Project Report for Embedded Systems Design 4840

MANIC

Music All Night In Columbia

Implementation of ADPCM decoding system using Xilinx SpartanIIE, Microblaze and other peripherals

Team:

Prakash G.S

pg2132@columbia.edu

Devyani Gupta

dg2168@columbia.edu

Vijayarka Nandikonda

vn2107@columbia.edu

11th May, 2004

Contents

1. Introduction

The original goal of our project was to design an MP3 player. However, we have revised this goal towards implement an ADPCM player owing to time constraints and numerous problems faced with respect to implementation. In addition, we also endeavor to implement a PCM player for 8 and 16 bit audio formats.

What is Pulse Code Modulation?

A common method of representing continuous analog values in digital form is pulse-code modulation, or PCM. In PCM, distinct binary representations (pulse codes) are chosen for a finite number of points along the continuum of possible states. When-ever the value is being measured and it falls between two encoded points, the code for the closer point is used.

This process is called quantization: the dividing of the range of values of a wave into sub ranges, each of which is represented by an assigned value. A series of these pulse codes can be transmitted in a pulse train, resulting in a pulse-code modulated signal. Because the samples of digitized speech referred to above are stored in the form of digital pulses, the stored waveform can be thought of as an example of pulse-code modulation.

What is Adaptive Differential Pulse Code Modulation?

ADPCM is an audio coding technique that is widely used throughout the telecommunications industry. It works by calculating the difference between two consecutive samples in standard pulse code modulation (PCM) and codes the error of the 'predicted' next sample increment (from the previous sample increment) to the true sample increment.
It is a lossy compression technique that achieves a compression ratio of 4:1. However, it is popular since is returns a high quality signal with very little processing power required for fast decoding.
There are primarily 2 different industry formats for ADPCM:

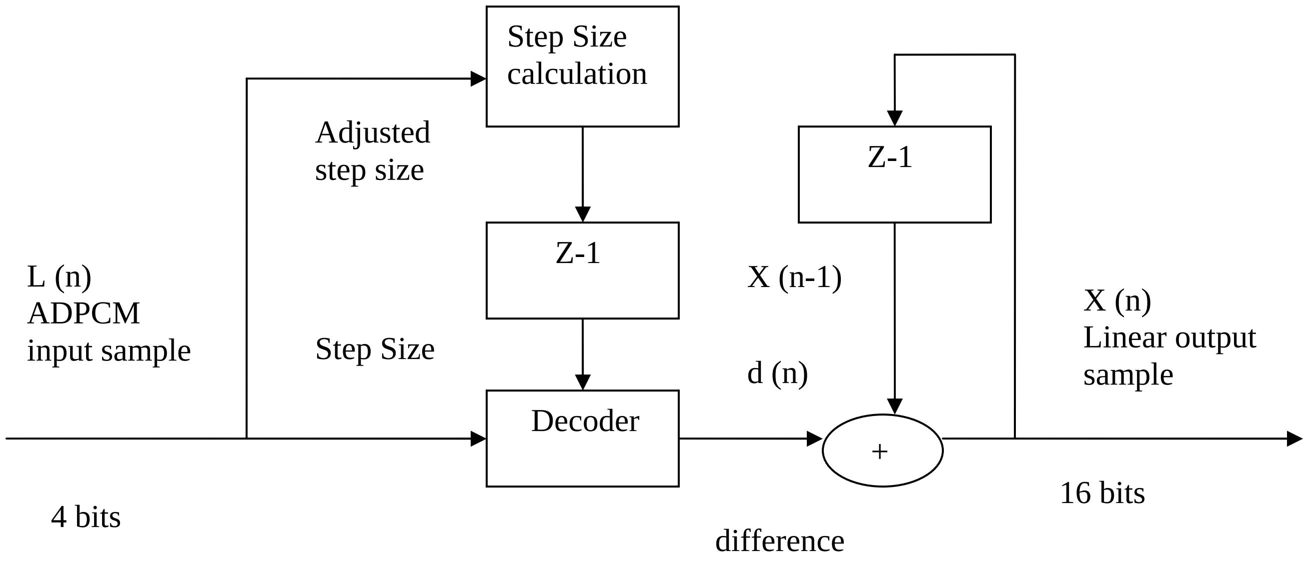
1. IMA/DVI ADPCM
2. Microsoft ADPCM

Our implementation is centered on the IMA format.

2. ADPCM Decoding

2.1 Algorithm

The diagram shows the steps involved in ADPCM decoding.



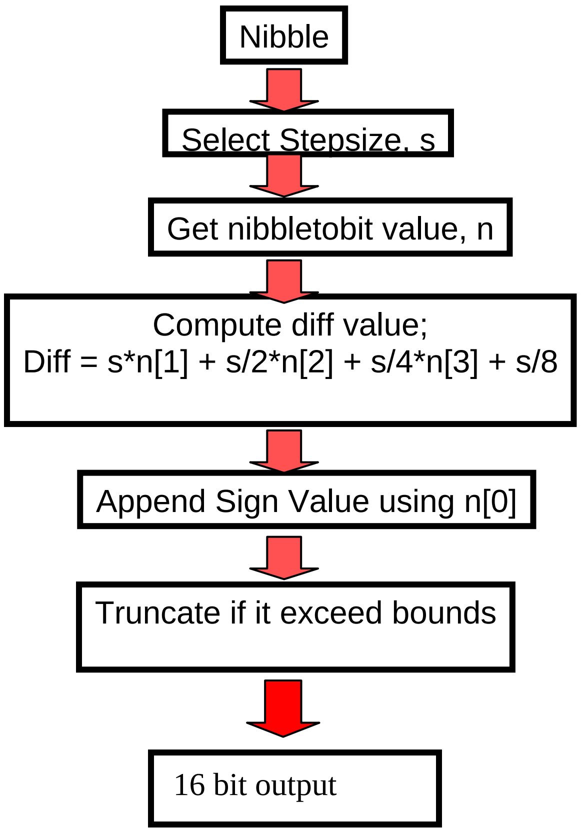
ADPCM decoding is comprised of the following steps:

Step-Size calculation:
The step-size is basically a coding scale for the ADPCM. It varies dynamically to accommodate the differences between small and large samples. The step size initially starts off at a preconfigured value. This value is then readjusted/predicted for the next sample, depending on the sample received. This process is called ***step-size adjustment***.

Decoding:
The decoding for each 4 bits then happens by using the current sample and the step-size. The difference of the decoded output and the previous sample is then taken. This difference yields a 16-bit linear PCM sample.

2.2 Implementation

The following flow chart defines our implementation. We maintain 3 arrays to store stepsize values, nibbletobit values and sign value. We also have an array to store the value of the last 128 samples.


Nibble
Select Stepsize, s
Get nibbletobit value, n
Compute diff value;
Diff = s*n[1] + s/2*n[2] + s/4*n[3] + s/8
Append Sign Value using n[0]
Truncate if it exceed bounds
16 bit output

ADPCM decoding sequence

2.3 Calculation of Step Size

For both the encoding and decoding process, the ADPCM algorithm adjusts the quantizer stepsize based on the most recent ADPCM value. The step size for the next sample, n+l, is calculated with the following equation:
ss(n+1) = ss(n) * 1.1M(L(n))

This equation can be implemented efficiently as a two-stage lookup table. First the magnitude of the ADPCM code is used as an index to look up an adjustment factor as shown in *Table 1*. Then that adjustment factor is used to move an index pointer in *Table 2*. The index pointer then points to the new step size. Values greater than 3 will increase the step size. Values less than 4 decrease the step size.

Table 1. `M(L(n)) Values

L(n)	Value	M(L(n))
1111	0111	+8
1110	0110	+6
1101	0101	+4
1100	0100	+2
1011	0011	-1
1010	0010	-1
1001	0001	-1
1000	0000	-1

Table 2. Calculated Step Sizes

No.	StepSize	No.	StepSize	No.	StepSize	No.	StepSize
1	16	13	50	25	157	37	494
2	17	14	55	26	173	38	544
3	19	15	60	27	190	39	598
4	21	16	66	28	209	40	658
5	23	17	73	29	230	41	724
6	25	18	80	30	253	42	796
7	28	19	88	31	279	43	876
8	31	20	97	32	307	44	963
9	34	21	107	33	337	45	1060
10	37	22	118	34	371	46	1166
11	41	23	130	35	408	47	1282
12	45	24	143	36	449	48	1411
49	1552						

This method of adapting the scale factor with changes in the waveform is optimized for voice signals, not square waves or other non-sinusoidal waveforms.

3. Design Alternatives

ADPCM implementation can be done in

- Software (in a programming language) like C or
- In hardware (in a hardware-description language) like VHDL.

We explored both alternatives, finally settling on the software approach. This was due to the following reasons:

1. The computations involved in the decoding were not of a time-consuming nature. So implementing in hardware would have given no substantial benefit.

   MicroBlaze runs at a clock frequency of 50 MHz, and our input ADPCM file is of 12 kHz. So we have 50M/12k = 4000 cycles for processing each sample. This gives us ample time to implement the algorithm in software.

2. There are no floating-point computations in the procedure.

MicroBlaze cannot do floating point computations. If there were floating point computations in the process, we would have had no choice but to do it in VHDL.

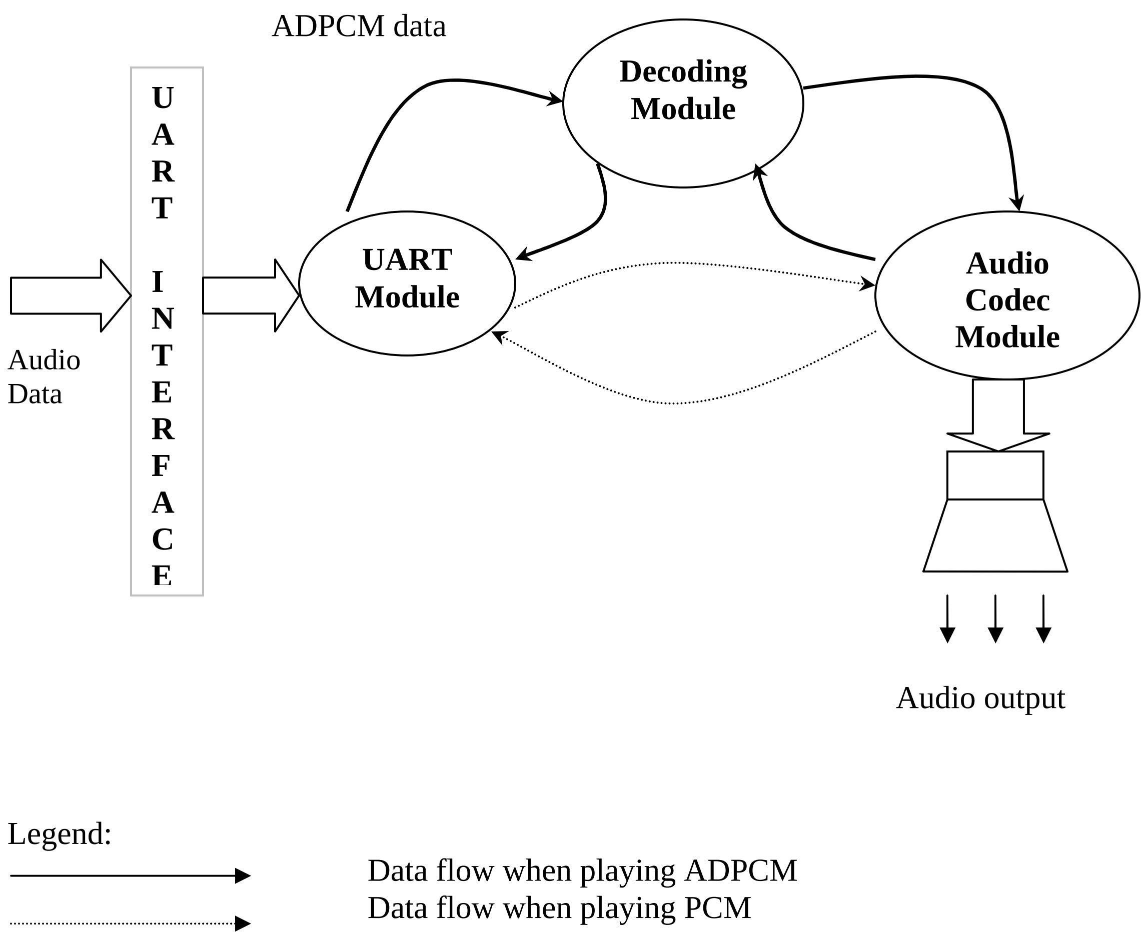
3. The procedure involves few simple multiplication operations (step-size x 1-bit) and these can be optimized in software into simple conditional statements with additions.

   Timing is the most crucial thing in high speed I/O. Code optimization reduced the time taken to process the samples, thus obeying the time constraints.

4. Final Design

Our design essentially composes of 3 modules, namely the UART module, the Audio Codec module and the Decoding module.

The diagram below shows interaction among the various modules.



4.1 Platform and Peripherals

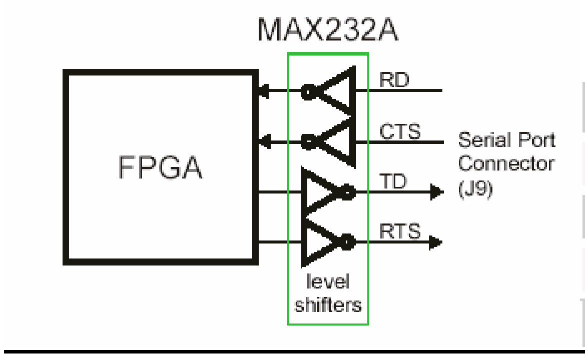
1. Xilinx SpartanIIE FPGA on the XSB Board(XC2S300E-SPQ208C)
2. Serial Port
3. Stereo Codec(AK4565)
4. 4Mbit SRAM(TC55V16256FT-15)

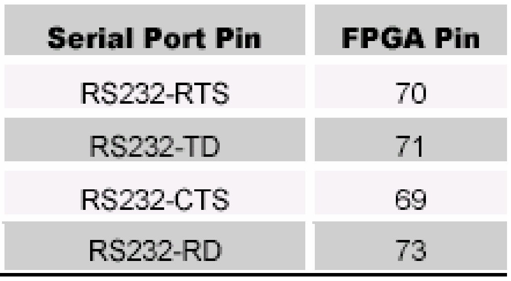
Below is the description of peripherals we used on the XSB Board

4.1.1 Universal Asynchronous Receiver Transmitter

This module is used to load PCM/ADPCM data from a workstation over the serial port. We extended the uart module which was part of lab 3. Audio data to be decoded was stored in a *uart application buffer* and fed to the decoding module. However, PCM data was streamed to the audio codec, without much processing, except for converting it from mono to stereo.
One feature of our UART module was that we had designed it for ***flow control***. This feature enabled us to communicate with the test232 program on the workstation to alternately start and stop flow of audio data as and when the *uart application buffer* was tending towards empty or full.

The FPGA handles the interface to the serial port. The four active lines of the serial port connect to the FPGA as follows.





Serial Port Pin	FPGA Pin
RS232-RTS	70
RS232-TD	71
RS232-CTS	69
RS232-RD	73

4.1.2 UART Configuration

We configured the UART to operate at a speed of 115200 bauds. This was just enough to play an 8 bit 12 Khz PCM WAV correctly. But the output for a 16 bit wav is strained. To play a 16 bit 12 Khz PCM WAV, we would have had to increase the UART speed further. Our handlers would have to be extremely fast then, and we would have lost data then. So, we settled on 115200 bauds.

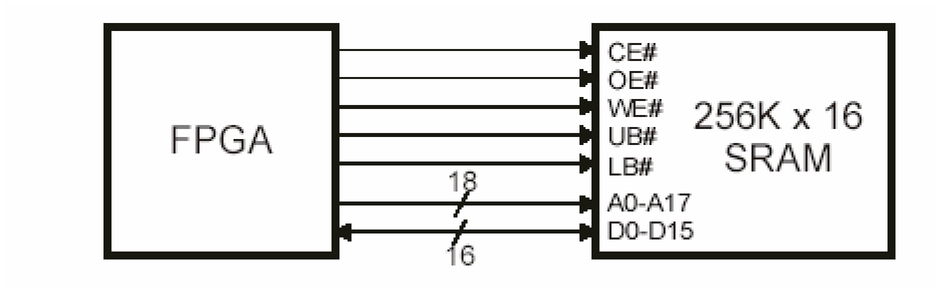
This speed is enough for 12 Khz ADPCM files though. Because after 1:4 decompression, we can manage to get the PCM samples at the correct frequency.

4.1.3 SRAM

We used the SRAM to create buffers for the input and output buffers. We have a 4kbyte UART buffer and 4kbyte audio buffer. We used the video memory section of the SRAM to make these buffers. This was helpful in debugging, because we could see how full/empty the buffers are at any time.

Our FPGA has access to a 256K x 16 SRAM (TC55V16256FT-15) for local storage of data.

SRAM pins:

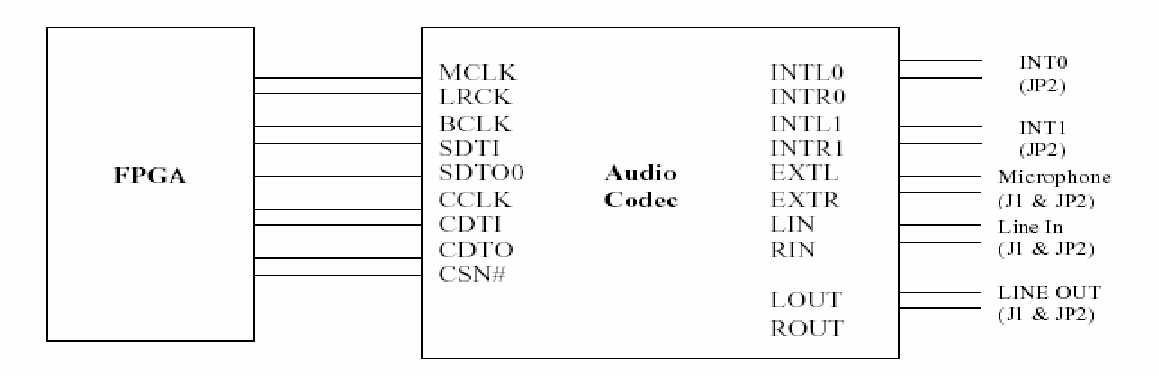


The SRAM was already configured for us. “Mylinkscript” defines which parts of software go in the SRAM. By following that, we were able to place out buffers in the SRAM.

4.1.4 Audio Codec

The FPGA streams the decoded bit stream serial (through the SDTI pin) to the audio codec through the BRAM buffer. The buffer can store 4 kilobits of data. Each sample is of 32 bits. So at a time, we can store upto 128 samples on it. If our sampling frequency is 44.1 Khz, 128 samples would require 2.6 ms. The audio controller would generate an interrupt after every 1.3 ms, indicating that the buffer is half full. From the buffer, the serial bit streams are synchronized with a clock from the FPGA that enters the codec on the BLCK signal. The master clock from the FPGA (MCLK) synchronizes all the internal operations of the codec. The FPGA uses the LRCK (Left Right Clock) to select the left or the right channel as the destination of the serial data.

Pin Configuration for Audio Codec:



.

4.1.5 Signal Definitions for CODEC

LRCK – left/right clock; when high data is for left channel, low for right channel
BCLK – bit clock; used to synchronize serial communications with chip
SDTO0 – serial output from chip to FPGA (encoding of analog in)
SDTI(i) – serial input from FPGA to chip (for analog out)

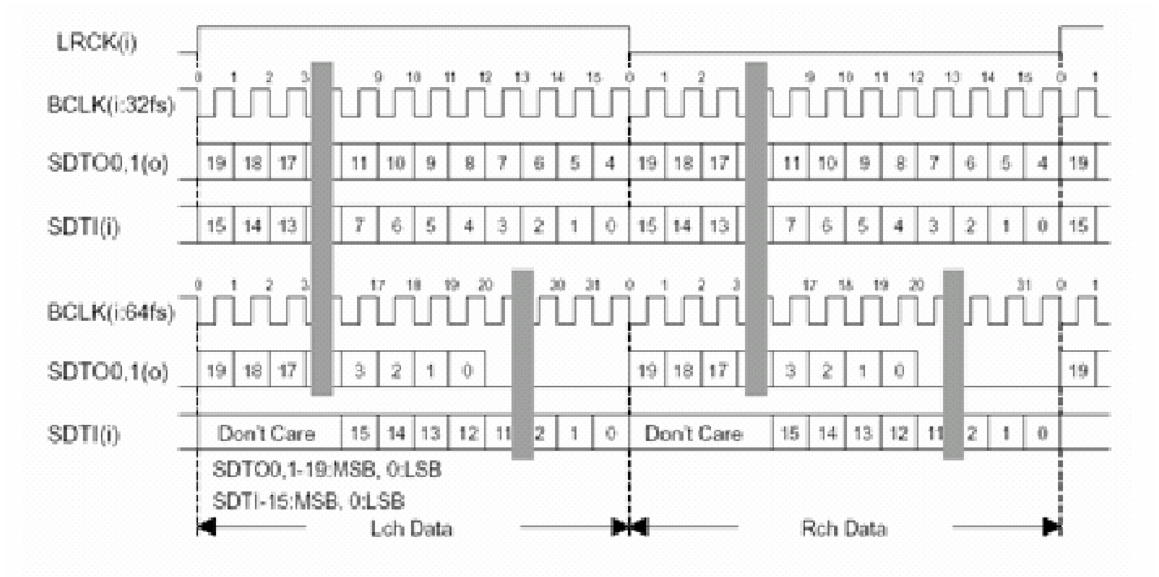


Figure 9. Audio Data Timing (No.0)

4.2 Execution

We modified the test232 script and added a communication protocol to it. Whenever it received a ‘1’ on its receiver buffer from the serial port, it stops transmitting any more characters. It restarts transmission only when it receives a '0'. This was done to prevent buffer overflow conditions. In this way, we would not lose out on any data because of the speed mismatch of the serial port and the decoding loop.

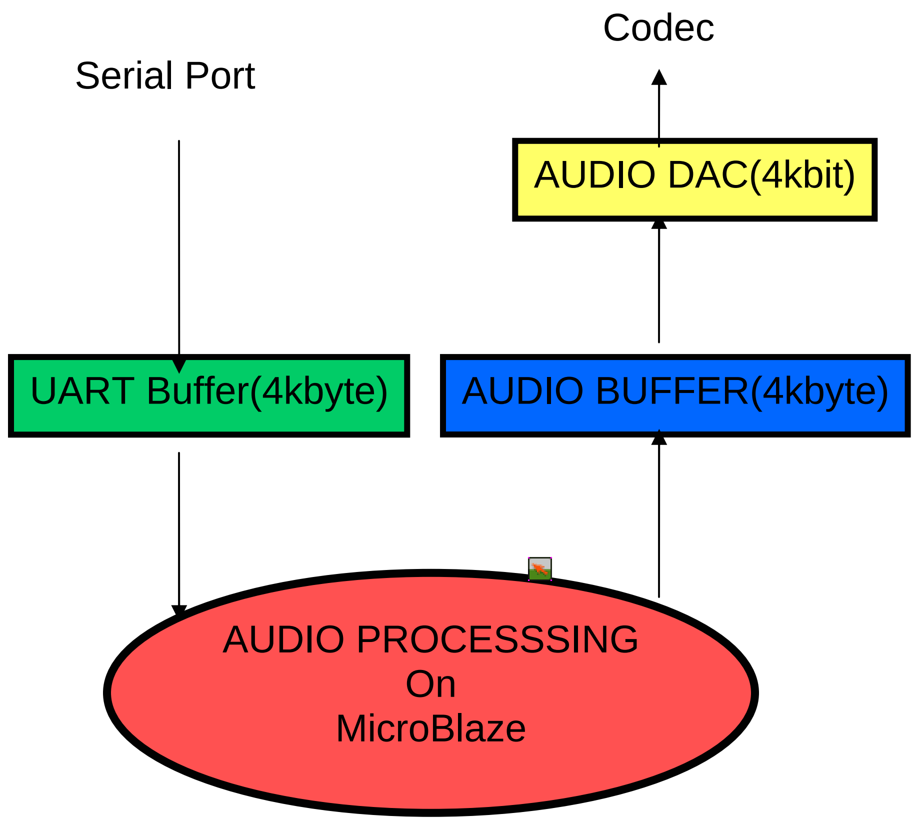
One the UART receives a character, it interrupts the main loop and the character is stored in the uart buffer. In the main loop, the first 44 bytes are read first to determine the file information. Based on this, decoding is done. In case of 8-bit mono PCM, one byte is read in and a 32 bit sample is made from it by first shifting it left by 8 bits and then doubling it to make it stereo. In case of 16-bit PCM, the same thing is done, but at a time 2 bytes are read in, and copied for left and right channels. Note: the codec is stereo, that's why we send data for both right and left channels

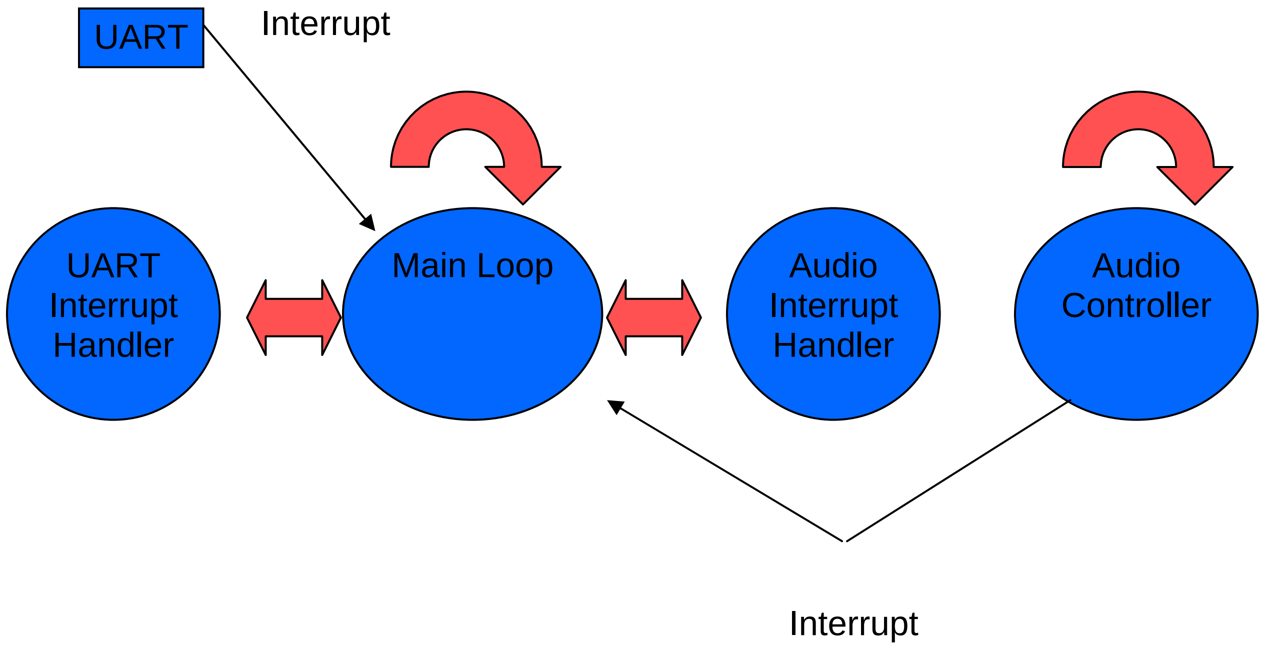
In case of an ADPCM file, the decoding function is called for a fixed number of bytes at a time. The decoded output is written to a temporary buffer.

Once a 32 bit sample is created in all these cases, it is written to the audio buffer. The audio interrupt controller reads in 64 samples at a time and writes them to the BRAM DAC, from where data is serially given to the codec. All this is controlled by the audio controller, a VHDL component. The audio controller is responsible for generating the clock signals for the codec, and for generating the interrupt for the microblaze processor.

The following diagrams explain the control flow and data flow during execution.

4.3 Data Exchange Flow





Control Flow Diagram

5. Project Material

Here is the list of files we created/modified. The path is relative to the main directory.

1. c_source_files/main.c
2. c_source_files/isr.c
3. c_source_files/main.h
4. c_source_files/fonts.h
5. c_source_files/decode.c
6. test232.c
7. myip/opb_xsb300_ak4565_v1_00_a/data/opb_xsb300_ak4565_v2_0_0.mpd
8. myip/opb_xsb300_ak4565_v1_00_a/data/opb_xsb300_ak4565_v2_0_0.pao
9. myip/opb_xsb300_ak4565_v1_00_a/hdl/vhdl/audio_ak4565.vhd
10. myip/opb_xsb300_ak4565_v1_00_a/hdl/vhdl/opb_xsb300_ak4565.vhd
11. system.mhs
12. system.mss

6. Conclusion & Contributions

This project has been a learning experience for us. It exposed us to challenges like making the right choice for implementation, how to decide between hardware and software based on critical timing analysis. We decided to do the decoding in software because the computational complexity of ADPCM decoding is not very high, and we made it even simpler by using simple operations like adding, shifting etc. But we realized that working in software gives you little control over the timing of the signals. We spent a lot of time in getting the right buffer sizes, synchronizing and optimizing execution, to meet the timing constraints.

Broadly speaking, the contribution of the team members can be classified into:
Devyani Gupta – VHDL modules (interfacing the application to the UART and audio codec modules, ROM modules for step size, shift tables), ADPCM Decoding.
Prakash Gowri Shankor – Application buffering modules, Audio Header processing module.
Vijayarka Nandikonda – ADPCM decoding, Debugging, tweaking serial port test code.

7. References

http://murray.newcastle.edu.au/users/staff/eemf/ELEC351/SProjects/hall/html/theoryad.html

http://www.xentec.be/products/vox_studio/help/ima_adpcm_format.htm

http://www.fact-index.com/p/pu/pulse_code_modulation_1.html

http://resource.intel.com/telecom/support/appnotes/adpcm.pdf