**FPGA based mp3 player**

CONTENTS Page No.

1. Acknowledgements 4

2. Abstract5

3. Introduction 6

4. System description 6

(a) System block diagram 7

(b) Description of subsystems 7

(c) System functional requirements and

performance specification 8

5. Background information on the MP3 format 8

(a) Standard for MP3 encoding and decoding 9

**(**b**)** Overview of MP3 encoding process9

6. Mp3 decoding 10

(a) Mp3 file structure 10

(b) Bitrate and sampling frequency table 13

(c) Initial reading 15

(d) Huffman decoding 15

(e) Re-quantization and reordering, 15

(f) Alias reduction, 15

(g) Inverse modified discrete cosine transform (IMDCT) 16

(h) Synthesis polyphase filter bank 16

7. Flow chart 17

8. The code 18

9. The hardware platform 22

10. Results 23

11. Observations and Debugging Procedures 23

12. Conclusion 23

13. Bibliography 24

**Abstract:**

Digital design using a Field Programmable Gate Array (FPGA) device is a rapidly evolving field. A complete embedded system can be built and programmed into a single FPGA chip for digital signal processing applications. The increasing density and capacity of these devices make it possible to implement an entire embedded system on a single chip. The goal of the project is to design an MPEG Layer III (*MP3*) player using a “Altera” FPGA board. The system will read an *MP3* file from a compact flash memory, decode the *MP3* bit stream into 16-bit pulse code modulated (PCM) outputs using a standard *MP3* decoding algorithm, and play the output through an external speaker. Hardware description language such as Verilog HDL is used to drive external peripherals, including the stereo Audio codec. The Audio codec converts the digital PCM outputs into an analog sound wave. The software and hardware designs are integrated on the Embedded Development Kit platform. In this project, data compression techniques are used in *MP3* encoding and decoding are explored and tested on hardware.

During the last few years the usage of the MPEG-1 layer-III (*mp3*) audio codec has exploded, and a large part of the global bandwidth consumed is used for transferring layer-III compressed audio data, or in casual language **“*mp3* files”.** During the first years of widespread *mp3* usages software decoders were the most common, but during the last couple of years portable and other stand-alone players gained in popularity. This report describes an attempt to create an efficient dedicated *mp3* decoder in hardware.

**Introduction:**  
 Digital design using a Field Programmable Gate Array (FPGA) is a rapidly evolving field. A complete embedded system can be built and programmed into a single FPGA chip for digital signal processing applications.   
  
  
The goals of this project are to—  
  
 (1) gain an in-depth understanding of hardware/software co-design using an FPGA,  
  
 (2) understand the specifications set for encoding and decoding *MP3* files., and  
  
 (3) build a FPGA-based MPEG Layer III (*MP3*) system, which implements the *MP3* decoding algorithm using VHDL and C language on the Embedded Development Kit (EDK) software platform

**System Description:**  
 The inputs to the FPGA *MP3* player system will be an *MP3* bit stream that is preloaded onto a compact flash memory(CFM) and any user interface control input. Using pushbuttons, the user will be enabled scan through the *MP3* file list, and then select, play, pause, and/or stop the song. In addition, volume control is triggered by a change in the on-board rotary encoder dial position.

The outputs of the *MP3* decoder, that is, 16-bit pulse code modulated (PCM) outputs and play the audio files through an external speaker. The PCM outputs need to be converted to analog format via the on-board stereo audio codec hardware chip before the audio can be heard with an external speaker that can be attached through the audio jack with a 15 mW amplifier. The system block diagram is as shown below in figure 1.1.

**System block diagram:**

Compact flash memory(input)

User interface

Display

**FPGA**

-Play/ pause

-Select mp3 files

-On/off

(using the pushbuttons present in FPGA)

External speakers

Stereo

Audio codec

**Figure 1.1:** Block diagram of FPGA based *mp3* player

**The system includes**:

* ***MP3* Decoder**

A *MP3* decoder runs on the FPGA that will decode the selected MP3 stream with the sampling frequency specified in the *MP3* header. A typical sampling frequency is 44.1 kHz. The software and hardware designs are integrated on the Xilinx Embedded Development Kit platform.

* **External Peripherals:**

Verilog HDL is used to drive all external peripherals. Most applications utilize devices by means of high-level device-generic commands. Driver software accept these generic high-level commands and break them into a series of low-level device-specific commands.

* **User Interface**

The user interface provides the inputs to control the *MP3* player, such as selecting, playing, pausing, and stopping the *MP3* files. It will also allow outputting related information on the LED.

* **Compact Flash Memory Card**

The 2 Gb capacity compact flash memory(CFM) supplies the preloaded *MP3* files for the *MP3* decoder system in the FPGA. *MP3* files are loaded onto to the CFM using a PC and memory card reader.

* **Onboard Stereo Audio Codec**

The Audio codec is used to convert the PCM format signal from the *MP3* decoder into an audio signal, which is fed into an external speaker through an audio jack.

**System functional requirements and performance specifications:**

* **Input *MP3* bit stream requirements:**

The *MP3* player will decode *MP3* inputs with various bit rates (from 128 kbps to 320 kbps) and different sampling frequencies (32 kHz, 44.1 kHz or 48 kHz)

* D**ecoding speed:**

The ultimate objective of decoding speed is to process *MP3* files in real-time. The execution time of the *MP3* decoding will be profiled and measured. If the real-time specification can not be met, further optimization will be needed.

**Background information on the *MP3* format:**

The need to reduce the size of audio files without any noticeable quality loss was stated in the 1980ies by the International Organization for Standardization (ISO). A working group within the ISO known as the Moving Pictures Experts Group (MPEG), developed a standard that contained several techniques for both audio and video compression. The audio part of the standard included three modes with increasing complexity and performance, as shown in Figure 6-1. The third mode, called Layer III, manages to compress music by a factor of 12 with almost no audible degradation. This technique is known as MP3 and has become very popular and widely used in applications today.

**Table 1.2:**

**Overview of *MP3* Encoding Process:**

*MP3* encoding involves representing a song as a bit stream (an array of 0’s and 1’s) that can be recovered by a *MP3* decoder (player). The high percentage compression involved in *MP3* encoding allows songs to be stored and shared rather easily and quickly on computers and through the internet without losing any perceptible quality. This lossy compression works by first masking inaudible frequency components to the human ear, and then using several data compression techniques that remove data redundancies.

First the analog audio is sampled at a specific sampling rate, typically at 44.1 kHz. This is due to Nyquist frequency’s rule, in which the sampling rate must be at least two times greater than the largest possible frequency component present in the data. And since the range of audible frequencies to the human ear is roughly 20 Hz to 22 kHz, this sampling rate is usually chosen. The signals are quantized using pulse code modulation, where each sample amplitude is represented by 16 bits.

To remove redundancies and compress data, frequency analysis techniques are used. The PCM samples are filtered for 32 equal frequency spectrums, called subbands using a polyphase architecture that yields in a higher computational efficiency. A discrete cosine transformation is then applied to remove low energy signals from high frequency components.

Further compression is achieved by using a lossless compression technique known as Huffman encoding that is based on statistical behavior of data. Finally, the bit stream is arranged into frames that the MP3 decoder will analyze to reconstruct the MP3 sound.

***MP3* decoding:**

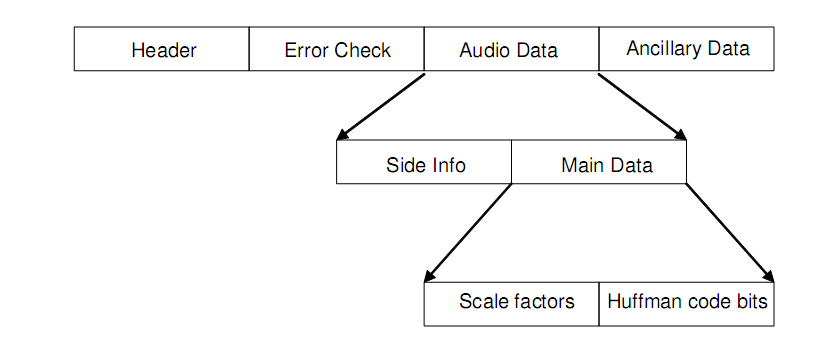
*MP3* decoding is the reverse process of *MP3* encoding. Fortunately, decoding is not nearly as complex, since it does not require a psychoacoustic model (a virtual model of the human ear and how it perceives different frequencies). The *MP3* decoder’s role is to recover the original audio by analyzing certain sections of a frame to gain information about encoding parameters used and then use reverse procedures to reconstruct PCM samples.

Each frame consists of exactly 1152 PCM samples and contains at least two sections:

A header section that contains important encoding parameters, such as bit rate and sampling frequency used and an audio data section that holds the encoded bit stream. Some MP3 bit streams contain an optional ID3 tag frame that can be used to store MP3 related information including the title and author of the song.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TAG  (optional) | Frame 1 | Frame 2 | Frame 3 | ……… | TAG  (optional) |

The first four bites of each frame is Header and the rest are data. The size of each frame varies according to bitrate.



**Figure 1.3:** MPEG I layer 3 frame format

The various fields in a frame of audio data are discussed below.

**Header** is 4 bytes long and contains sync word to indicate the start of frame. Header contains Layer information (MPEG Layer I, II or III), bitrate information, sampling frequency and mode information to indicate if the stream is mono or stereo.

**Error Check** This fields contains a 16 bit parity check word for optional error detection with in the encoded stream.

**Side information** Contains information to decode Main data. Some of the fields in side information are listed below--

1. It contains scale factor selection information, that indicate the number of scalefactors transferred per each subband and each channel. Scalefactors indicate the amount bywhich an audio sample needs to be scaled. Since, human ear response is different for signals at different frequencies, the entire audio spectrum is divided into subbands. The samples in the more sensitive bands are scaled more than the samples in the lesser sensitive region of the spectrum.

2. It contains global gain which needs to be applied to all the samples in the frame.

3. Information regarding the number of bits used to encode the scalefactors. To achieve compression, even the scalefactors are encoded to save the bits. This information in the sideinfo will indicate the number of bits to encode a particular scalefactor.

4. Information regarding the huffman table to be selected to decode a set of samples. This information specifies one of the 32 huffman tables used for huffman decoding. Main data The main data contains the coded scale factors and the Huffman coded bits.

5. Scalefactors are used in the decoder to get division factors for a group of values. These groups are called scalefactor bands and the group stretches over several frequency lines. The groups are selected based on the non-uniform response of human ear for various frequencies.

6. The quantized values are encoded using huffman codes. The huffman encoding is used to code the most likely values with lesser number of bits and rarely occurring values with larger number of bits. The huffman codes are decoded to get the quantized values using the table select information in the sideinfo section of the frame.

**Ancillary data** This field is the private data and the encoder can send extra information like ID3 tag containing artist information and name of the song.

The frame size in bytes varies from song to song, and in some cases, even within one song (when using variable bit rates). The general equation for calculating the frame size in bytes is found in Equation 1.4.

**Equation 1.4**

Frame size (in bytes) = (144\* bit rate)/ (sampling rate + padding)

Where 144= (1152 PCM/frame) / (8 bits/byte) and where padding is an integer number to ensure that the frame size is an integer number Bit rate is the rate at which the compressed bit stream is delivered from the storage medium to the input of a decoder while sampling frequency defines the umber of samples per second taken from a continuous signal to make a discrete signal. For MP3 encoding, there are several allowed bit rates and sampling frequencies that can be used.

Typically, a sampling rate of 44.1 kHz is used and is known as “CD quality” while 48 kHz is referred to as “DVD quality.”

For MP3 encoding, there are several allowed bit rates and sampling frequencies that can be used, as illustrated in Table 1.5 and 1.6 respectively. These tables are copied directly from the ISO standard document.

**Table 1.5:**

|  |  |  |  |
| --- | --- | --- | --- |
| Bitrate index | Bitrate specified(Kbps) | | |
| Layer I | Layer II | Layer III |
| 0000  0001  0010  0011  0100  0101  0110  0111  1000  1001  1010  1011  1100  1101  1110  1111 | Free  32  64  96  128  160  192  224  256  288  320  352  384  416  448  Forbidden | Free  32  48  56  64  80  96  112  128  160  192  224  256  320  384  Forbidden | Free  32  40  48  60  64  80  96  112  128  160  192  224  256  320  Forbidden |

**Table 1.6:**

|  |  |
| --- | --- |
| Sampling frequency | Frequency specified |
| 00  01  10  11 | 44.1  48  32  Reserved |

The *MP3* decoding process is shown in Figure 1.7. It includes the following stages:

1) Initial reading

2) Huffman decoding

3) Re-quantization and reordering,

4) Stereo decoding, alias reduction,

5) Inverse modified discrete cosine transform (IMDCT) and

6) Synthesis polyphase filter bank

***Mp3* bitstream**

Re-quantization

Huffman decoding

Initial reading

Alias reconstruction

Synthesis filter bank

IMDCT

**16 bit PCM samples**

**Fig 1.7:** Diagram of decoding process

**Initial reading:**

The incoming data stream is split up into individual frames. The header section of each frame is analyzed to obtain parameters used in the encoding process (i.e. bit rate and sampling frequency). The first action is the synchronization of the decoder to the incoming bit stream by checking if the first 12 bits of the header section are 1’s. Scalefactors and Huffman table selection bits are also decoded.

**Huffman decoding:**

The Huffman algorithm is used for lossless data compression. The basic idea of the technique is to assign shorter binary codes to more frequent samples and longer codes to less frequent samples. The Huffman decoding procedure is based on tables that are used to map the Huffman binary codes to the original samples.

**Re-quantization:**

During the encoding process, the outputs of the MDCT, or frequency domain samples, were pre-quantized in an attempt to use more precision when needed. It turns out that finer frequency resolution is needed for low volume sounds and larger values are coded with less accuracy. Afterwards, the values were scaled, or multiplied by a scalefactor, a value that is based on the absolute threshold of the human ear (a frequency dependent function). Larger scalefactors are needed if the frequency components are more difficult to hear. So for the decoding process, the values need to be requantized. Afterwards, de-scaling is required.

**Re-ordering:**

In the MP3 encoding process, the use of short windows would generate frequency lines ordered first by subband, then by window and at last by frequency. In order to increase the efficiency of the Huffman coding the frequency lines for the short windows case were reordered into subbands first, then frequency and at last by window, since the samples close in frequency are more likely to have similar values. The reordering block in the MP3 decoding process will re-sort the samples by subbands, then on windows and then on increasing frequency. For a description on windowing, refer to the IMDCT block.

**Alias reduction block:**

Aliasing is the overlap of frequency components when energies greater than Nyquist frequency are present. This is the result of the decimation or the reduction of sampling rate in the analysis filter bank process where overlapping of adjacent subband filters is inevitable. In the encoding process, these aliasing effects are removed to reduce the amount of information that needs to be transmitted. This can be achieved by using a series of butterfly computations that add weighted, mirrored versions of adjacent subbands to each other. In the decoding process, aliasing artifacts must be added to the signal again in order to obtain a correct reconstruction of the audio signal. The alias reconstruction calculation consists of eight butterfly calculations for each subband.

**Inverse Modified Discrete Cosine Transform (IMDCT):**

The Inverse Modified Discrete Cosine Transform (IMDCT) is the inverse of the modified discrete cosine transform used in MP3 encoding. The MDCT was used to represents signals as a sum of cosine waves, essentially transforming them to the frequency domain. Compared to the DFT and other well-known transforms, the MDCT has a few properties that make it very suitable for audio compression. First of all, the MDCT has the energy compaction property common to discrete cosine transforms. This means most of the information in the signal is concentrated to a few output samples with high energy. The term modified is used since there is a 50% overlap. The lower 18 values are added with the higher 18 values from the previous

frame, and used as output. The higher 18 values are then stored and used the same way when the next frame is being decoded. This overlapping that avoids sharp discontinuities.

**Synthesis polyphase filter bank :**

The synthesis polyphase filter bank is the final step in the decoding process. It is used to combine the signal energies from all the 32 subbands. The result output for each frame is 1152 16 bit PCM samples. A polyphase architecture is used since the decimation of the sampling rate allows the use of a lower number of filter coefficients, and thus improves computational efficiency. The method recommended by ISO standard for transforming subband samples to the Pulse code modulated format involves shifting, matrixing with a 32 point discrete cosine transform that represent band pass filter coefficients, a 512 point window to improve filter quality, and finally a summation for all the subbands. Various algorithms can be implemented that observe symmetry properties and reduce the number of computations. For example, the DCT function can be calculated using a method known as the fast DCT in a similar manner that a DFT function can be more efficiently computed using the FFT. The method recommended by the ISO standard document [5] for transforming the subband samples into the PCM format is illustrated in Figure

**Flowcharts:**

The *MP3* decoding algorithm described thus far is implemented completed in software using C language. High level flowcharts for the main program, as well as the select\_song and play\_song functions are illustrated below.

**Select\_song**

**DIP SW UP?**

**Enter main**

NO

YES

**Enable cache**

**Play\_song**

**Disable cache**

**Fig 1.8 :** Flow Chart for FPGA *mp3* player

**Code:**

#include<stdio.h>

#include <math.h>

#include "xio.h"

#include "sleep.h"

#include "xparameters.h"

#include <xbasic\_types.h>

int c;

#define MY\_AC\_BASEADDR XPAR\_OPB\_AC\_CONTROLLER\_REF\_0\_BASEADDR

#define AC\_InFIFO MY\_AC\_BASEADDR

#define AC\_OutFIFO MY\_AC\_BASEADDR + 0x4

#define AC\_FIFO\_Status MY\_AC\_BASEADDR + 0x8

#define AC\_Control MY\_AC\_BASEADDR + 0xC

#define AC\_RegAddr MY\_AC\_BASEADDR + 0x10

#define AC\_RegRead MY\_AC\_BASEADDR + 0x14

#define AC\_RegWrite MY\_AC\_BASEADDR + 0x18

#define AC\_IN\_FIFO\_OFFSET 0x0

#define AC\_STATUS\_OFFSET 0x8

#define AC\_InFIFO\_Full 0x01

#define AC\_InFIFO\_Half\_Full 0x02

#define AC\_OutFIFO\_Full 0x04

#define AC\_OutFIFO\_Empty 0x08

#define AC\_Reg\_Access\_Finished 0x10

#define AC\_CODEC\_RDY 0x20 <stdio.h>

#include <math.h>

#include "xio.h"

#include "sleep.h"

#include "xparameters.h"

#include <xbasic\_types.h>

int c;

#define MY\_AC\_BASEADDR XPAR\_OPB\_AC\_CONTROLLER\_REF\_0\_BASEADDR

#define AC\_InFIFO MY\_AC\_BASEADDR

#define AC\_OutFIFO MY\_AC\_BASEADDR + 0x4

#define AC\_FIFO\_Status MY\_AC\_BASEADDR + 0x8

#define AC\_Control MY\_AC\_BASEADDR + 0xC

#define AC\_RegAddr MY\_AC\_BASEADDR + 0x10

#define AC\_RegRead MY\_AC\_BASEADDR + 0x14

#define AC\_RegWrite MY\_AC\_BASEADDR + 0x18

#define AC\_IN\_FIFO\_OFFSET 0x0

#define AC\_STATUS\_OFFSET 0x8

#define AC\_InFIFO\_Full 0x01

#define AC\_InFIFO\_Half\_Full 0x02

#define AC\_OutFIFO\_Full 0x04

#define AC\_OutFIFO\_Empty 0x08

#define AC\_Reg\_Access\_Finished 0x10

#define AC\_CODEC\_RDY 0x20

#define AC\_ExtendedAudioStat 0x2A

#define AC\_PCM\_DAC\_Rate 0x2C //from xac\_l.h

#define AC\_PCM\_ADC\_Rate 0x32 //from xac\_l.h

#define AC\_PCM\_DAC\_Rate0 0x78

#define AC\_PCM\_DAC\_Rate1 0x7A

#define AC\_Reserved0x34 0x34

#define AC\_JackSense 0x72

#define AC\_SerialConfig 0x74

#define AC\_MiscControlBits 0x76

#define AC\_VendorID1 0x7C

#define AC\_VendorID2 0x7E

// Volume Constants

#define AC\_VolMute 0x8000

#define AC\_VOL\_MIN 0x1f1f //0x3F3F

#define AC\_VOL\_MID 0x1010// 0x0a0a //0x1010

#define AC\_VOL\_MAX 0x0000

// Macros for reading/writing AC core registers

#define XAC\_mGetRegister (BaseAddress, offset) XIo\_In32((BaseAddress + offset))

#define XAC\_mSetInFifoData (BaseAddress, value) XIo\_Out32 ((BaseAddress) +

AC\_IN\_FIFO\_OFFSET, (value))

#define XAC\_mGetOutFifoData (BaseAddress) XIo\_In32 ((BaseAddress + AC\_OUT\_FIFO\_OFFSET))

#define XAC\_mGetStatus (BaseAddress) XIo\_In32 ((BaseAddress + AC\_STATUS\_OFFSET))

#define XAC\_mSetControl (BaseAddress, value) XIo\_Out32 ((BaseAddress) + AC\_CONTROL\_OFFSET,

#define XAC\_mSetACRegisterAccessCommand (BaseAddress, value) \

XIo\_Out32 ((BaseAddress) + AC\_REG\_CONTROL\_OFFSET, (value))

#define XAC\_mGetACRegisterData (BaseAddress) XIo\_In32 ((BaseAddress +

AC\_REG\_READ\_OFFSET))

#define XAC\_mSetACRegisterData (BaseAddress, value) XIo\_Out32 ((BaseAddress) +

AC\_REG\_WRITE\_OFFSET, (value))

// Status register macros

#define XAC\_isInFIFOFull (BaseAddress) (XAC\_mGetStatus (BaseAddress) & AC\_IN\_FIFO\_FULL)

#define XAC\_isInFIFOEmpty (BaseAddress) (XAC\_mGetStatus (BaseAddress) &

AC\_IN\_FIFO\_EMPTY)

#define XAC\_isOutFIFOEmpty (BaseAddress) (XAC\_mGetStatus (BaseAddress) &

AC\_OUT\_FIFO\_EMPTY)

#define XAC\_isOutFIFOFull (BaseAddress) (XAC\_mGetStatus (BaseAddress) &

AC\_OUT\_FIFO\_FULL)

#define XAC\_isRegisterAccessFinished (BaseAddress) \

((XAC\_mGetStatus (BaseAddress) & AC\_REG\_ACCESS\_BUSY) == 0)

// (XAC\_mGetStatus (BaseAddress) & AC\_REG\_ACCESS\_FINISHED))

#define XAC\_isRegisterAccessError (BaseAddress) \

((XAC\_mGetStatus (BaseAddress) & AC\_REG\_ACCESS\_ERROR) > 0)

#define XAC\_isCodecReady (BaseAddress) \

(XAC\_mGetStatus (BaseAddress) & AC\_CODEC\_RDY)

#define XAC\_isInFIFOUnderrun (BaseAddress) (XAC\_mGetStatus (BaseAddress) &

AC\_IN\_FIFO\_UNDERRUN)

#define XAC\_isOutFIFOOverrun (BaseAddress) (XAC\_mGetStatus (BaseAddress) &

AC\_OUT\_FIFO\_UNDERRUN)

#define XAC\_getInFIFOLevel (BaseAddress) \

((XAC\_mGetStatus (BaseAddress) & AC\_IN\_FIFO\_LEVEL) >> AC\_IN\_FIFO\_LEVEL\_RSH

void XAC\_Delay (Xuint32 value) {

while (value-- > 0);

}

#define AC\_CLEAR\_IN\_FIFO 0x1

#define AC\_CLEAR\_OUT\_FIFO 0x2

#define AC\_ENABLE\_IN\_FIFO\_INTERRUPT 0x4

#define AC\_ENABLE\_OUT\_FIFO\_INTERRUPT 0x8

#define AC\_ENABLE\_RESET\_AC 0x10

#define AC\_DISABLE\_RESET\_AC 0x0

#define AC\_CLEAR\_FIFOS AC\_CLEAR\_IN\_FIFO | AC\_CLEAR\_OUT\_FIFO

/#define XAC\_mSetControl (BaseAddress, value) \

/ XIo\_Out32 ((BaseAddress) + AC\_CONTROL\_OFFSET, (value))

/

#define AC\_CONTROL\_OFFSET 0xC //works if C, E or F...does not otherwise

void XAC\_ClearFifos (Xuint32 BaseAddress)

{

Xuint32 i;

XAC\_mSetControl (BaseAddress, AC\_CLEAR\_FIFOS);

for (i = 0; i < 512; i++)

XAC\_mSetInFifoData (BaseAddress, 0);

}

void WriteACReg (int reg\_addr, int value)

{

XIo\_Out32 (AC\_RegWrite, value);

XIo\_Out32 (AC\_RegAddr, reg\_addr);

//while ((XIo\_In32 (AC\_FIFO\_Status) & AC\_Reg\_Access\_Finished) == 0);

usleep (10);

}

int ReadACReg (int reg\_addr

(

XIo\_Out32 (AC\_RegAddr, reg\_addr | 0x80);

// while ((XIo\_In32 (AC\_FIFO\_Status) & AC\_Reg\_Access\_Finished) == 0);

usleep (10);

return XIo\_In32 (AC\_RegRead); }

#define AC\_IN\_FIFO\_FULL 0x01

void XAC\_WriteFifo (Xuint32 BaseAddress, Xuint32 sample)

{

while (XAC\_isInFIFOFull (BaseAddress));

XAC\_mSetInFifoData (BaseAddress, sample);

}

int SetupAC (int samplerate) {

WriteACReg (AC\_Reset, 0);

while (! (XIo\_In32 (AC\_FIFO\_Status) & AC\_CODEC\_RDY)) {};

XAC\_ClearFifos (MY\_AC\_BASEADDR); /\*\* Clear FIFOs \*\*/

//xil\_printf("-- Set DAC rate to %d Hz \r\n",samplerate);

WriteACReg (AC\_PCM\_DAC\_Rate, samplerate);

WriteACReg (AC\_PCM\_DAC\_Rate0, samplerate);

// xil\_printf ("-- Volume settings initialized \r\n");

//need these settings, especially PCMoutVol...otherwise no audio heard

WriteACReg (AC\_MasterVol, AC\_VOL\_MAX);

WriteACReg (AC\_HeadphoneVol, AC\_VOL\_MAX);

WriteACReg (AC\_MasterVolMono, AC\_VOL\_MAX);

WriteACReg (AC\_PCBeepVol, AC\_VolMute);

WriteACReg (AC\_PhoneInVol, AC\_VolMute);

WriteACReg (AC\_CDVol, AC\_VolMute);

WriteACReg (AC\_VideoVol, AC\_VolMute);

WriteACReg (AC\_AuxOutVol, AC\_VolMute);

WriteACReg (AC\_PCMOutVol, AC\_VOL\_MAX);

WriteACReg (AC\_RecordGain, AC\_VolMute); //added

WriteACReg (AC\_PowerDown, 0x0100); //added

WriteACReg (AC\_LineInVol, AC\_VolMute); //added

}

//these variables must be initialized to zero outside volume\_control function

//otherwise, incorrect results

int q=0;

Xuint32 VI\_old=0;

void volume\_control (int VI)

{

//initial method for checking encoder position and updating volume register value

//works, but only 4 different settings AND encoder values (VI) are not predictable

//one would think that values would increase by some constant for every increment in encoder position

//However, VI values are very random for the 360 degree rotation

//Majority of positions are VI==0

// if (VI==1) {WriteACReg (AC\_HeadphoneVol, AC\_VOL\_MAX) ;}

// if (VI==2) {WriteACReg (AC\_HeadphoneVol, AC\_VOL\_MIN) ;}

// if (VI==3) {WriteACReg (AC\_HeadphoneVol, AC\_VOL\_MID) ;}

// if (VI==4) {WriteACReg (AC\_HeadphoneVol, AC\_VolMute) ;}

int volume\_setting;

//xil\_printf("VI:%d \n\r", VI);

if(VI!=VI\_old)

{

q++;

volume\_setting=8000 - (1000\*q);

//1000 is an arbitrary number, but very good choice for noticeable difference in volume

//xil\_printf ("vol: %s\n\r", volume\_setting); //--for debugging purposes

if (volume\_setting<1) {q=0 ;}

WriteACReg (AC\_HeadphoneVol, volume\_setting);

}

VI\_old=VI;

//xil\_printf ("VI\_old:%d \n\r", VI\_old); //--for debugging purposes

}

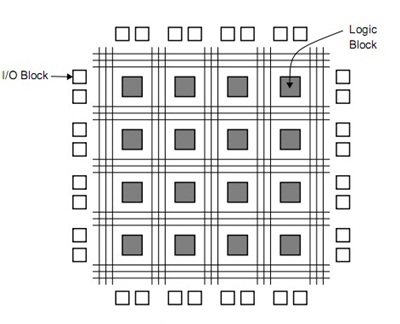
**The Hardware plateform:**

The basic hardware platforms used are FPGA and the stereo audio codec.

**FPGA - Field Programmable Gate Array:**

FPGA is a silicon chip with unconnected logic gates. It is an integrated circuit that contains many (64 to over 10,000) identical logic cells that can be viewed as standard components. The individual cells are interconnected by a matrix of wires and

programmable switches. Field Programmable means that the FPGA's function is defined by a user's program rather than by the manufacturer of the device. Depending on the particular device, the program is either 'burned' in permanently or semi-permanently as part of a board assembly process, or is loaded from an external memory each time the device is powered up.



**Fig 1.9**. The FPGA block

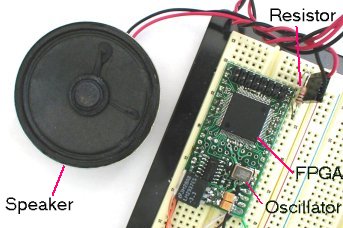
The Field-Programmable Gate Arrays (FPGAs) provide the benefits of custom CMOS VLSI, while avoiding the initial cost, time delay, and inherent risk of a

conventional masked gate array. The FPGAs are customized by loading configuration

data into the internal memory cells.

**Stereo Audio Codec:**

The Audio codec is used to convert the PCM format signal from the *MP3* decoder into an audio signal, which is fed into an external speaker through an audio jack.

****

**Fig1.10**. The functional diagram

**Results:**

The first few weeks have been dedicated to becoming familiar with the Xilinx ISE software. Xilinx tutorials and board demonstration projects have been studied. In addition, peripheral setup, compilation, and debugging procedures have been practiced extensively.

The majority of specifications and goals were met. MP3 files can be accessed and read from the compact flash memory. The list of songs on the compact flash can be scanned and selected. The MP3 bit stream can be decoded in real time and heard on an external speaker with adequate quality. The “STOP,” “PAUSE,” and “Volume control” functions was completed in the last week. Rewind, forward modes and the LCD display were never experimented due to time constraints.

**Observations:**

Testing and debugging on the Spartan XC2S100 board proved inconclusive. Failure to get results with the project schedule time winding down, a switch to use to the Altera development board that would allow the use of a full user-interface subsystem.

**Conclusion:**

In this project, a FPGA-based MP3 decoder has been implemented on the Altera development board. It can read MP3 files from a compact flash memory device, then decode and play it through the stereo audio codec. Different controls such as song selection, pause and stop modes are included.

**Bibliography:**

A. Abdel-Gawad, “A full hardware implementation for an MP3 decoder chip

using VHDL”, Project report, University of California at Santa Barbara.

A Full Hardware Implementation for an MP3 Decoder Chip using VHDL, Ahmed H.

Abdel-Gawad, Saad F. Abdel-Aziz, et al.

<http://www.fpga4fun.com>

<http://www.wikipedia.com>

<http://www.xilinx.com>

<http://www.indianengineer.tk>