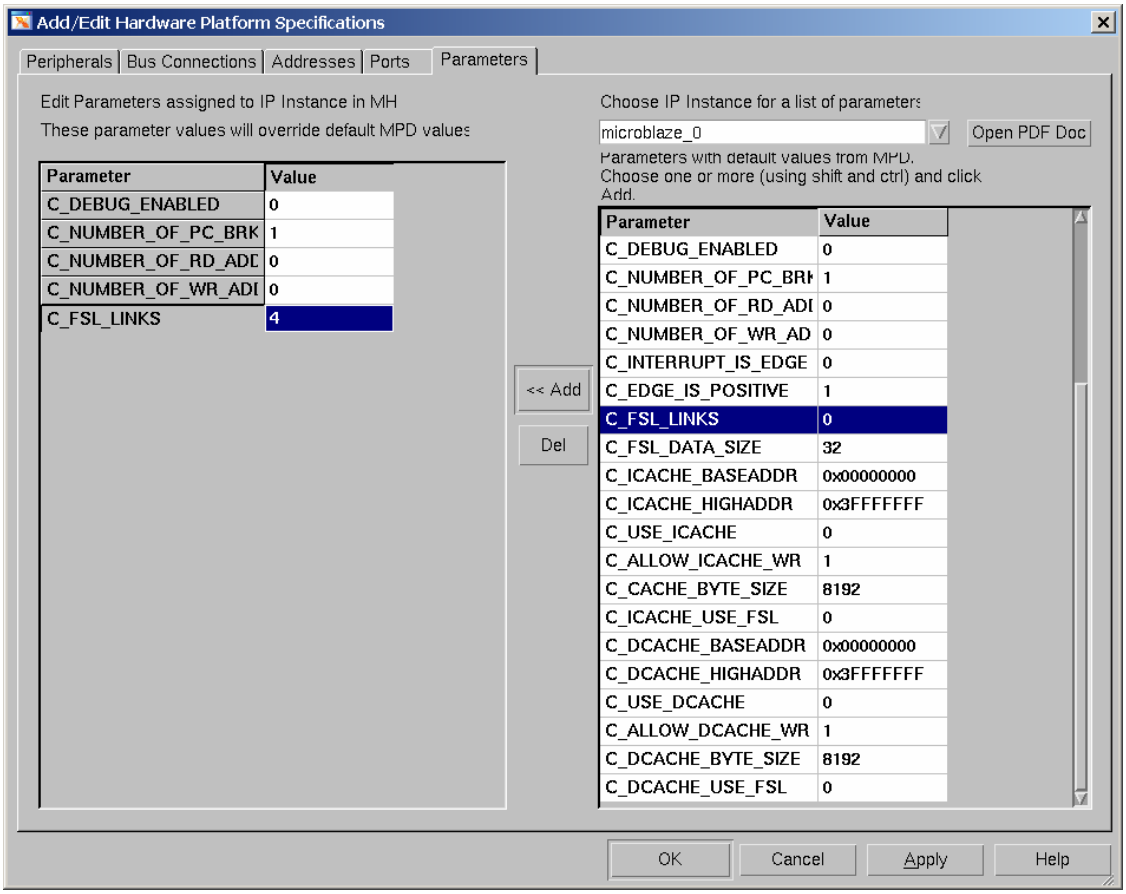
FSL stands for Fast Simplex Link. Essentially the FSL is a high speed FIFO for direct oneway links between hardware cores and the MicroBlaze processor. Upto 8 FSL busses can be used in a given system, giving the designer great flexibility. Because the GIPO bus is already populated with 3 cores that constantly send data back and forth, bandwidth on this bus will therefore be limited. It is unlikely that we can fit audio playback and recording abilities on this bus and still maintain other functions. Therefore, a dedicated FSL link was established to interface the MicroBlaze processor with the AC97 CODEC to transfer PCM encoded audio data. Since the FSL bus is unidirectional, two links were used for the AC97 CODEC. One is dedicated for receiving audio data from the CODEC and the other is used exclusively for sending out audio data.

Additionally, two more FSL links were used to interface the MicroBlaze with the audio processing core. Data is sent via the first FSL bus to the core for audio processing and retrieved via the second FSL bus. This allows high-speed transfer of data between the processor and the core since we can simultaneously send and receive data similar to duplex communication.

Setting up the system to use multiple FSL busses is very straightforward. Again this can be done within XPS's GUI or manually by editing the MHS configuration file. Within the GUI, multiple FSL buses can be added by going into the "Add/Remove Cores" dialog and going to the "Bus Connections" tab. Add in the desired number of FSL instances on the selection box on the right.



Finally, go to the "Parameters" tab and set the C_FSL_LINKS to the number of FSL buses that will be used in the system.



Within the C Programming environment, Xilinx has provided four API calls to access the FSL through blocking and non-blocking functions. Application note #529 (located at: http://www.xilinx.com/bvdocs/appnotes/xapp529.pdf) shows an example of how to integrate a user-defined core, which include cores generated within CoreGen, into a MicroBlaze system built within XPS. This application note was used to integrate our core into the MicroBlaze system. It contains instructions on how to write the appropriate wrappers to integrated the core with the FSL.

3.1.4 OPB System ACE Controller (v1.00b):

The System Advanced Configuration Environment (System ACE) was designed to configure the FPGA at power-up. It has several options for storing configuration data. Storage media that can be used are: standard Compact Flash memory cards (types I and II) and IBM Microdrives (upto 8Gb in capacity).

We are taking advantage System ACE compact flash interface and used it to store data used by the software. The System ACE interface provides an external storage solution making it an excellent choice for this project because of its requirement for large amounts of storage.

The OPB System Ace Controller is an IP core supplied by Xilinx that acts as a bridge between the microprocessor and the actual System ACE controller chip that also resides on the multimedia board. The System ACE controller chip in turn interfaces with the compact flash card for reading and writing data. This core is included as a list of available cores that can be instantiated within XPS in the "add/remove cores" dialog box.

The XilFatFS library is available as an API for use that provides file read/write capabilities. For more information on this API please refer to chapter 8 of the EDK OS and Libraries Reference Manual located here: http://www.xilinx.com/ise/embedded/oslibs_rm.pdf.

The following is a list of other useful documents that were referenced when we were researching the System ACE interface:

System ACE White Paper – Introduces the System ACE CF storage option (page 3):
http://www.xilinx.com/bvdocs/whitepapers/wp151.pdf

System ACE CF datasheet – Contains port descriptions and timing information:
http://direct.xilinx.com/bvdocs/publications/ds080.pdf

Main documentation page for the OPB_Sysace IP core:
http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=opb_sysace&iLanguageID=1

System ACE main page:
http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=system_ace

3.1.5 External Memory Controller (1.10b):

To mitigate the risk of not being able to fully access the compact flash resource to store data, we also implemented data storage using the onboard ZBT RAM memory blocks. This memory controller, when instantiated within XPS, allocates a block (or several blocks depending on how much is needed) of memory addresses to allow access to the ZBT RAM. The lower and upper limit of the memory space that represents the ZBT RAM can be found in the MHS file or in the "Address" tab of the "Add/Remove cores" dialogue box.

This feature was based on tutorial module 8 (version 6.3) from the ECE532H1S website. Please refer to http://www.eecg.toronto.edu/~pc/courses/edk/modules/6.3/m08.pdf for more information.

3.1.6 MicroBlaze Embedded Processor (v3.00a):

The Xilinx MicroBlaze soft processor is the main control unit of the whole system. It is responsible for writing and retrieving sound information from the CF and data processing during playback. Reading and writing data is accomplished using the XilFatFS library that utilizes the OPB_SysACE.

The embedded processor is also responsible for providing a user interface for playback control. It provides the current state of operation of the music player - such as recording and playback. Communications with the user is accomplished using the RS232 interface on the physical layer. The MicroBlaze soft processor interfaces with the UART controller (which in turn interfaces with the RS232 Physical layer driver) via the GPIO (General Purpose IO) bus.

The MicroBlaze system is created using Xilinx's EDK (Embedded Development Kit), particularly within XPS. It is a 32-bit RISC processor embedded within the FPGA. As for the Xilinx Multimedia board, the MicroBlaze processor resides in the Virtex-II FPGA device. Upto four soft processors can be instantiated.

All information regarding MicroBlaze can be found at its homepage in the Xilinx Website: http://www.xilinx.com/microblaze.

4. Description of the Software

The following section describes the following code blocks used in the project.

1. User interface C code
2. Audio record/playback subroutine
2. C code to read/write to ZBT
3. C code to read/write to compact flash card
4. C code to read/write audio through AC97
7. C code for audio processing (fast forward, slow down)

The executable is stored in ZBT RAM because it is too big to fit in the on-chip BRAM.

4.1 User Interface

The main user interface software resides in the is written in C. According to the input from the user, this code executes helper functions to accomplish the desired task. The following keys are used to control the functionality of the user interface:

r: to start recording audio, calls recording subroutine
p: play back whatever was recorded, calls playback subroutine
s: stop recording or stop playing
s: copy recorded data to Compact Flash

l: copy data from CF to memory

options during playback:
1: no effect
2: slow down / low frequency effect
3: fast forward / high frequency effect
4: echo effect

Prompts are displayed through the Xilinx COM port through the RS232 interface to indicate the operational mode and status of the system.

4.2 Audio Record/Playback from AC97 and save to ZBT Subroutine

The audio record and playback calls up the fsl block read and block write functions to send and retrieved audio data from the AC97 CODEC. Once the data is retrieved, it is immediately saved to ZBT Ram. Currently, one ZBT RAM chip is used, allowing for upto 2MB of audio storage.

```
void record()
{
        Xuint16 soundbyte;

        sound = (Xuint16*)MEM_BASE_ADDR;
        printf("Recording\n\r" );
        while( sound < (Xuint16*)MEM_TOP_ADDR &&
                XUartLite_mIsReceiveEmpty(XPAR_RS232_BASEADDR) )
        {
                microblaze_bread_datafsl(soundbyte, 0);
                *sound = soundbyte;
                sound++;
        }
}
```

4.3 Compact Flash card Reading/Writng subroutine

The user has a choice to copy the data from ZBT RAM to Compact Flash for long term storage. The code reads/writes one sector of the Compact Flash card at a time using the function write_sector() which comes as a XilFatFS library. Data retrieval is done using the read_sector() function, also from the XilFatFS library. To learn how to set up Compact Flash reading and writing, please see the Appendix.

```
const int number_of_sector = 1300;
const int sector_size = 1024;
```

```
void saveCF()
{
        int i;
        printf("Saving to CF\n\r");
        sound = (Xuint16*)MEM_BASE_ADDR;
        for( i = 0; i < number_of_sector; i++ )
        {
                write_sector( i+1, (BYTE*)sound );
                sound = sound + sector_size/4;
        }
        printf("\r Done.\n\r");
}

void loadCF()
{
        int i;
        printf("Loading from CF...\n\r");
        sound = (Xuint16*)MEM_BASE_ADDR;
        for( i = 0; i < number_of_sector; i++ )
        {
                read_sector( i+1, (BYTE*)sound );
                sound = sound + sector_size/4;
        }
        printf("\r Done.\r\n\n");
}
```

Since we used a digital camera to format the CF, we had no control of the number of sectors and the size of each of the sector. Therefore, *number_of_sector* and *sector_size* are derived experimentally with the multimedia board. In our case, we found that we could continuously write to 1300 sectors, and each sector contains 512 bytes.

4.4 Software Audio Processing Subroutine

Two software audio processing were implemented. The first speeds up playback by 2x. This is accomplished by playing back every other audio samples. For example, if there are 10 samples of audio data, only the 1st, 3rd, 5th etc. sample will be played. This effectively speeds up playback. The second function slows down playback by a factor of 2. To accomplish this, each audio sample is played twice.

```
void slowDown()
{
        Xuint16 soundbyte;
        printf("low freq\n\r");
        while( sound < (Xuint16*)MEM_TOP_ADDR &&
                XUartLite_mIsReceiveEmpty(XPAR_RS232_BASEADDR) )
```

```
        {
                soundbyte = *sound;
                microblaze_bwrite_datafsl(soundbyte, 0);
                microblaze_bwrite_datafsl(soundbyte, 0);
                sound++;
        }
}

void fastForward()
{
        Xuint16 soundbyte;
        printf("high freq\n\r");
        while( sound < (Xuint16*)MEM_TOP_ADDR &&
                XUartLite_mIsReceiveEmpty(XPAR_RS232_BASEADDR) )
        {
                soundbyte = *sound;
                microblaze_bwrite_datafsl(soundbyte, 0);
                sound+=2;
        }
}
```

5. Description of the User Defined Hardware

5.1 FIR Filter for Audio Filtering

A Distributed Arithmetic Filter was chosen to do audio filtering. This is generated using CoreGen software that comes with ISE. We first attempted to do a bass boost function. The bass boost was designed to have gain from 0KHz to 5KHz. The coefficients of the filter is derived using MatLab and the frequency response is given in the figure below:

<insert frequncy response plot>

The impulse response, which will be used to simulate the filter is given in the following:

The coefficients are:

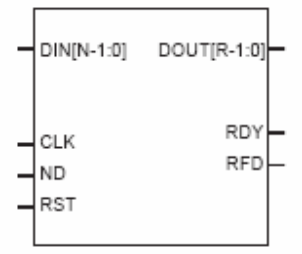
```
-3, -5, -6, -6, -2, 3, 10, 18, 25, 29, 31, 29, 25, 18, 10, 3, 2, -6, -6,-5, -3
```

We are using V9.0 of the core and its data sheet can be found at the following url: http://www.xilinx.com/ipcenter/catalog/logicore/docs/da_fir.pdf.

The filter is configured to the following specifications:
• Single Rate Filter configuration

• 21 taps, symmetric
• input data width 16
• registered outputs (which means that the output value will remain there until the next data is ready)

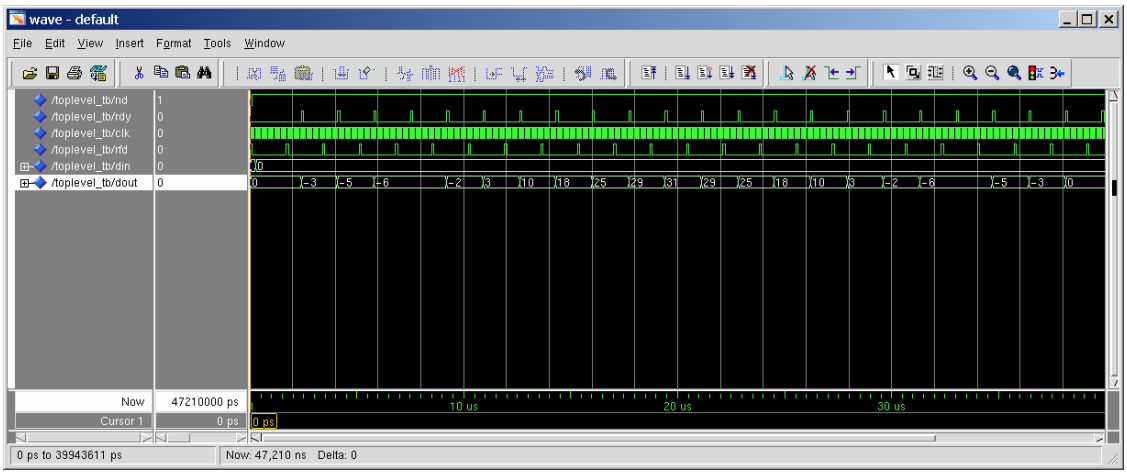
The input and output ports are as follows:



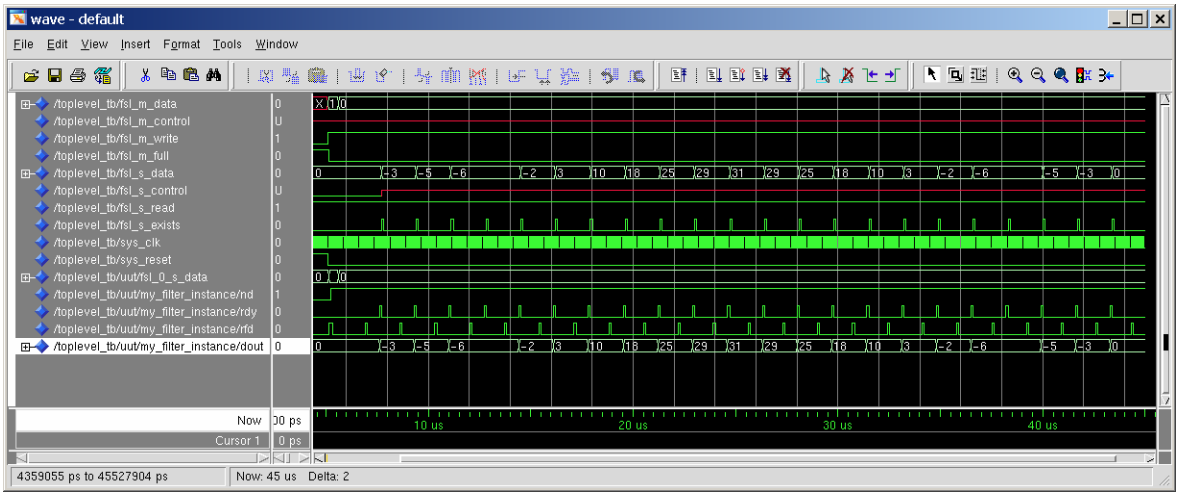

```
Inputs:
DIN -> Data In
CLK -> Clock
ND  -> New Data (for the core) active high
RST -> Reset (active hgh)

Outputs:
DOUT -> Data Out
RDY  -> Output Data Ready
RFD  -> Ready for (new) data
```

CoreGen creates the VHDL file and netlist after the core has been configured. The VHDL code is instantiated in a top level test wrapper in ISE for simulation. ISE was chosen for simulation becuase it has the graphical test-bench generator and automatic ModelSim script generation.

During simulation, a 1 is applied followed by a chain of zeros. The output should be the same as the values in the impulse response. We were able to confirm this by simulating the core alone and have the output show the impulse response values:



Next the filter is integrated with two FSL buses as show in the system block diagram and the simulation results also match the impulse response:



5.2 Alternative Hardware Audio Core -- Echo

To show that we're not stupid and we can actually get a core working, we've decided to use a different core that was not generated by CoreGen. Instead of using a FIR filter we replaced the FIR with a FIFO to delay audio data to create an echoing effect. The audio samples are passed into the fifo. The data that pops back out of the FIFO is therefore audio from an earlier time. This data is combined with the audio that is currently being played back to create the echoing effect.

6. Description of the Design Tree

The structure of our design is described below:

system.mhs: hardware configurations

system.mss: software configurations

data/system.ucf: user constraint file, specifications of pin connections

drivers/ac97_v1_00_a: drivers for the ac97 sound controller

pcores/.

- clk_align_v1_00_a
- echo_v1_00_a : adds echo effects to sound
- fir_filter_v1_00_a :
- fsl_v20_v1_00_b
- gen_zbt_addr_v1_00_a
- opb_ac97_controller_v3_10_a

code/.

system.c : contains code for UI, sound processing, and interact with echo core and ac97 controller

sysace_test.c : code that we used to test the compact flash with the xsysace driver.

simulation/.

fir_sim : simulation of the fir filter

fsl : simulation of the FSL

APPENDIX 1: Step-by-Step instructions on how to Setup the System ACE Compact Flash hardware for reading and writing data

Assumptions:
• A base system has already been configured manually or with the BSB
• Have read through the documentation presented in section 3.1.5
• Interrupts are not used with the System ACE

Step 1: Instantiate the OPB_Sysace into the MicroBlaze system

This is done by going into Project --> Add/Remove Cores. In the "Peripherals" tab (the first one), add in the opb_sysace peripheral. This is the controller core that will interface with the external System ACE chip. Do not close the dialog yet.

Step 2: Connect the OPB_Sysace to the MicroBlaze system

In the second tab called "Bus Connections", the opb_sysace peripheral will be listed as an item to be connected to the system bus. Connect this as a slave on the OPB.

Step 3: Assign an address location for the OPB_Sysace

In the third tab called "Addresses", you will need to define an address location for the opb_sysace peripheral. The minimum size is 64k. You can define any address that is currently not occupied by any other bus peripheral. You can let XPS do the job for you by first checking the "lock" box for all other peripherals, then hitting the "Generate Addresses" button on the bottom of the dialog box.

Step 4: Add the OPB_Sysace ports into the MicroBlaze system

In the "Ports" tab, scroll along the peripherals list until you see the listing for opb_sysace_0 (the instance name). Underneath the name are the available ports for this peripheral. Select all the ports except for the one called `SysACE_IRQ` or `SysACE_MPIRQ` since these relate to interrupt features which we are not using.

Connect the OPB_clk and SysACE_clk signals to sys_clk_s, the system clock. Select all other remaining ports and press the "Make External" button on the right.

Step 5: Set data bus width for OPB_Sysace

In the last tab, the "Parameters" tab, use the drop down menu and choose the OPB_sysace entry. A parameter called C_MEM_WIDTH is displayed. Depending on how wide your data bus is, you may need to change this value. For example, if you are using 8-bit data, then change the width to 8. More information can be found from the data sheet by clicking on the "Open PDF Doc" button.

Step 6: Save the new hardware configuration data into the MHS file

Hit the OK button to save.

Step 7: Update the UCF (user constraint file) to make connections between the FPGA and the System ACE chip

The FPGA pins are used for mapping. You can copy this into your UCF file.

```
Net opb_sysace_0_SysACE_MPA<0> LOC = AJ1;
Net opb_sysace_0_SysACE_MPA<1> LOC = AF4;
Net opb_sysace_0_SysACE_MPA<2> LOC = AG3;
Net opb_sysace_0_SysACE_MPA<3> LOC = AK2;
Net opb_sysace_0_SysACE_MPA<4> LOC = AE8;
Net opb_sysace_0_SysACE_MPA<5> LOC = AF9;
Net opb_sysace_0_SysACE_MPA<6> LOC = AH5;
Net opb_sysace_0_SysACE_MPD<0> LOC = AE3;
Net opb_sysace_0_SysACE_MPD<1> LOC = AD6;
Net opb_sysace_0_SysACE_MPD<2> LOC = AD7;
Net opb_sysace_0_SysACE_MPD<3> LOC = AF1;
Net opb_sysace_0_SysACE_MPD<4> LOC = AG1;
Net opb_sysace_0_SysACE_MPD<5> LOC = AD4;
Net opb_sysace_0_SysACE_MPD<6> LOC = AE4;
Net opb_sysace_0_SysACE_MPD<7> LOC = AD8;
Net opb_sysace_0_SysACE_MPD<8> LOC = AE7;
Net opb_sysace_0_SysACE_MPD<9> LOC = AG2;
Net opb_sysace_0_SysACE_MPD<10> LOC = AH2;
Net opb_sysace_0_SysACE_MPD<11> LOC = AD5;
Net opb_sysace_0_SysACE_MPD<12> LOC = AE5;
Net opb_sysace_0_SysACE_MPD<13> LOC = AC9;
Net opb_sysace_0_SysACE_MPD<14> LOC = AD9;
Net opb_sysace_0_SysACE_MPD<15> LOC = AH1;
Net opb_sysace_0_SysACE_CEN LOC = AH6;
Net opb_sysace_0_SysACE_WEN LOC = AJ4;
Net opb_sysace_0_SysACE_OEN LOC = AK4;
```

The system is now ready to read and write to the Compact Flash.

APPENDIX 2: Step-by-Step instructions on how to Setup software to achieve basic Compact Flash read and write functions

Step 1: Set up the XilFatFs library for use in XPS

In the XPS environment, go to Project --> Software Platform Settings. At the bottom of the "Software Platform" tab, there is a list of API libraries that you can include. Check the one called XilFatFs. Click OK to exit the dialog box. Finally go to Tools --> Generate Libraries and BSPs. This will update your MSS file and copy the necessary header files to your "include" directory in your project folder.

Now the software environment is ready to be used.

Step 2: Setting up Initiating the System ACE in your code

At the top of the file you will need to declare the following line

#include <sysace.h>

Several function calls are required to properly initialize the System ACE. The following is an example of what you need to do to initialize the code to run System ACE. This is the same code used in our project:

```
int main()
{
        …
        init_ace();
        …
}
```

Step 3: Writing Data to the Compact Flash

Since we were not able to use the API calls described in chapter 8 of the following document http://www.xilinx.com/ise/embedded/oslibs_rm.pdf, we used the lower level API calls to read and write to the Compact Flash card. The functions are sector_read () and sector_write() to send and receive data from the Compact Flash Card. It is best illustrated using an example:

```
int main()
{
        const int number_of_sector = 1300;    //number of sectors on CF
        const int sector_size = 1024;    // number of bytes in each sector
        BYTE data[sector_size];
```

```
        …
        init_ace();
        …
        write_sector( 1, data );
        …
        read_sector( 1, data );
}
```

A few noteworthy items:

1. When writing sectors, it is important to know the size of each sector so that you can properly write data to the compact flash.
2. Make sure the compact flash is properly formatted