FPGA based Music Player

Project Report

Hashan Roshantha Mendis
Y6356306

MSc. Digital Systems Engineering
Department of Electronics
University of York

August 2011

Supervised by: Dr. Gianluca Tempesti

Professor Andy Tyrrell

Acknowledgement

I would like to thank my project supervisor, Dr. Gianluca Tempesti, for sharing his broad-knowledge, guidance, patience and his generosity in entertaining the massive amounts of queries which was subjected towards him during this acamdemic year. His encouragement, good ideas and excellent teaching made this project successful.

I would like to thank Professor Andy Tyrell, Dr. Andy Pomfret, Dr. Jez Wells, Mr. Jonathan Dell, Mr. David Hunter and all other academic staff in the University of York, Electronics Dept., Digital Systems Engineering Programme, for their guidance and help throughout this project. Their in-depth lectures and laboratory exercises have helped in many instances where I was faced with engineering challenges.

I would like to thank all my group members for their ideas, assistance provided and collaboration throughout the academic year.

I would like to thank my parents and sister, for their love and support throughout my postgraduate studies. I also thank my extended family members and friends for their kindness and generosity.

I would also like to thank Ms. Bithika Mookherjee for teaching me good coding practices, Mr. Colin Bonney for his kind help in debugging several software/hardware issues and the Xilinx Engineers for their support provided via the Xilinx community forums.

Abstract

This project report presents an implementation of audio decoding in the form of MP3 and ADPCM on an Field Programmable Gate Array (FPGA). The decoder is purely a software based sub-system of a larger embedded system – a music player. A decoder is necessary to enable playback of encoded(compressed) audio. The hardware aspect of the embedded system is a custom designed PCB, incorporating an FPGA(Xilinx Spartan 3E), DAC, SDRAM, SDCard adapter, Colour LCD, PROMs and other surface mounted components.

The decoder algorithm lies at the heart of the music player. Audio compression is necessary to be able to store a large number of songs on a SD Card as opposed to using raw uncompressed audio (e.g.: WAVE). MP3 is a widely used commercial audio codec able to achieve compression rates of 10:1, however it is a computationally expensive task. ADPCM on the other hand allows compression rates of 4:1 but relatively less processing power is needed.

The 'MAD' MP3 decoder library is investigated and successfully ported on to the MicroBlaze platform. However due to resource constraints the ADPCM decoder was ultimately the final choice of decoder for the music player and has been successfully integrated with the overall system.

Implementation of a bootloader for the system was explored – to convert the design into a standalone embedded system. Alternatively the MP3 decoder was tested using XMD with MicroBlaze caching enabled. The best case results achieved was – decoding 1 second of MP3 music in 0.9475 seconds.

Contents

1 Introduction

Digital Signal Processing(DSP) and digital design using Field Programmable Gate Arrays (FPGA) is a rapidly evolving field. A commercial level complete embedded system can be built and programmed into a single FPGA chip for a digital signal processing application.

The goal of this project was to design a music player using a custom built PCB incorporating FPGA technology. The system would utilise the MicroBlaze soft-core processor running on a Xilinx Spartan 3E 1200E (FG400) FPGA with additional peripherals like a DAC, LCD Screen, SD Card, SDRAM and PROMs. The **primary objective** of the project was to implement an audio decoding algorithm on the embedded device to be able to play-out compressed audio and **secondary goals** were to add extra music player functionality such as play song, pause song, select song, fast forward, skip backwards etc.

The decoder is a valuable component of the system comprising of entirely software (Embedded C code). Without a decoder the system would be able to play **WAVE** files – which in it's uncompressed form are very large files (1 minute of 44.1Khz WAVE file = 5MB). Hence compression is crucial to be able to store many music files in a small storage device like a 2GB SD Card. Audio decompression is however a processor expensive task and hence many compressed audio formats exist in commercial usage.

According to the design document it was decided by the group that **MP3** (MPEG Layer 3) would be an ideal compressed audio format for the music player to support. As backup choices **ADPCM** (Adaptive Differential Pulse Code Modulation) and **WAVE** (raw uncompressed audio) was chosen. The decoder algorithm implementation was largely a software (Embedded C) oriented task utilising existing decoder libraries.

Chapter 2 of this report will describe the steps taken to test the MP3 decoder library on a Xilinx FPGA platform(Virtex-5). **Chapter 3** describes the two solutions explored to test the MP3 decoder library on a Xilinx Spartan 3E FPGA, with relatively low resources. **Chapter 4** explains the steps taken to implement, test and integrate the backup codec – ADPCM, into the final system. **Chapter 5** discusses some future work that will be carried out on this project as well as any design changes that can be explored at a future date. The **appendix** section of this report contains the source code developed during the implementation and testing phase of the respective tasks. Several difficulties and problems were faced during this project – and they have been listed in the appropriate sections of this report along with steps taken to overcome these challenges. A software CD is also bundled with the report and contains full listings of the software and hardware (EDK projects) used for the project and any test results achieved.

1.1 Overall system design

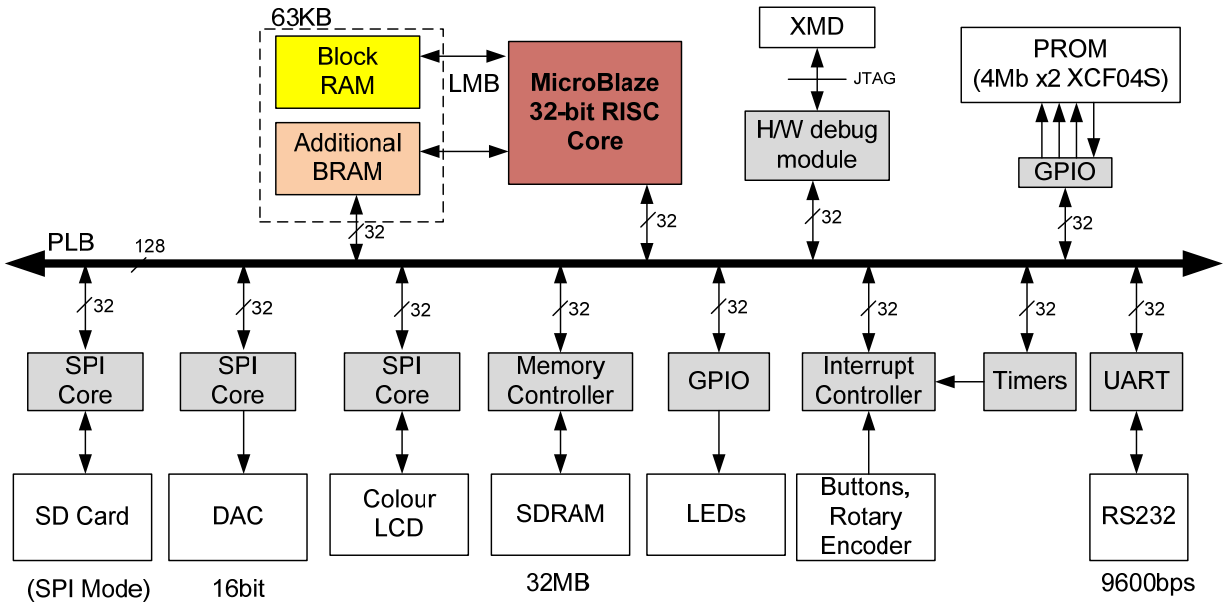


Figure 1.1 - High Level Diagram of the Overall System (adapted from design document)

Figure 1.1 shows a high level block diagram of the entire system. Usage of the different components are given in Table 1.1.

MicroBlaze	Main processing unit – code written in C is executed.
BRAM	Stores the program code (Local Memory)
SDCard	To store the music files – to be decoded and played out by he device later on.
Colour LCD	To display the graphical user interface.
DAC	To send out the PCM samples of the decoded audio.
SDRAM	To buffer the song prior to decoding/playing out.
LED	To show status output
Buttons/Rotary Encoder	For user interaction – play/pause/rewind/fast-forward/volume control etc.
Timer	To accurately output the PCM samples to the DAC, according to the sample rate of the song.
RS232/UART	To output debug information.
PROMs	To store the application data and FPGA configuration data (non-volatile). 2nd PROM was included to store 'only' the application code. PROMS were configured using iMPACT. The final version of the project stored both config. and application in a single PROM.
JTAG port	To configure FPGA via PC and for debugging (using XMD - Xilinx Microprocessor Debugger)

Table 1.1 - System components and their usage

1.2 System Hardware Design

All hardware components in the system other than the Rotary Encoder and the button debouncers, are Xilinx IP Cores, available in the Xilinx EDK. Using these off-the shelf IP Cores saved development and testing time significantly.

Below are a list of the cores used in the final version of the complete system.

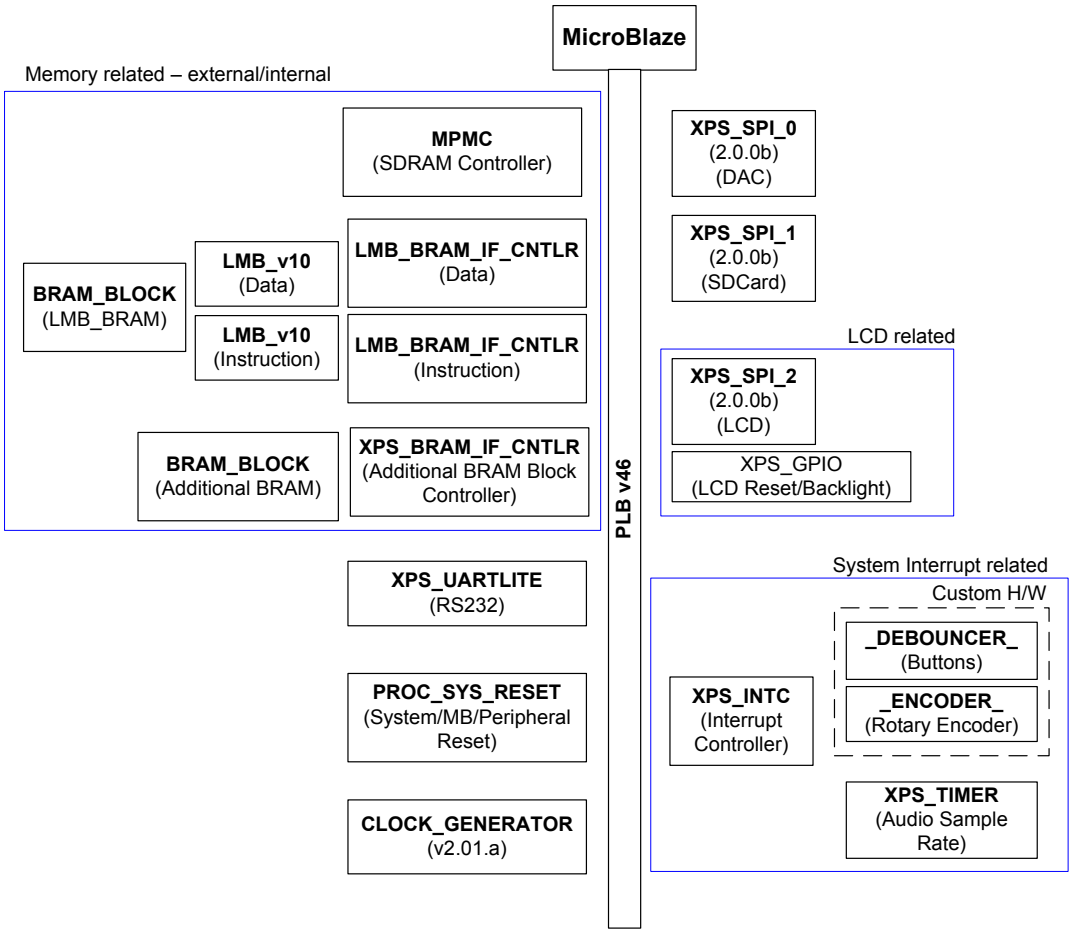


Figure 1.2 - Hardware Design of the system - the different Xilinx IP Cores used

One major challenge faced when using these cores – is that the documentation (datasheets) given for them are complicated to understand and at most times have information missing, poorly written or not all the information is in one coherent document – but scattered across multiple documents. Understanding and configuring the IPCores to suit our needs was a valuable skill gained throughout this project.

1.3 System Software Design

As Figure 1.2 shows the software used in the system is of two categories – Xilinx specific device drivers, and the custom software routines.

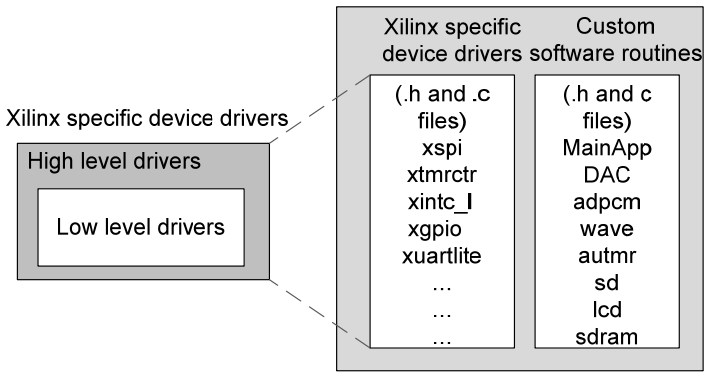


Figure 1.3 - Software design

Each Xilinx IP Core comes packaged with device drivers written in C. These need to be included in your software project when developing an embedded system – they provide easy access to the Xilinx peripherals, to configure and control them as necessary.

The custom software routines are basically what each member of the team have developed individually and added to the final system. They provide another level of abstraction to the system, where the main application (MainApp.c) can use the custom software routines to perform various actions specific to the application. The Main application will then use the functions in these separate .c files when and where needed to implement a fully functional music player.

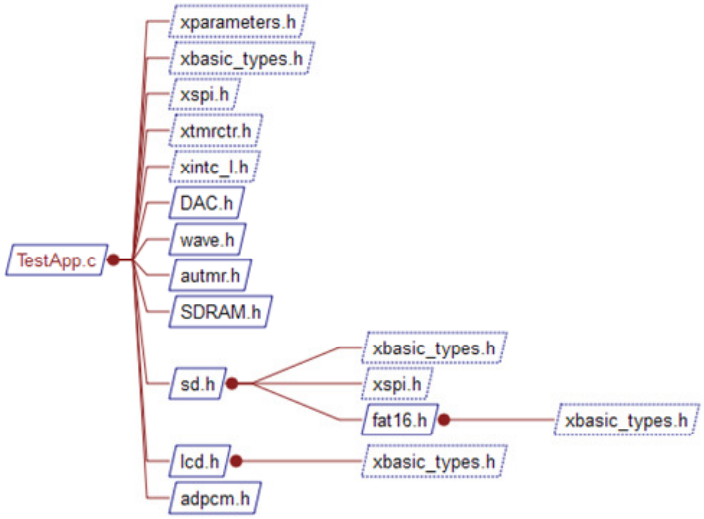


Figure 1.4 - Linking of different components to the Main Application (TestApp.c)

Dividing the separate routines into individual .c (code implementation) and .h (header) files enables to easily integrate different components and also helps in troubleshooting, debugging and isolating a bug. The main application calls the different functions implemented in the separate .c files and the .h files will contain the declarations (as shown in Figure 1.4)

1.4 High level system flow chart

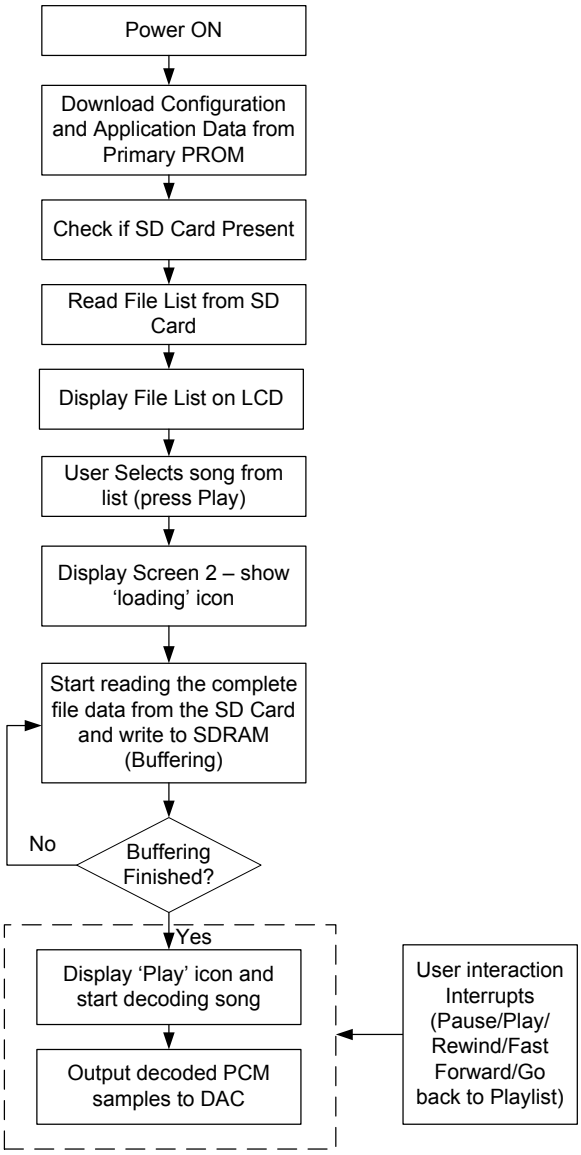


Figure 1.5 - High level system flow chart

- When the user first power cycles the device the FPGA is configured via the PROM and the application is loaded onto the BRAMs – ready for the MicroBlaze to execute.
- Screen 1 is displayed with the names of the files on the SD Card.
- The user can then select a song to play via the rotary encoder and buttons.
- Once a song is selected it is 'buffered' on to the SDRAM. This may take up to a minute due to the i/o speed limitations on the SD Card(SPI Mode). The screen will show a 'loading' icon while the buffering is taking place.
- As soon as the song is finished buffering, it will begin to play and the user can listen to the audio from the DAC.

- The user is then given the option of either stop/pause/play/rewind/fast-forward the song or go back and select a new song.

1.5 Accomplished user requirements

- Ability to 'decode' a previously encoded(compressed) song – ADPCM decoder.
- User can select a song (.wav – WAVE, .adp- ADPCM) from a list of songs stored in the SD Card.
- User can listen to the selected song being played (speakers/headphones should be used).
- User can pause/play/rewind/fast-forward a song.
- User can control volume of a song.
- User can Stop a song, go back to playlist and select a new song.

1.6 Responsibilities and contribution to the group project

The following list highlights the contributions made by the author to the group project during the Implementation/Testing and Integration phases (not in chronological order):

- Created a base EDK project for the custom PCB – which included the basic essential components – MicroBlaze, RS232, GPIO. This was done so that the other members of the group can easily and quickly start developing on their own devices (SD Card, DAC, LCD etc) without having to worry about how to configure a new Xilinx EDK project for the custom PCB.

- Ported the LIBMAD MP3 decoder library to the MicroBlaze Platform as a standalone application.

- Attempted to create a Bootloader to load the MP3 decoder on the Spartan 3E FPGA (which has limited resources when compared with the Virtex 5).

- Spent time understanding XMD and MicroBlaze Caching to try to load and test the LIBMAD decoder onto SDRAM via XMD.

- Ported a 3rd party ADPCM decoder into the Spartan 3E FPGA, and integrated successfully with the other system components. Added in several 'interface functions' for the ADPCM decoder to facilitate integration.

- Successfully integrated and tested MP3 decoder with DAC and SDRAM sub-systems.

- Implemented a user friendly GUI tool to perform ADPCM Encoding.

- Helped to test several components of the new custom designed PCB – JTAG, Monochrome LCD, RS232, DAC and PROMs.

- Researched how to use the wide range of options available in the Xilinx Tools – EDK(Embedded Dev Kit -including command line tools), iMPACT, XPS(Xilinx platform studio)

- SPI was the most common protocol used in the project – to interface with the SD Card, DAC and LCD. Hence several days were spent with the other team members trying to understand how the Xilinx SPI IP core software drivers worked and how they should be configured for communication with a SPI device. This involved connecting and analysing the pins through an Oscilloscope and searching for example code on the internet.

- Helped troubleshoot any bugs in other team members' source code. For example the SD Card routines had a couple of memory issues – assisted to help resolve them.

2 MP3 Algorithm – LIBMAD

This chapter introduces the LIBMAD decoder library, describes the steps taken to port it onto the MicroBlaze Platform, and the tests run on the Virtex-5 ML505 development board.

2.1 Introduction

The Libmad MP3 Decoder library was introduced in the design documentation of the project. To summarise what was mentioned:

LIBMAD has the following special features (taken from the Libmad source code documentation):

- Developed by Underbit Technologies[1]
- MAD (Libmad) is a high-quality MPEG audio decoder. All three audio layers (Layer I, Layer II, and Layer III a.k.a. MP3) are fully implemented.
- 24-bit PCM output
- 100% fixed-point (integer) computation (suited for architectures without a Floating Point Unit)
- Completely new implementation based on the ISO/IEC standards
- Distributed under the terms of the GNU General Public License (GPL)

2.2 Overview of the library

The API of the library can be accessed via the *'mad.h'* header file –which is generated after the library is built. The library can be accessed via the **low-level routines** or the **high-level routines**, the latter was chosen due to it's relatively easier complexity. The high-level routine uses **'callbacks'** in C, where a single 'decode' function can be called to decode an entire bit stream. Another useful feature of the library is it contained the facility to be able to process the MP3 bit stream either *asynchronously* (multiple threads/processes – using fork()) or *synchronously* – the latter option was chosen as asynchronous processing was only possible when running a RTOS (Real Time Operating System).

2.2.1 Integer performance

Several assembly versions of fixed point multiplication for different architectures have been defined in the library. It is up to the developer to choose an appropriate architecture (e.g.: INTEL, ARM, MIPS, SPARC, PowerPC etc). The **DEFAULT** option was chosen as it is the most portable option – but probably not the fastest. Multiplication is the most costly, commonly used operation performed in the library – hence great effort has been used to optimise these routines. The first priority of this task was to be able to 'port' the code to the MicroBlaze platform – optimisation was a secondary goal.

2.2.2 Audio quality

The PCM samples produces by Libmad are of 24-bit precision, hence they had to be scaled down (truncated) to 16-bit to support the onboard DAC (DAC 8532).

2.2.3 Files contained in the library

Table 2.1 gives a brief explanation of the files in the library – it was important to get a clear understanding of the structure of the code before attempting to modify it.

global.h, config.h	Contains options for the library (e.g.: HAVE_ASSERT_H- does the platform have the 'assert.h' C library or not, HAVE_STDINT_H – include the stdint.h library). Most of these features were switched 'off' when porting to the Xilinx platform – to save on space.
bit.c, bit.h	Bit manipulation functions, CRC checks
decoder.c, decoder, h	Handling of the call-back routines, asynchronous/synchronous processing management, initialising the MAD data structures.
fixed.c, fixed.h	Fixed point multiplication, division and absolute routines.
frame.c, frame.h	Decode mp3 header info, decode a single mp3 frame from a bit stream, mute function
huffman.c, huffman.h	Contains the Huffman codeword tables, Huffman data structures (extremely large number of arrays – acting as lookup tables)
layer12.c, layer12.h	Decode functions for Layer I and II. Quantisation tables (many arrays)
layer3.c, layer3.h	Decode functions for Layer III, Scale factor tables (large number of arrays)
stream.c, stream.h	Bit stream manipulation related functions and data structures. Libmad error codes,
synth.c, synth.h	These files contain the most amount of heavy processing – DCT, sub band synthesis, large amount of multiplies, additions and subtractions etc.
timer.c, timer.h	Functions to calculate and manipulate MPEG frame duration.
version.c, version.h	Set of #defines for keeping track of versions, target architecture versions etc.
minimad.c	Test application – which uses the Libmad library for decoding mp3 files.
D.dat, imdct_s.dat, qc_table.dat, rq_table.dat sf_table.dat	Data files – coefficients for sub band synthesis, coefficients/trigonometric values for IMDCT, ISO specified quantisation/re-quantization tables, scale factor values for layer I and II,

Table 2.1 - Explanation of the different .c/.h files used in the Libmad

2.3 Implementation plan – testing on the Xilinx Platform (EDK)

As described in the design document the implementation plan for this task is as follows:

(1) Build the library on a PC and test against different MP3 files.

(2) Gradually remove functionality of the library – to make it "lighter". Some of the features that were cut down are as follows(described in detail in section 6.3.6.5 of the design document)

   a. Only able to support - constant bit rate.

   b. Only able to support - single channel audio

c. Remove Emphasis feature.

d. Remove MPEG Layer 1 and 2 support.

(3) Port the library to the Virtex5 – ML505 XUPv5 Development board (MicroBlaze Platform)

a. Remove all unnecessary standard C header includes.

b. Test if dynamic memory allocation is possible.

c. Replace all printf() calls with xil_printf(). And any other Xilinx specific functions.

d. Check if all data types are supported by MicroBlaze libraries.

e. Verify if the library can be compiled on the MicroBlaze Platform → milestone.

(4) Write a short MicroBlaze test application that will read a very short MP3 file (stored in BRAM), decode it and output PCM samples via RS232 – verify samples against development application running on PC.

(5) Testing the Libmad library on the Spartan 3E board. This was the most difficult section – as different design strategies had to be explored (i.e. Booting from PROM, Loading from XMD, Enabling Cache on MicroBlaze etc.)

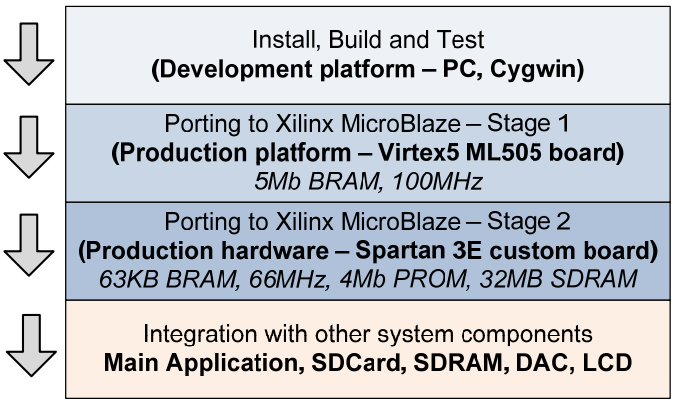


Figure 2.1 - Different stages of porting and integrating Libmad into the final system

2.3.1 Building and installing Libmad (Development platform – PC)

The first step towards porting the library was to build it and test it on a PC. Building the library creates the gateway to the API – 'mad.h'. The documentation had mentioned 'cygwin' was the best platform to build the library hence this was first development platform. Cygwin is a collection of tools (including the GNU tool chain) which provides a Linux look and feel environment for Windows. Steps taken to build Libmad is given below:

(1) Configure for 'default' floating point math routines and disable shared libraries. This executes a script (written in bash, which is very complicated but checks the development environment for various features, and prints out debug messages if any crucial dependencies are missing)

```
Rosh@roshlap2 /home/DSE/libmad-0.15.1b
$ ./configure --disable-shared --enable-fpm=default
```

(2) Type '**make'** to compile the package. This will create the individual object files and most importantly the 'mad.h' header file that can be linked to your project to access the Libmad API.

(3) Create the 'minimad.c' test application – this executable will read in a MP3 file from 'stdin' and output to 'stdout' 16-bit PCM samples. The plan is to decode the exact same MP3 file on the Spartan 3E board and verify the PCM sample output against the 'minimad' application output.

```
Rosh@roshlap2 /home/DSE/libmad-0.15.1b
$ make minimad.exe
```

The minimad application was run in the following fashion (takes a file from stdin and outputs the PCM data to another file.)

```
Rosh@roshlap2 /home/DSE/libmad-0.15.1b
$ ./minimad.exe < speech_utterance_24000Hz_64kbps.mp3 > speech_200811.pcm
```

2.3.2 Removing features from Libmad

The main objective for removing functionality was to try and reduce the executable size as much as possible. After each feature had been removed the entire library was tested against Libmad – to verify that the removal had not introduced any errors or bugs.

Step	Description	Size reduction (compiled)
1	Libmad 'default' version(FPM=default) compiled and tested against minimad.c. The original version would naturally compile according to FPM=INTEL (because the PC architecture was Intel Dual Core)	15 KB reduction in library size.
2	All unimportant headers were removed in all c/header files: <sys/types.h>, <sys/wait.h>, <unistd.h>, <fcntl.h>, <errno.h>.<limits.h> Care was taken if any functions/data types from these standard C libraries were being used. Hence <stdlib.h>, <stdio.h>, <string.h> was not removed and kept till the next step.	18KB reduction in library size.
3	Replaced 2 important standard C functions – memcpy, memmove. These were used in Layer3.c and are defined and implemented in <string.h>. There are many implementations of these standard functions. An implementation for memcpy and memmove was obtain from [2]. Removed <stdio.h> header from timer.c	2KB reduction in library size
4	Removed function calls to Asynchronous processing (remove usage of fork(), functions written to pass messages etc). This was relatively simpler, as the code was organised properly using #defines (#define USE_ASYNC 0)	2 KB reduction in library size
5	Removed Layer12.c (MPEG Layer I and II functionality) from the library. This involved modification of some MAD data structures and any reference to Layer12.h. The makefile of the library was also modified to complete unlink the Layer12.c (.h) from the linking and compiling process.	9KB reduction in library size.

Table 2.2 - Steps for removing functionality from LIBMAD

The final version contained only 1 standard C library header <stdlib.h>, for using malloc() and free() which according to the Xilinx documentation [3] and [4] have been implemented in full and will only cause a 4KB increase in compiled code size. Hence this header did not seem necessary to be removed (and would have been cumbersome to convert dynamic memory allocation to static memory allocation). **The final library size compiled on cygwin was about 192KB (libmad.a – archived format)**

Multiple MP3 files were tested against this version of LIBMAD – as described in the Test and integration plan document. The output PCM samples were played in Gold Wave to verify if the audio was correct. As a reference – a 24Khz 16bit 2 minute MP3 file was used – which contained various tones of music. Loudness, any loss of treble/bass tones and pitch were checked qualitatively.

2.4 Compiling the Library on the MicroBlaze Platform

After reducing the code size as much as possible – the next step was to try to compile the code on the MicroBlaze platform (on the EDK). The first goal was to be able to execute the code on the Virtex 5 board , and if successful proceed to the Spartan 3E chip.

A quick check on the Virtex 5 datasheet [5] showed that the Virtex 5 XC5VLX110T chip had over 5Mb (666KB) of BRAM available – which meant the Libmad library can be run on the BRAM easily without the use of any external memory.

2.4.1.1 Difficulties faced when porting to the MicroBlaze platform

Difficulties faced	**Solution**
Libmad produces a set of object files (.o) and a archive file (.a) using 'GNU libtool'. This was a fairly new concept and hence a bit of research was done to understand how libtool worked. Essentially it is a tool that can be used to create shared libraries. These shared libraries can then be linked with other third party applications without the need of distributing the actual source (the library is pre-built).	Linking the pre-built libraries seemed complicated and hence the Libmad code (.c, .h, .dat) files were directly imported to the EDK and compiled from scratch. However learning how to use shared libraries would have been a valuable skill to possess.
The EDK compiler complained about the 'assert' functions.	The assert statements were replaced with Xilinx specific '**XASSERT_NONVOID'** functions.
Since the library was built from ground up, certain	**#define FPM_DEFAULT 1** had to be included in the

files required knowledge of the architecture (for Fixed point multiplication)	fixed.h file to allow proper compilation
The compiler complained about the following statement: mad_fixed_t **const** (*sbsample)[36][32];	The 'const' keyword was removed to fix this issue.
The EDK project was first created with 32KB LMB BRAM. The entire code did not fit into this space.	**BRAM** size had to be increased to **256KB** – as seen in Figure 2.2
After the first compilation and running the application the execution failed when malloc() was being called	After further investigating the code it was found that the **malloc()** was being done for a quite large data structure (*sync). When calculated the size of sync was about **30KB**, and the heap had to be increased to accommodate the malloc call to this structure – as seen in Figure 2.3

Table 2.3 - Difficulties faced when porting Libmad to Xilinx platform

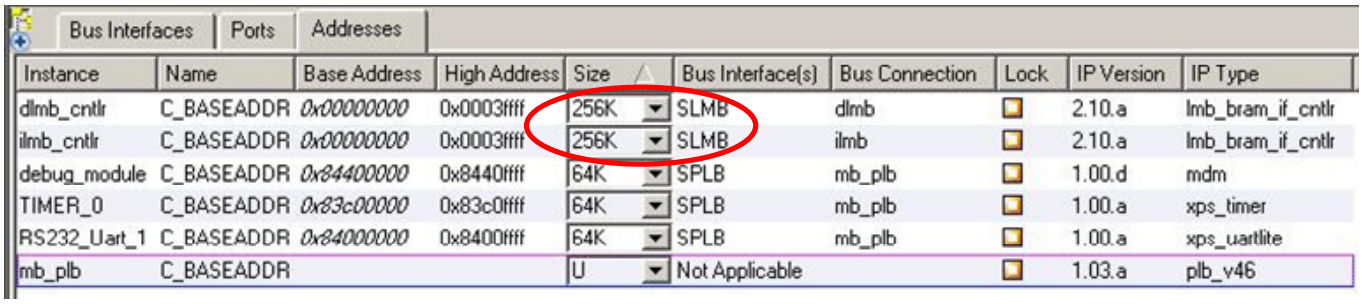


Figure 2.2 - Increasing BRAM size to accommodate the large Libmad code

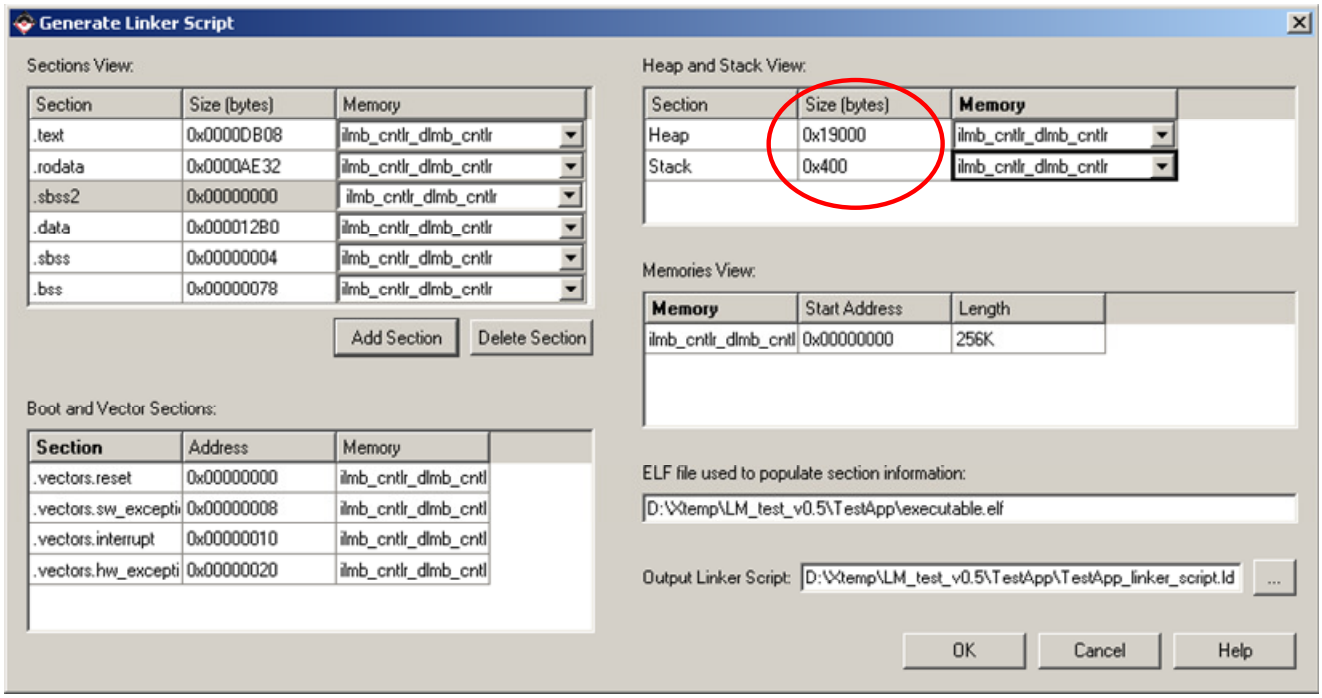


Figure 2.3 - Increasing Heap and Stack limits to be sufficient for Libmad

2.5 Testing and results

The test application(TestApp.c) which utilises the Libmad Library can be found in the Appendix 9.6 of this document. The application basically stores a short MP3 clip in a C header and feeds it directly to the decode() function in Libmad. The test application is largely adapted from the minimad.c application that is bundled with Libmad. The major modification that was made was replacing the printf() output functions with xil_printf() – which essentially sends an output to stdout(RS232).

The 'bin2h' tool (freeware) [6] was used to convert the mp3 file into a C-style header. Which was then included in the TestApp and directly fed into the decoder. The PCM output of the TestApp running on the MicroBlaze is then logged onto a text file, converted into a binary file and played in the Gold Wave software (evaluation version [7] which recognises raw PCM data).

The test mp3 file, was a speech utterance of "turn" which was originally a 24Khz, 16bit, 0.43seconds, mono wav file compressed into MP3 using Gold Wave (using LAME mp3 codec). A short Python script was written to convert the PCM sample output strings (taken from RS232) into a binary file that Gold Wave can read.

The test setup is shown in Figure 2.4, as explained in the design document – however time permitted only the simplest test (a block of mp3 file saved as a header) and there was not enough time to test input via UART (which would have facilitated testing longer MP3 samples)

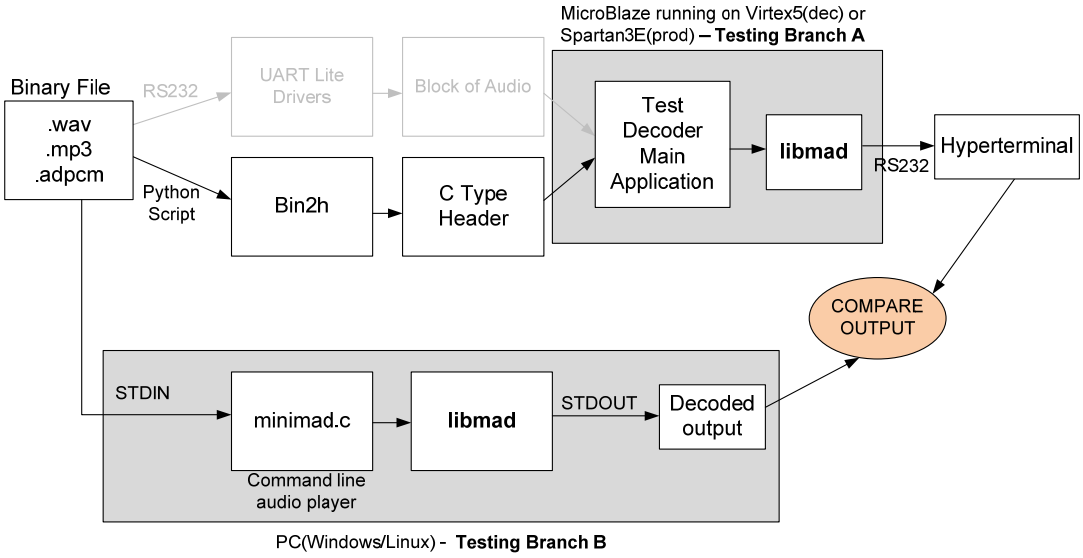


Figure 2.4 - The flow used to test the Libmad library on the Virtex 5 board

2.5.1 Results from running LIBMAD on the Virtex-5 ML505 board

Appendix 9.1 of the report shows the a detailed breakdown (and explanation) of the section headers of the final executable.elf – using *'mb-objdump'* tool in the EDK. Using *'mb-size'* it was possible to

obtain the final size of the code (.text, .data, .bss etc). Table 2.4 shows the final size of the TestApplication (executable.elf).

text (program instructions)	**data** (read-write data)	**bss** (uninitialized data)	**dec** (Total size in decimal)	**hex** (Total size in Hex)
100766	4808	103548	**209122**	330e2

Table 2.4 - LIBMAD Test executable size, obtained using mb-size (Bytes)

Hence according to this, the entire library (minus some overheads introduced by Xilinx functions) is about 205KB large. However about 4KB of this is the test MP3 speech-sample-file converted into a C Header as raw hex data. Hence the total memory required by LIBMAD is about **201KB**. This clearly proved the amount of BRAM available in the Spartan 3E FPGA is not sufficient to hold the decoder program code.

2.5.1.1 Checking MP3 decoding latency

The next test was related to timing. It was necessary to find out how fast the decoder performed on the Virtex-5 board running with sufficient resources. To calculate the time taken to decode a MP3 bit stream the following setup (utilising a hardware timer – XPS_TIMER) shown in Figure 2.5 was employed.

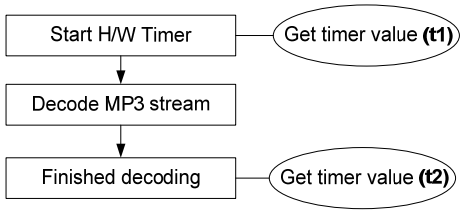


Figure 2.5 Calculating the decoding time (on Virtex-5) using timers

The clock cycles elapsed during decoding the MP3 data is given by *t2-t1*. And for the test done on the Virtex-5 chip the results were as follows:

t2-t1 = 18653383 system clock frequency = **100 MHz**

Therefore the elapsed time = 18653383 * (1/(100^6)) = **0.186533 seconds.**

Duration of the MP3 audio clip = 0.432 seconds. Hence, for **1 second of MP3 it would take 0.4317 seconds** to complete decoding.

Hence the test confirmed that the Virtex 5 chip has both the speed & spatial requirements necessary to decode MP3 in real-time. Doing a quick calculation for a **66MHz** (our reference oscillator clock on the custom PCB), showed that for such a clock frequency it would take 0.5784 seconds to decode a 1 second MP3 stream. In the Design document – section 5 (feasibility analysis), a rough requirements

analysis for real-time mp3 playing was outlined. Figure 2.6 illustrates this requirement and the latencies involved.

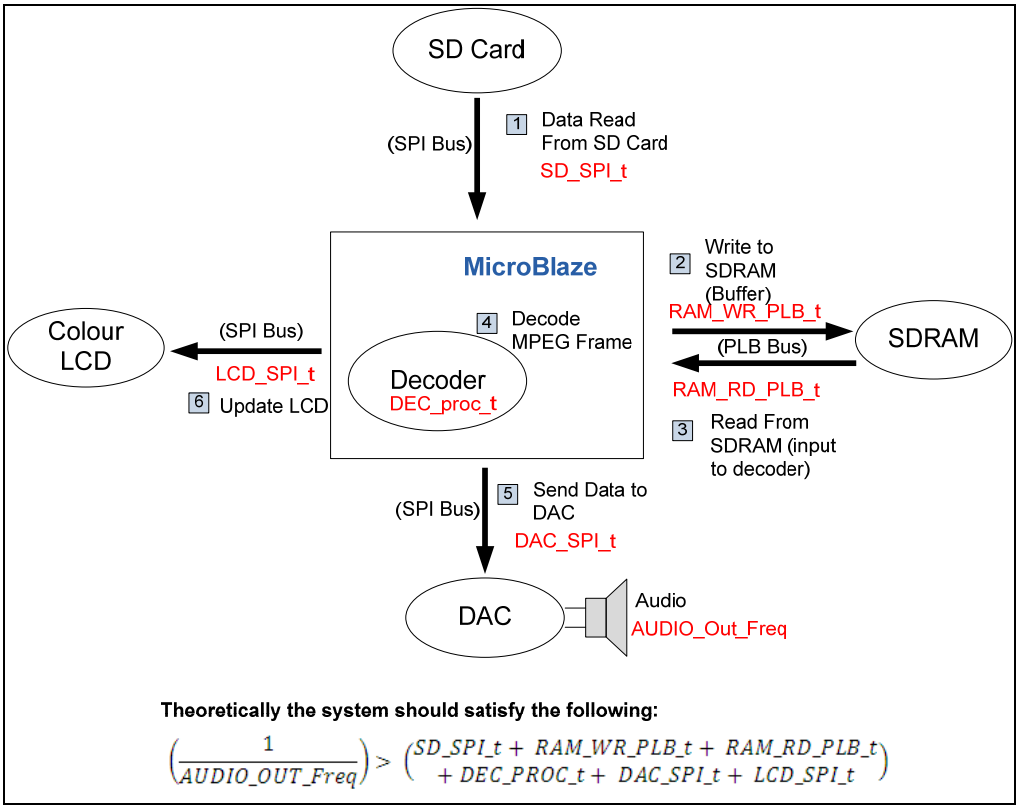


Figure 2.6 - Theoretical requirement for MP3 decoding and playing in real-time (taken from Design Doc.)

A large portion of time and effort was spent trying to execute and test the library on the Spartan 3E platform with limited resources. Therefore a separate section of this report has been dedicated to explain the effort taken to reach this goal.

2.6 Learning Experience

- Understanding how to read and understand a large C library.
- Understood the concept of callbacks in C.
- Gained experience in Modifying→Testing of a third party code base. Successfully reducing functionality of a third-party C code.
- Gained experience in some of the GNU tools used in open source projects: libtool, makefiles, shell scripts, cygwin etc.
- Experience in performance benchmarking code on the MicroBlaze platform.
- Experience in using the MicroBlaze specific EDK command line tools (mb-objdump, mb-size etc)
- Exposure to linker scripts, changing BRAM size, changing stack/heap size. Stack/Heap size limitations in embedded systems.

3 Running LIBMAD on Spartan 3E

This section of the report describes the different attempts taken to execute and test the Libmad library on the Spartan 3E platform.

3.1 Design flow

3.1.1 Problem definition

The primary problem faced in this situation is the lack of **BRAM** available on the Xilinx Spartan 3E chip (**63KB**). The compiled Test application described in Chapter 2 of this report confirmed the size of the **application** to be **201KB** (MP3 decoder library – Libmad, small test application and a MP3 speech sample). Not enough memory to store the application is a common hurdle encountered in embedded system design.

3.1.2 Feasible design solutions

The only feasible solution is to somehow store and execute the entire application from external memory (SDRAM). Two possible design flows were explored:

(1) **Load the entire application from the secondary PROM**. Configure the FPGA with a 'bootloader' application which upon boot-up copies the application from PROM and stores(writes) it to the SDRAM. After loading the program into the SDRAM the processor should then start executing form the SDRAM. This design will not have any dependency on a host machine and can therefore operate independently of any other computer system(stand-alone embedded system), and probably the method used commercially. Figure 3.1 illustrates this design flow.

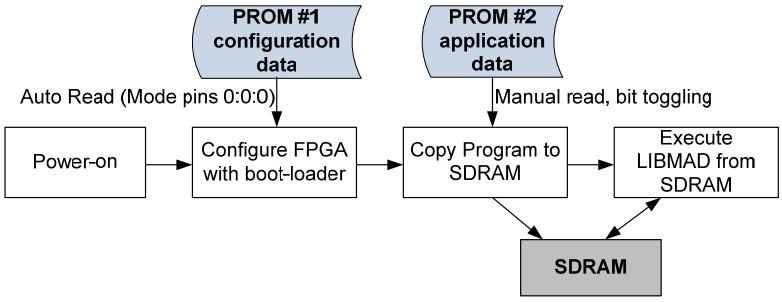


Figure 3.1 - Design flow for a bootloader application

(2) The second option is to configure the FPGA with the default 'boot_loop' application which essentially keeps the processor in an infinite loop. Then **use XMD (Xilinx Microprocessor Debugger) to download the main application to SDRAM**. The processor can then be instructed to execute the application from SDRAM. This process is useful for debugging since one can quickly re-download new executables via XMD. However once power-cycled the

application will need to be re-downloaded via XMD. This flow depends heavily on a host-machine, and is not a stand-alone embedded system. Figure 3.2 illustrates this design flow.

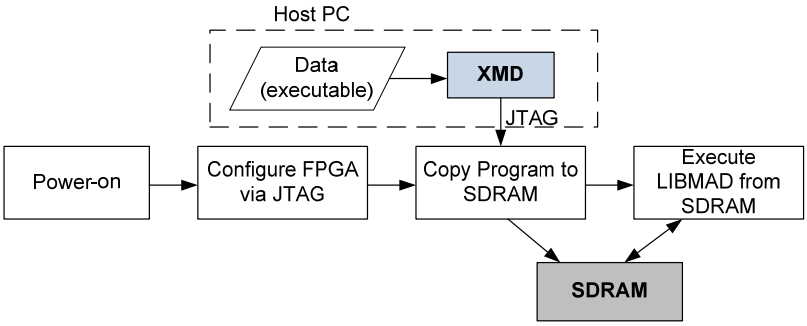


Figure 3.2 - XMD based design flow for storing and executing application in SDRAM

3.2 Implementation of Solution (1): Bootloading from PROM

The first step was to research for some reference designs on the Xilinx website. Xilinx has published a couple of reference designs and documents [8], [9] and [10] facilitate the implementation process greatly – specially[8] – XAPP482.

[8] XAPP482 describes a system where the software code, user data and configuration are all in one PROM, it explains how to load applications from a PROM, how to partition a PROM and has provided reference code for a bootloader written for the MicroBlaze processor. For our design we will not store the FPGA configuration in the second PROM but only the application code. The steps for this task are as follows:

(1) Store the application code in PROM (a simple test application was written to test the concept – outputs "HelloWorld" through RS232)
(2) Write a bootloader application in C where it reads the PROM (serially) and stores the data (word by word) onto the SDRAM(from the first location – 0x86000000)
(3) Once storing is complete the application should 'jump' to the first address of the SDRAM (the start address of the program code).
(4) If the TestApplication is able to run from the SDRAM, then replace the test app with the actual Libmad library and test application.
(5) Reset the Reset and Interrupt Handler memory locations (so that interrupts can be serviced properly)

3.2.1 Storing the application code in PROM

Data in a PROM is stored as a **Intel MCS-86 Hexadecimal Object (.mcs)** file and the structure of such a file is explained in [11]. Fortunately the Xilinx tool – iMPACT is able to generate a MCS file from a .bit file(EDK generates the bit file from the executable to download the program to an FPGA)

First a marker has to be put before the application data – to indicate where the application starts from (as described in [8]).

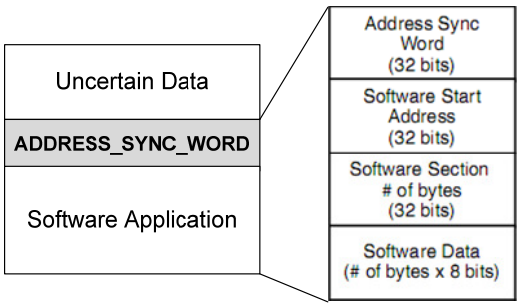


Figure 3.3 - Adding a application start marker to the PROM MCS file (adapted from [8])

The bootloader application will have to read through the PROM until this ADDRESS_SYNC_WORD (32-bit word = 0x9F8FAFBF) is found (Figure 3.3), and then copy all data from that point till the end word 0xFFFFFFFF word is found. XAPP482[8] also includes a tool that can insert a ADDRESS_SYNC_WORD into the MCS file. Adding the software sections to the MCS file is illustrated in the diagram given in Figure 3.4 (adapted from [8]). iMPACT was used again to download the MCS file to the 2nd PROM.

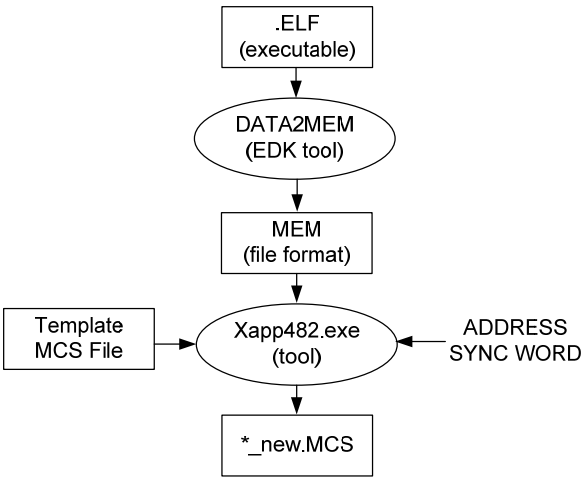


Figure 3.4 - Flow for adding the Address sync word and the software application to the MCS file (extracted from [8])

3.2.2 Reading from the PROM

PROM connectivity is **serial** and data has to be read on a bit-by-bit basis. XAPP482 [8] and XAPP544[9] explains how to read the PROM. The schematic given in Figure 3.5 shows how the two PROMs are connected in our custom PCB.

As seen in Figure 3.5 the two PROMs share a common Clock (PROM_CLK) and Data (PROM_D0) line. Hence the first PROM must be disabled (INIT_B=0 produces High-Z on PROM1) when reading from the second PROM. Initial tests were done without disabling the first PROM and as a result the data was all 1's (incorrect) – debugging this involved rechecking the schematic and reading through the datasheets.

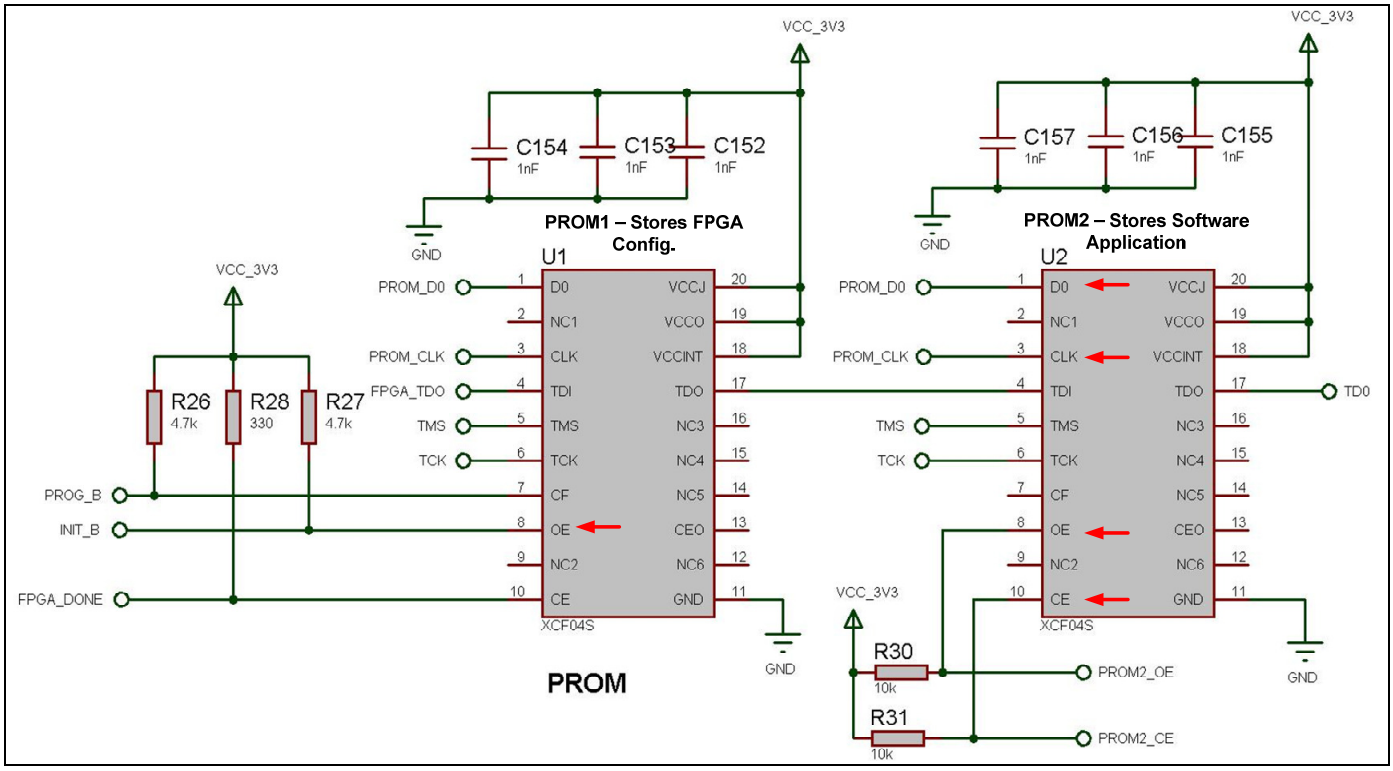


Figure 3.5 - Schematic of the PROM connections

The PROM control lines that must be manipulated by the bootloader application to read the 2nd PROM (U2) are as follows:

PROM2_CLK = bit toggle (1 | 0), active High.
PROM2_CE = 0 (low)
PROM2_OE = 1 (high)

This was done using Xilinx GPIO IP Cores within EDK. 3 GPIO connections were needed one to control the OE/CE/CLK of PROM2, another one to obtain the data input and the final one to disable PROM2. The software example given in XAPP 482 [8] uses obsolete IP Cores (used XIo), hence only the logic of the code was taken, and updated to suit the new IP Cores.

Essentially, after every toggle of the CLK, the PROM_D0 pin would have a data bit. This is then shifted into a 32 bit wide buffer(Xuint32 variable, acting as a shift register). Once the buffer is full the contents of the buffer is written to an address in SDRAM. The process is illustrated in Figure 3.6.

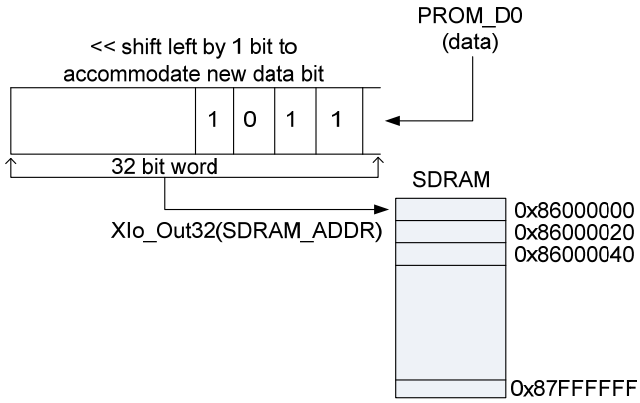


Figure 3.6 - Reading from PROM and storing in SDRAM

3.2.3 Changing the program counter

After all the application data has been written to the SDRAM – a jump was made to the start address of the application (located in SDRAM). This was done as follows (taken from XAPP482[8]):

```
//declare before main()
// Function point that is used at the end of the program
// to jump to the address location stated by PROG_START_ADDR
#define PROG_START_ADDR 0x80180000
int (*func_ptr) ();
// declare after main()
// function point that is set to point to the address of
// PROG_START_ADDR
func_ptr = PROG_START_ADDR;
// jump to start execution code at the address
// PROG_START_ADDR
func_ptr();
```

This essentially converts into an assembly code branch instruction. However this did not seem to work. After the jump the booloader application **reset itself**. Further investigation led to the point that the Linker script should be changed to reflect the new _start address of the program (to indicate the program starts from SDRAM_BASEADDRESS (0x86000000). When this was done, the program seemed to just crash after the jump instruction. A couple of reasons why this may be happening is given below:

- MPMC could be swapping the words when transmitting to the SDRAM.
- The program counter must not be set properly.
- An issue with the linker/linker script.

Since time was limited – the next method was explored – Solution (2), using XMD. The test code for the PROM bootloader can be found in the Appendix 9.7

3.3 Implementation of solution (2): Loading from XMD

XMD can be used to interact with the microprocessor while your application is being executed. Setting breakpoints, stepping through code, viewing contents of registers/memory locations, downloading programs etc are a few of the functions that can be performed using XMD – it's a valuable software debug tool.

This method was much more straight-forward than loading from PROM. The documents [12] and [13] provides good instructions on how XMD should be used to download the program. The following steps were followed:

(1) Build a new EDK project with the **On Chip H/W debug module** enabled.
(2) Set the **Program Start Address** to the base address of the SDRAM.
(3) Download a dummy bootloop application to the FPGA.
(4) Connect XMD to the MicroBlaze and download the executable.elf and restart the processor. This is done by issuing a few commands specified in [12] and [13]. Figure 3.7 shows XMD being run and connecting to the processor.

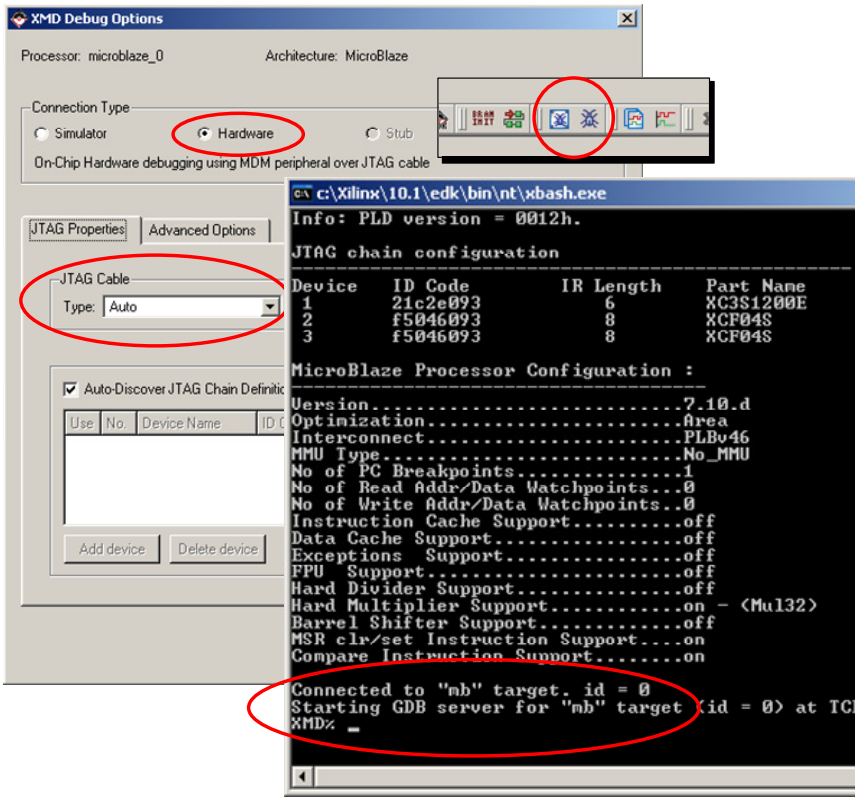


Figure 3.7 - Connecting to the MicroBlaze via XMD

The following two tests were performed to verify the operation of the LIBMAD library on the Spartan 3E FPGA.

(1) Calculate execution time taken for the decoder (exact same test performed in Chapter 2 on the Virtex-5). The design flow of the test was exactly the same as shown in Figure 2.5(Chapter 2)

(2) Integrate DAC and listen to the output to determine if output is audible/recognisable.

The test configuration involved the following configuration settings – as shown in Figure 3.8

- Program Start address should be set as the base address of the SDRAM.
- Stack and Heap should be increased (previous stack size didn't seem to work)
- Mark the bootloop application to run from BRAM, then download the new application runnable from SDRAM via XMD.

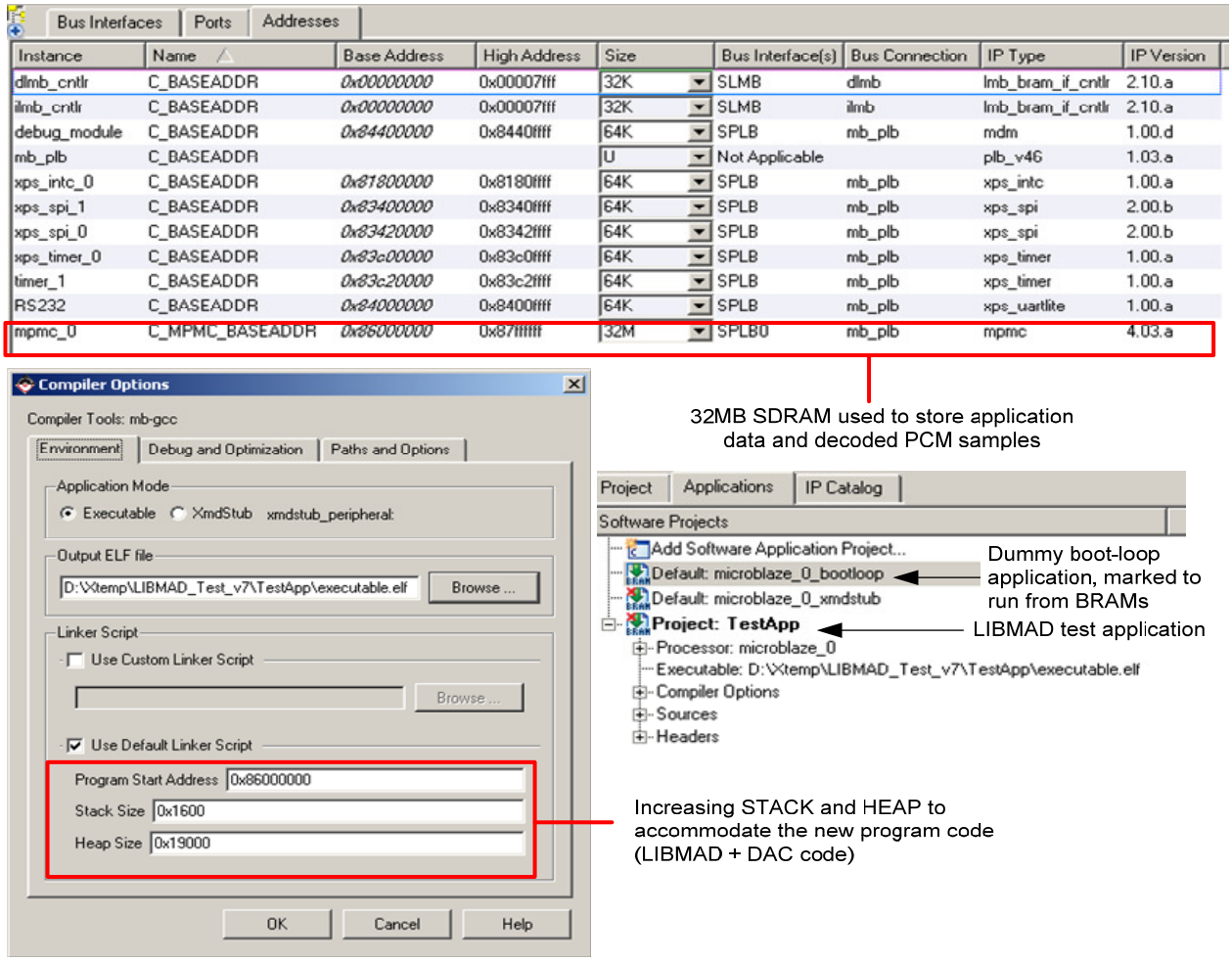


Figure 3.8 - Necessary configuration changes being made on the project to execute from SDRAM

A detailed trace of the XMD session can be seen in Appendix 9.2

3.3.1 LIBMAD Decoder test results – execution on Spartan 3E

The test results were disappointing on two levels:

(1) The execution time of the decoder had reduced greatly. The timer value after decoding was :

t2-t1 = **776639075**, clock speed = **66MHz**.

Hence duration = **11.7673 seconds** to decode 0.432 seconds of MP3 data.

Two possible reasons could have effected this large processing delay:

- The system **clock** was **reduced by 34%** (when compared with 100MHz clock on Virtex5)
- The program is now running from **SDRAM** – much **slower memory** than BRAM.

Hence it was clear that MP3 decoding was too slow with the current resources and, decoding→output to DAC could not be done in real-time. One solution explored was to store the data of the 'decoded' PCM samples in the SDRAM 'after' decoding. Hence the SDRAM had to be partitioned logically as shown in Figure 3.9:

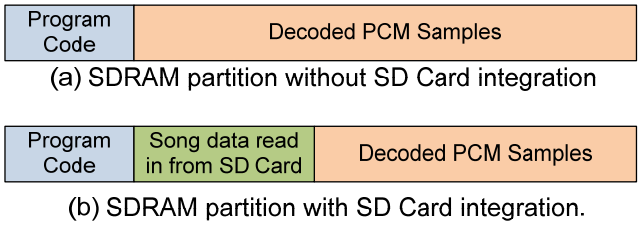


Figure 3.9 - Two possible SDRAM partitioning methods for a MP3 Player

Out of the two methods displayed above – method (a) was used as a quick test to listen to the audio decoded by the MP3 decoder. Method (b) can be used in final integration if (a) succeeds. The flow of the test is shown in Figure 3.10

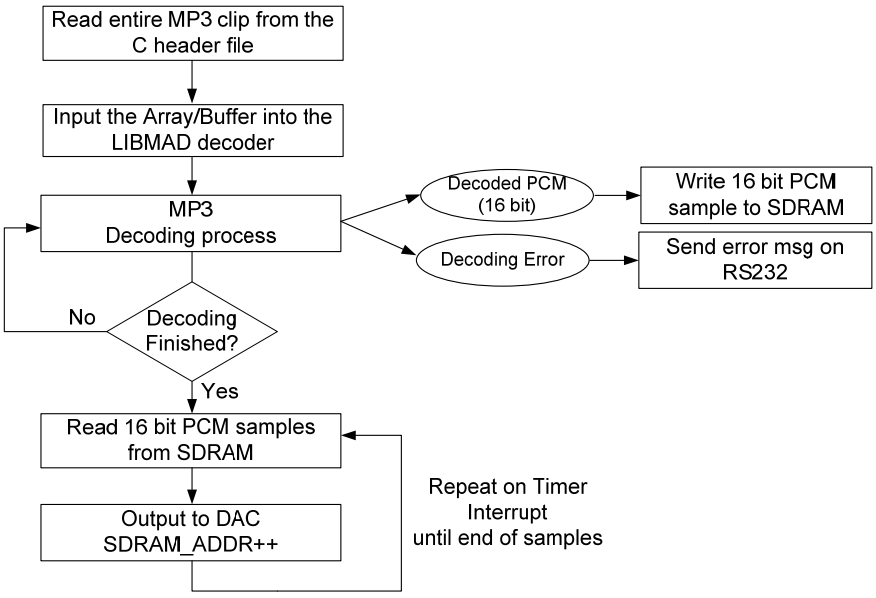


Figure 3.10 - Test flow of LIBMAD Decoder integrated with DAC

(2) **Test Results:** The sound output by the DAC was distorted – but the pitch and tone were slightly similar to the word *"turn"* (the MP3 speech sample). Interestingly - the PCM samples were captured and played on Gold Wave to confirm it's correctness – and they were correct and audible. Hence it was very strange that the sound output coming from the DAC were not correct. It was then clear that the bottleneck now was the *Fetch – Decode – Execute – Write Back* cycle was being done directly with the SDRAM (slow memory) and hence it was probable that the instruction execution latency of the processor was too slow – to match the required sampling rate of the sound clip.

3.3.1.1 Enabling Cache on the MicroBlaze

One solution was to explore caching on the MicroBlaze. If caching was used, then probably certain instructions would be stored temporarily in BRAM (*'Principle of Locality'*[18]) – hence should in theory speed execution. From the resources on the internet – it seemed only a configuration change was needed on the EDK project to 'enable cache'. The MPMC datasheet and the MicroBlaze reference manual had to be studied to understand how to enable cache, also the Xilinx forums were searched. The screen captures shown in Figure 3.11 and Figure 3.12 show the various configuration settings that had to be adjusted to enable caching on the MicroBlaze.

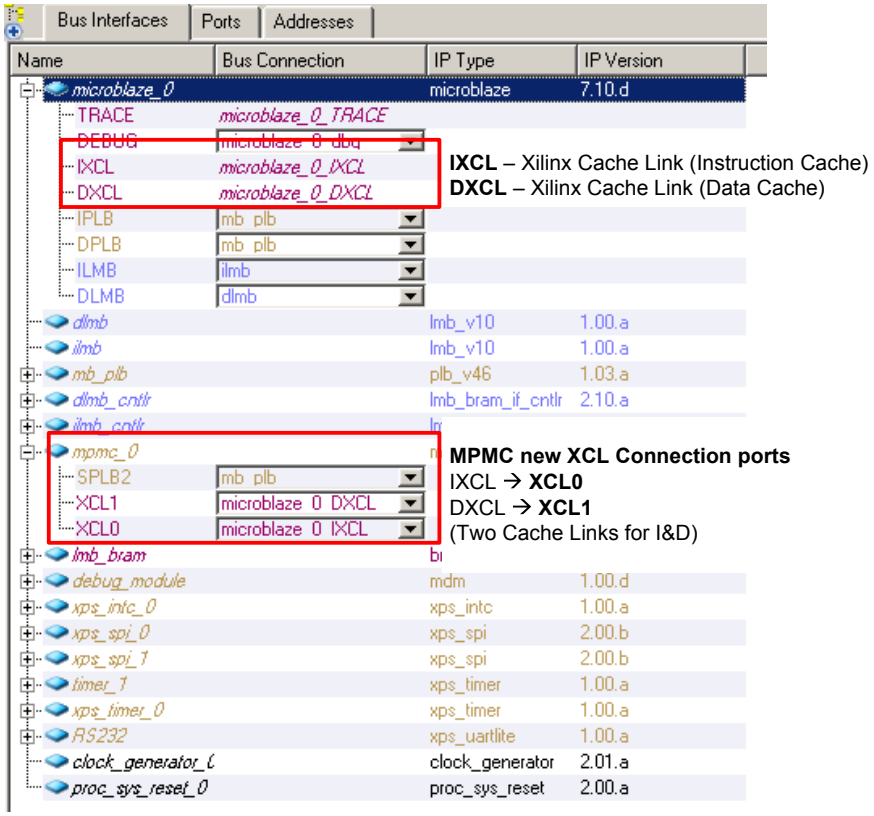


Figure 3.11 - MicroBlaze and MPMC XCL Connections

Two XCL (Xilinx Cache Links) were needed for each Cache type (I=Instruction, D=Data cache). The MPMC (Multi Port Memory Controller – SDRAM Controller) needed to be configured so it has 3 buses – 1 PLB and 2 XCL. The further configuration settings made in the two IP cores are displayed in Figure 3.12.

Instruction Cache = 8KB, Data Cache = 8KB.

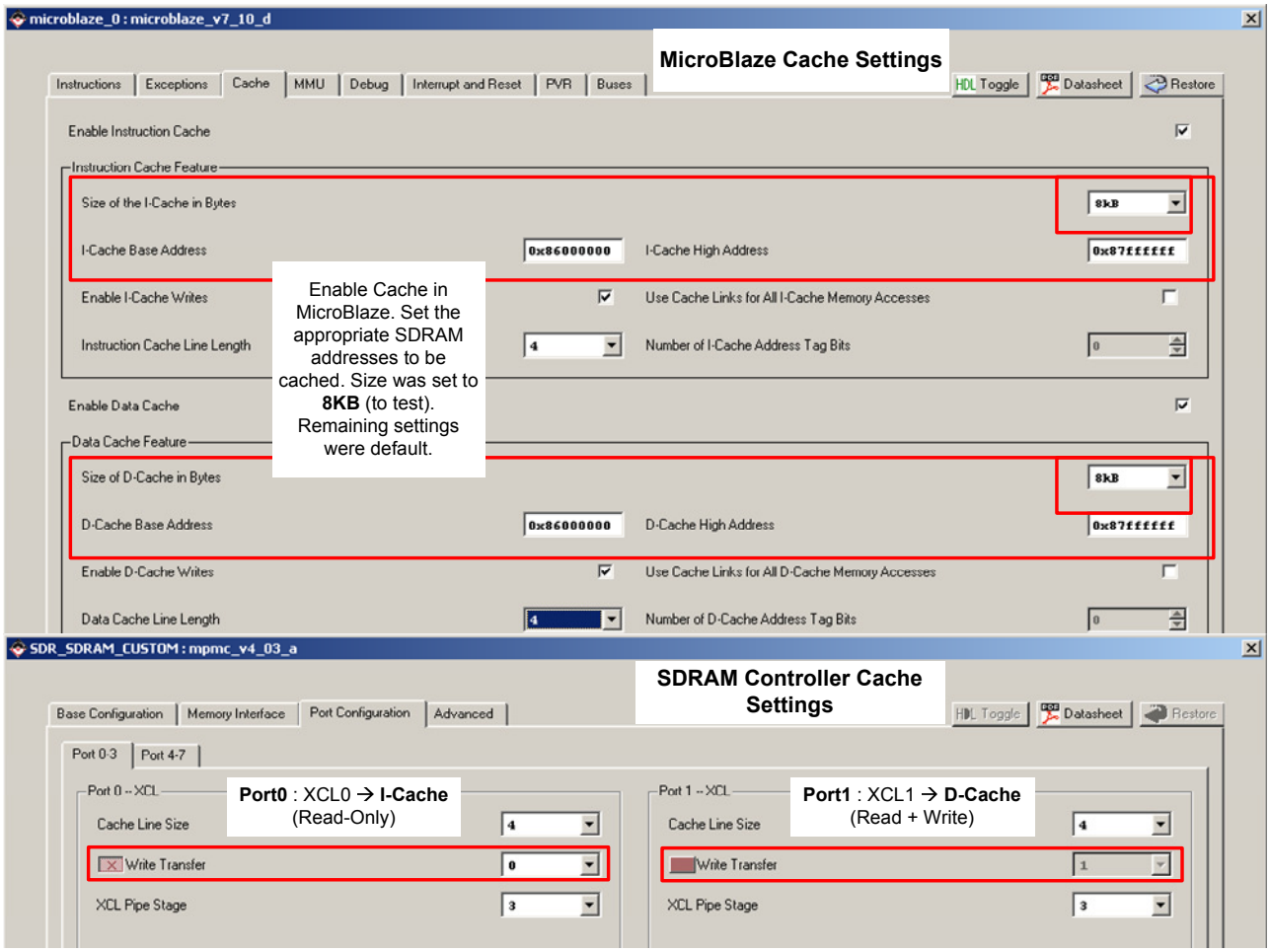


Figure 3.12 - MicroBlaze and MPMC Cache settings

On chip BRAM usage had to be reduced to 16KB to accommodate the additional BRAM required for the cache. Once these settings were done cache needs to be enabled in the software – by using the commands *"microblaze_enable_icache()"* and *"microblaze_enable_dcache()"*, which enables both instruction and data cache. Once these changes were done and cache was properly enabled, the decode timings were checked.

3.3.2 Test results from Libmad running on Spartan 3E (with MicroBlaze Caching enabled)

The results obtained showed a significant improvement in execution time overall. The previously carried out timer test now yielded the following results:

(t2-t1) = **34855049**, clock speed = 66MHz

Hence **0.528 seconds to decode a 0.432 seconds of MP3** data. However this is still slower than required - for real-time decoding = (decode speed per second of MP3 audio < 1 second).

*Also the DAC output was now clear and audible. Further tests were carried out with higher sampling rates (tested 8Khz, 16Khz and 24Khz). All performed successfully – no audible loss of samples.

However the significant performance increase when cache was enabled was very good news. Hence the following modifications were made to the cache:

Instruction Cache = increased to 16KB, Data cache = 8KB, **BRAM reduced to 8KB**. BRAM can now be reduced to a minimum, as only SDRAM is being used for processor memory.

The test was carried out and the new timings were as follows:

(t2-t1) = 27015737, clock speed = 66Mhz,

Hence **0.40932 seconds** to decode 0.4332.

This was again an improvement to the previous test. And now it was evident MP3 decoding can be done in real-time. Now the important questions is, would the system satisfy the requirement shown in Figure 2.5 - when all the inter-component communication latencies are taken into account? However further improvement/tests could not be carried out as time was running out.

	Time taken to decode (0.432s of MP3)	Time taken to decode (1 second of MP3) - extrapolated
Decoding on Virtex-5	0.1865 sec	0.4317 sec
Decoding on Spartan 3E (code execution from SDRAM, no cache)	11.7673 sec	27.2391 sec
Decoding on Spartan 3E (code execution from SDRAM, and cache enabled, I&D Cache = 8KB)	0.5281 sec	1.2222 sec
Decoding on Spartan 3E (code execution from SDRAM and I-Cache increased to 16KB)	0.4093 sec	**0.9475 sec**

Table 3.1 - Summary of Libmad decoding test results (latency), improvement in every step

3.3.2.1 Profiling analysis on the Spartan 3E – using 'mb-gprof'

'mb-gprof' is Xilinx version of the GNU profiling tool. A brief description on profiling is given in the design document. 'mb-gprof' basically uses a hardware timer to measure the execution speeds and other statistics of each C function in the application. The recorded data is then stored in a separate area of memory and can be downloaded to the PC via XMD. The purpose of this test was to find the bottleneck in the decoding process. The Xilinx profiling guide [14] explains the steps necessary to perform this task.

(1) A hardware timer must be added to the project and connected to profile interrupt handler – this samples the Program Counter at every interval. Software support has to be enabled via the C application at the start of execution.

(2) Using XMD the sampling frequency and profile memory location must be given.

(3) Run the program at least once. And set a breakpoint for the profiling to stop.

The results obtained were very interesting and can be seen in Appendix 9.3. A small subset of the results are shown in Figure 3.13.

To summarise – the results showed the bottleneck was the ***'__mulsi3'*** (integer multiply operation – software emulation routine) and ***'imdct'*** (inverse-modified-discrete-cosine-transform function) which are computationally expensive tasks. The results closely matched the other literature found on LIBMAD – explained in the Design Document (Section 6.3.2.3).

Note: xil_printf() prints out to RS232 (serial transfer) and probably a blocking operation – hence the large amount of time taken.

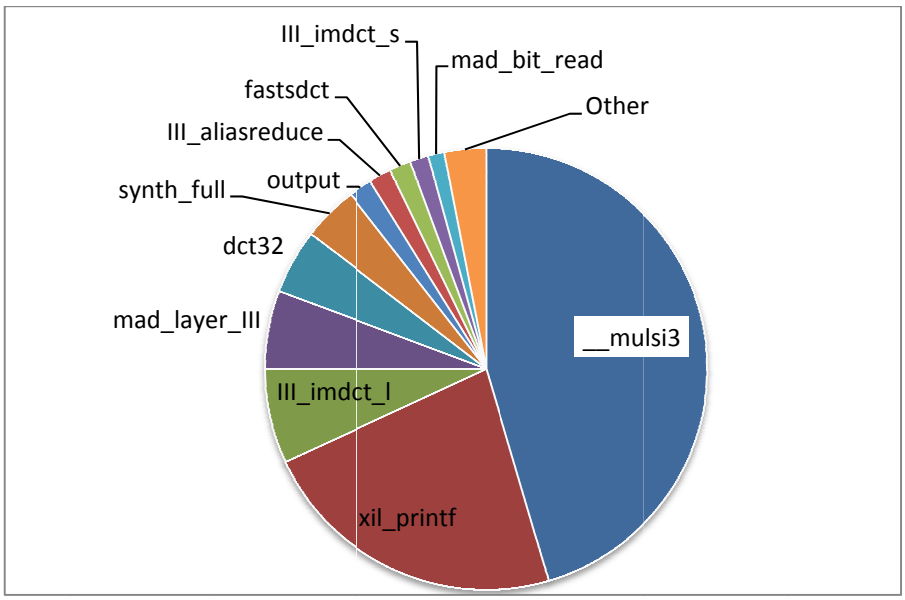


Figure 3.13 - LIBMAD profiling test results (function processing durations) showing how expensive 'multiplication' is

3.4 Learning experience

- It was now clear that software based MP3 decoding is an extremely complicated processing task for a small microprocessor like the MicroBlaze. However results from the various tests run – using caching on the MicroBlaze shows than the best achieved decoding time of a 1 second MP3 clip is less than 1 second (0.9475 sec). This result is promising and with further optimisations it maybe possible to use Libmad to decode a MP3 file AND output to the DAC in real-time (without taking into account – time taken to service any intermediate user-level tasks)

- Gained much understanding of PROMs, ROM file formats (MCS, HEX, BIN etc), the process involved when iMPACT downloads configuration and program data to the FPGA.
- Using an oscilloscope to probe external pins – understanding serial synchronous communication.
- Exposure to writing a custom boot-loader. Concepts of 'boot-loading' why it is needed. Using mb-objdump to disassemble an executable file – into Assembly code. What each instruction in C gets converted to in assembly language.
- Using mb-object dump and other Xilinx command line tools to dissect an executable – to see which functions/data objects are taking up most space.
- Usage of XMD & JTAG - to debug, to download code, to set breakpoints and view memory.
- How to profile code – using mb-gprof. Learning how to profile your code is a quick way to find bottlenecks and to test performance of the code – the execution times may not be accurate – but it definitely helped to understand 'why' MP3 decoding is such a complicated process to implement in software. Commercially MP3 decoding may be performed in Hardware/using low level routines (assembler).
- Using cache in a micro-processor. Performance increases gained by using cache – Instruction/Data. In-depth exposure to the MPMC SDRAM memory controller.
- Partitioning external memory – for various design requirements. Reading/Writing from SDRAM.

4 ADPCM Decoder

This section describes the final decoder used in the music player – ADPCM. The steps taken to integrate it into the final system is described as well as tests performed.

4.1 Introduction

ADPCM(Adaptive Differential Pulse Code Modulation) was the backup decoder choice made by the group at the design stage. It was briefly introduced in the Design Document(Section 6.3.11). The type of ADPCM used in the project was the IMA(Interactive Multimedia Association) format. A few important points regarding ADPCM are given below:

- ADPCM is based on encoding **differences** of successive PCM samples, rather than the PCM samples itself.
- The differences are encoded as **4-bit** values : hence a **compression** ratio of **4:1** is achieved.
- The term 'Adaptive' is used to describe the quantizer – which varies it's step-size to match the input signal.
- The ADPCM encoder outputs 4-bit samples. The **decoder converts these 4-bit samples to 16-bit signed PCM values**. The process is a lossy – due to quantisation – however more efficient than direct PCM or DPCM.
- The encoder and decoder uses a set of lookup tables used in quantisation, provided by the IMA. Figure 4.1 shows the exponential step sizes used. Speech signals contain more energy at low frequencies than high frequencies – hence the appropriate curve shown in the graph.

Predefined Quantisation Table

```
QuantizerLUT[89] = {
   7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
   19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
   50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
   130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
   337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
   876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
   2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
   5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
   15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767
};

TableofIndexChanges[16] = {
   -1, -1, -1, -1, 2, 4, 6, 8,
   -1, -1, -1, -1, 2, 4, 6, 8,
};
```

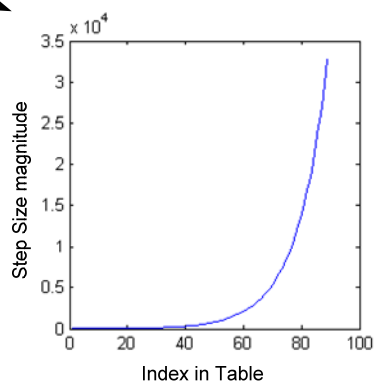


Figure 4.1 - Look up tables used by the Quantizer, exponential step size change

- The only computationally expensive section in the ADPCM decoder is the multiplication of the difference(delta) and the step size value of the inverse quantizer – however these are replaced by shift operations. A block diagram of the encoder and decoder were given in the Design Document (Section 6.3.11.2) and is also shown in Figure 4.2

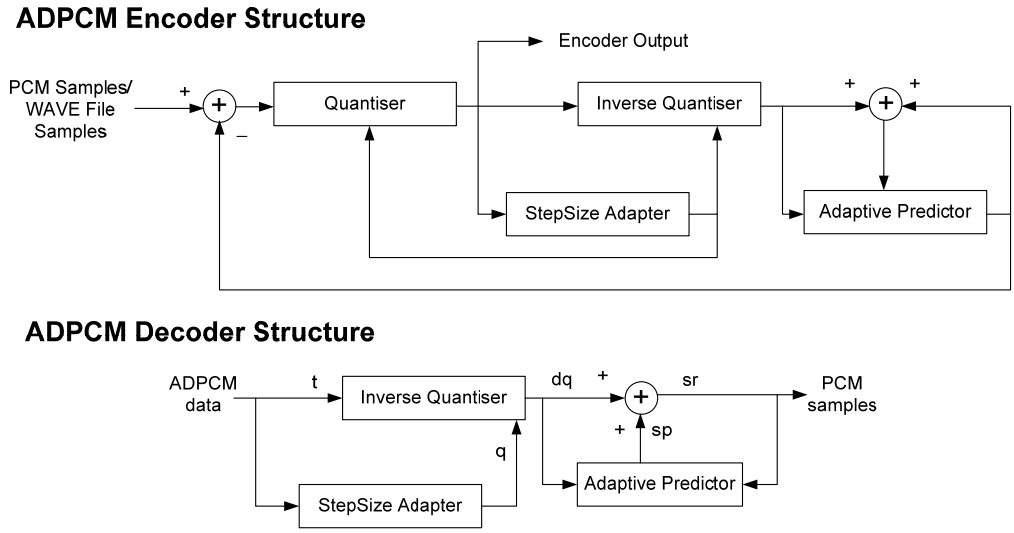


Figure 4.2 - ADPCM Encoder/Decoder structure

4.2 Implementation

The algorithm implementation for the ADPCM encoder and decoder has been given in [16] and implemented in C by J.Jansen [17]. This was the implementation adapted for the project. Certain custom additions needed to be made to facilitate integration into the final system. The '**adpcm.c**' and '**adpcm.h**' files were written to separate the functionality of the decoder from the main application. The following functions are included in the ADPCM decoder (.c) files:

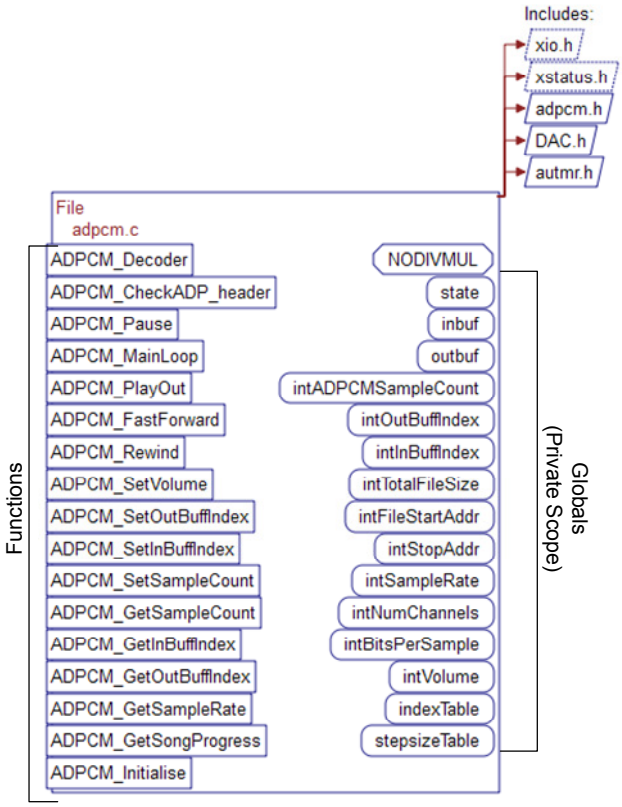


Figure 4.3 - ADPCM.c function declarations

Function Name	Description
ADPCM_Decoder	Able to decode a **block**(fixed size) of 4-bit ADPCM samples
Getters/Setters	To receive or change the values of important global variables used in the decoder
ADPCM_CheckHeader	ADPCM Header checker was added (to both the Encoder and Decoder), the header will contain the following: **TAG <'ADPCM'>** (5 Bytes), **NUM_CHANNELS** (1 Byte), **SAMPLE_RATE** (4 Bytes), **BITS_PER_SAMPLE**(2 Bytes). The main application can now set the sample-rate dynamically. The 'number of channels' field will in future support multi-channel ADPCM decoding.
ADPCM_MainLoop	This function was also written to facilitate integration – the function obtains a block of samples from memory to a temporary buffer and feeds the buffer to the ADPCM_Decoder function. This is done iteratively until all the samples of the file is processed.
ADPCM_PlayOut	The outbuffer contains the decoded PCM samples. This function sends each decoded 16-bit sample to the DAC.
ADPCM_Initialise	Initialises all global variables appropriates. The main application needs to tell the decoder where the data is stored and the file size.
ADPCM_FastForward	Increment input buffer pointer
ADPCM_Rewind	Decrement input buffer pointer
ADPCM_GetSongProgress	Calculate the song play out progress as a percentage.

Table 4.1 - ADPCM.c function explanations

4.2.1 ADPCM Encoding GUI tool

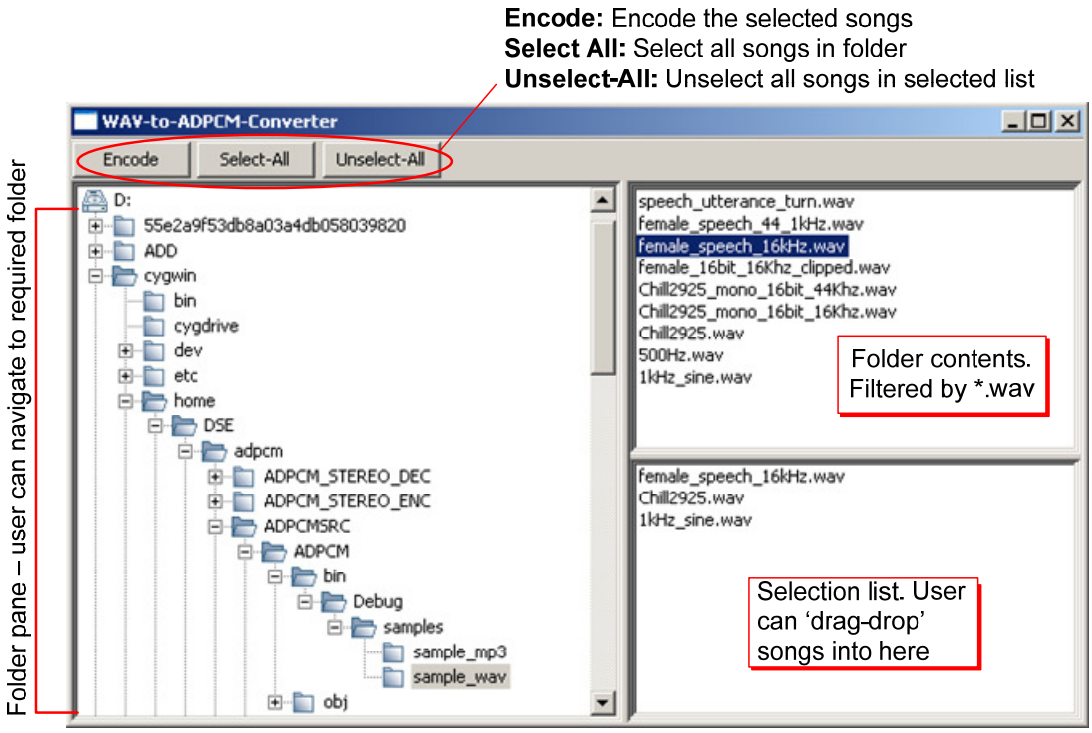


Figure 4.4 - GUI tool implemented to encode .wav files into ADPCM (.adp)

The ADPCM 'encoder' implemented in [17] was used to encode the 16-bit .WAV files into ADPCM format (**.adp** extension). However the program included in [17] was a command line tool and was not

user friendly. The only modification to the encoder that was made was to include the ADPCM – Headers. A GUI tool (Figure 4.4) was developed as a wrapper around the command line tool (coded in *Python, wxPython*) to help the encoding process, as many .WAV files of varying quality needed to be tested on the final system, and process needed to be simple and fast. Also it is a good tool to bundle with the Music player if commercialised.

4.3 Testing and results

4.3.1 Testing on the development environment

The ADPCM encoder and decoder were first tested on the PC – windows environment. The objective of the tests were to find out if:

(a) Did the encoder produce a 4:1 ratio compression ?
(b) Does the modified encoder apply the header correctly before encoded data ?
(c) Can the decoder successfully decode the ADPCM file to correct 16-bit samples ? (qualitatively measure – listen for any noise introduced, loss of quality etc)

Different .WAV files were used to perform the tests:

Varied **sampling frequency** : 8KHz, 16KHz, 24KHz and 44KHz.
Varied song **duration** : 1 sec, 20 sec, 1 min, 4 mins.
All tests carried out on the PC passed successfully. Sound quality degradation was negligible.

4.3.2 Testing on the MicroBlaze platform – Spartan 3E

The next step was to test the ADPCM decoder on the MicroBlaze platform. The flow of this test was exactly similar to the flow illustrated in Figure 2.3 in chapter 2 (Libmad tests).

- All tests were successful – except when trying to decode 32Khz and above sound files – gaps in the music were heard – which verified the processor could not cope with a large sampling rate (the time taken to decode is too much)
- One other strange behaviour observed was when the samples were multiplied with a ‘volume’ factor (0 → 1, fraction). It was clear that the sound volume was low when low values (0.2, 0.3, 0.4, 0.5 etc..) were used, but when large values e.g.: 0.75 were tested, there was a ‘scratching’ noise. Which meant the audio PCM samples were being distorted or voltage level was too high for the DAC. Hence the upper limit of the volume factor was kept at 0.5.

Once these tests were passed integration to the final system was carried out.

4.4 ADPCM integration with other system components

Integration of the ADPCM decoder was the final integration step that was performed – after all the other components and the .WAV decoding had been integrated and tested. Since separate ADPCM related functions had been written in the adpcm.c file – the Main application now only had to call these functions as necessary. An example flow of the final system is given in Figure 4.5 (emphasis on the decoder elements), it shows how the main application now has to decide on which decoder's functions to call based on the file being chosen and played. If more decoders (e.g.: MP3 decoder) were to be added to the final system, the conditional statements would grow – but each decoder would have it's own set of interface functions which would help them to easily 'hook' into the main system.

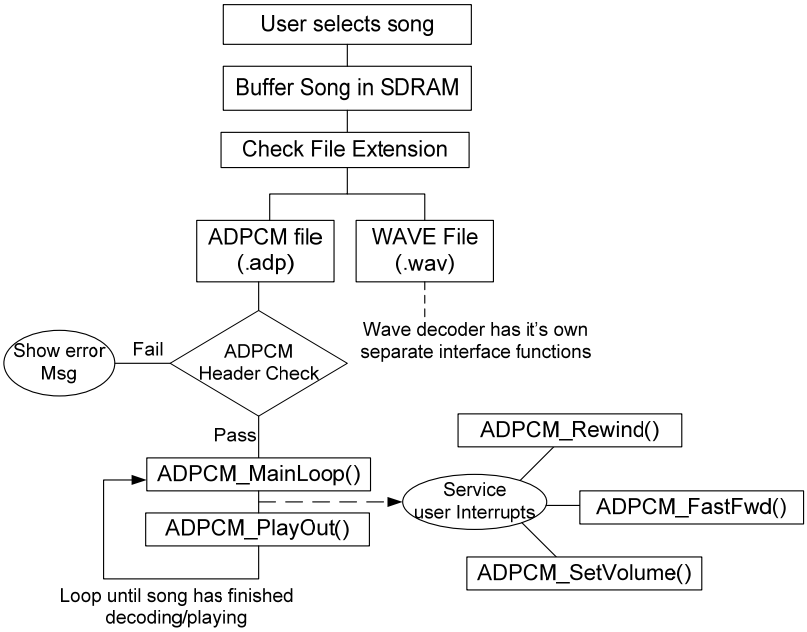


Figure 4.5 - Flow of the final system after ADPCM integration

4.4.1 Issues faced during integrating ADPCM with complete system

The main issue faced when integrating the ADPCM decoder was **running out of local memory** (**BRAM 32KB**, but **program size was over 37KB**). Hence effort had to be put to try to increase the amount of BRAM. The FPGA had a total of 63 KB BRAMs – however the tools would only allow allocation in powers of 2 (e.g.: 8KB, 16KB, 32KB, 64KB etc). Hence some research had to be done to figure out how to 'add' an extra 8KB or 16KB of BRAM.

In the IP Catalog of EDK – there was a core to add an additional BRAM and a BRAM controller. Connecting the new core was straightforward – as it did not have many connection ports. The only adjustment that had to be made was to set an appropriate 'size' (in the address map), for some reason the tools did not allow increasing the size to 16KB (neither did it allow adding 2 additional BRAM

cores/controllers or size 8KB). Figure 4.6 shows the final BRAM controller size settings used for the system (**Additional 8KB – xps_bram_if_cntrl,** was added to the project settings)

The next step was to change the settings in the linker script to use the new BRAM controller. The system did not have contiguous portions of BRAM. Hence the complication was to find out which parts of the program header should to to which BRAM. The final allocation of headers given in the linker script in Figure 4.7 shows the **.rodata** (read-only variables) and **.data**(static and global variables with initial values) were placed in the new BRAM (xps_bram_if_cntrl_0). These two were the second and third largest sections (.text – was the largest section, means large program *code* not data)

It was found out from the EDK error messages – that certain section headers needed to be placed in contigous sections of the memory and cannot be separated.

Bus Interfaces | Ports | Addresses

Instance	Name	Base Address	High Address	Size	Bus Interface(s)	Bus Connection	ICache	DCache	IP Type
debouncer_0	C_BASEADDR	0xc7a00000	0xc7a0ffff	64K	SPLB	mb_plb			debouncer
encoder_0	C_BASEADDR	0xcd800000	0xcd80ffff	64K	SPLB	mb_plb			encoder
dlmb_cntlr	C_BASEADDR	0x00000000	0x00007fff	32K	SLMB	dlmb			lmb_bram_if_cntlr
ilmb_cntlr	C_BASEADDR	0x00000000	0x00007fff	32K	SLMB	ilmb			lmb_bram_if_cntlr
mb_plb	C_BASEADDR			U	Not Applicable				plb_v46
xps_bram_if_cntlr_0	C_BASEADDR	0x85008000	0x85009fff	8K	SPLB	mb_plb	☐	☐	xps_bram_if_cntlr
xps_gpio_LCD_setup	C_BASEADDR	0x81400000	0x8140ffff	64K	SPLB	mb_plb			xps_gpio
xps_intc_0	C_BASEADDR	0x81800000	0x8180ffff	64K	SPLB	mb_plb			xps_intc
xps_spi_1	C_BASEADDR	0x83400000	0x8340ffff	64K	SPLB	mb_plb			xps_spi
xps_spi_0	C_BASEADDR	0x83420000	0x8342ffff	64K	SPLB	mb_plb			xps_spi
xps_spi_2	C_BASEADDR	0x83440000	0x8344ffff	64K	SPLB	mb_plb			xps_spi
xps_timer_0	C_BASEADDR	0x83c00000	0x83c0ffff	64K	SPLB	mb_plb			xps_timer
RS232	C_BASEADDR	0x84000000	0x8400ffff	64K	SPLB	mb_plb			xps_uartlite
mpmc_0	C_MPMC_BASEADDR	0x86000000	0x87ffffff	32M	SPLB0	mb_plb	☐	☐	mpmc

Figure 4.6 - Additional BRAM in the EDK project, setting size to 8KB

.rodata and **.data** placed in 2nd BRAM. These were the second and third largest sections. .text is too large to fit into 8KB

Generate Linker Script

Sections View:

Section	Size (bytes)	Memory
.text	0x000064DC	ilmb_cntlr_dlmb_cntlr
.rodata	0x000015C2	xps_bram_if_cntlr_0
.sdata2	0x00000004	ilmb_cntlr_dlmb_cntlr
.sbss2	0x00000000	ilmb_cntlr_dlmb_cntlr
.data	0x000001B8	xps_bram_if_cntlr_0
.sdata	0x00000000	ilmb_cntlr_dlmb_cntlr
.sbss	0x00000000	ilmb_cntlr_dlmb_cntlr
.bss	0x000006C0	ilmb_cntlr_dlmb_cntlr

Add Section | Delete Section

Boot and Vector Sections:

Section	Address	Memory
.vectors.reset	0x00000000	ilmb_cntlr_dlmb_cntl
.vectors.sw_exception	0x00000008	ilmb_cntlr_dlmb_cntl
.vectors.interrupt	0x00000010	ilmb_cntlr_dlmb_cntl
.vectors.hw_exception	0x00000020	ilmb_cntlr_dlmb_cntl

Heap and Stack View:

Section	Size (bytes)	Memory
Heap	0x0	ilmb_cntlr_dlmb_cntlr
Stack	0x400	ilmb_cntlr_dlmb_cntlr

Memories View:

Memory	Start Address	Length
ilmb_cntlr_dlmb_cntlr	0x00000000	32K
xps_bram_if_cntlr_0	0x85008000	8K
mpmc_0_C_MPMC_BASEADDR	0x86000000	32768K

ELF file used to populate section information:

D:\Xtemp\JF_COMPLETE_v6\TestApp\executable.elf

Output Linker Script: emp\JF_COMPLETE_v6\TestApp\TestApp_linker_script.ld ...

OK | Cancel | Help

Figure 4.7 - Linker script configured to place certain section headers(.rodata and .data) into the new BRAM

Certain sections like the .sdata and .sbss needed to be contiguous – hence the selection of which section should go to which BRAM was not arbitrary, but had to be spread out logically.

4.4.2 ADPCM system integration testing results

The following tests shown in Table 4.2 were carried out during the integration phase- when integrating the ADPCM decoder (integration was done in stages – not at once, helped in debugging).

	Test description	Result
#1	Tested decoding of a Sine wave – BRAM storage (8Khz sampling freq.)	Pass
#2	Tested decoding of a Speech sample – BRAM storage (8Khz sampling freq)	Pass
#3	Read in ADPCM (.adp) file from SD Card – view file data via RS232	Pass
#4	Read in ADPCM file from SDCard, buffer whole file in SDRAM. Check time taken to buffer	Buffering time 4 times lower than buffering a WAV file. This was a good improvement.
#5	Read in ADPCM file (speech sample) from SD Card – buffer into SDRAM – once buffering finished, start decoding in blocks of 1000 bytes, output to DAC and check audio.	Pass, however the block size needed to be reduced – taking up too much BRAM space
#6	Reduced block size to 50 bytes (on both ADPCM encoder/decoder), repeat test #5	Pass, BRAM usage reasonable
#7	Repeated test #6 with longer music file – 16Khz sample rate	Pass.
#8	Repeated test #7 , but now tested Pause/Play/Rewind/Fast Forward functionality	All tests Passed.
#9	Repeated test #7 but now added volume control functionality	For some reason the ADPCM decoder would only produce good quality audio for a volume range of 0→0.5. Anything over 0.5 would produce screeching noises.

Table 4.2 - ADPCM Integration testing results

4.5 Learning experience

- Understanding how linear/non-linear/adaptive quantization works in an audio context. How basic audio compression algorithms are implemented.
- Gained experience how to write a module in C, and integrate it to another code base (linking adpcm.c/.h into final system). Usefulness of interface functions.
- Using timers to output data to the DAC at the correct frequency.
- Communication with other team members to ensure correct integration. Testing and Integration process.
- Adding extra BRAM to a system – maximum amount of BRAM that can be used.
- Understanding how a program is partitioned into different sections (section headers). The meaning of each header.
- GUI programming using Python/wxPython.

5 Quality Assurance and Project management

Teamwork, task management, being organised and communication was key to completing this project much more than our technical competencies. The QA manual outlines several quality assurance metrics – out of those in a general sense the following principles were followed:

5.1 Quality assurance principles

Reporting

- Regular updates were given to every group member – specially the group leader and the project supervisor.(weekly updates to supervisor, daily updates to group leader).
- Any difficulties/accomplishments were reported to the group leader and design/integration manager.

Bug reporting

- Any minor bugs were fixed immediately before adding a new feature.
- Any major bugs were discussed with the group members.

Design changes

- Limitations of the MP3 decoder were explained to the group members. A group decision was then made to focus effort on the ADPCM decoder.

Feedback to/from group members

- Code appraisal: The code other group members developed were checked regularly and any optimisations/bugs were pointed out.
- Code change review: changes to any code was first discussed with other group members.

5.2 Responsibilities as a Software and Documentation Manager

- Review team-members code regularly. Provide feedback. Help resolve major bugs.
- Provide status update to Group Leader.
- Maintain source code versions/backups.
- Outline how the code should be structured. Adhere to the coding standards (e.g.: commenting styles) mentioned in the QA Manual.

- Remind team-members of necessary documentation. Highlight any weaknesses/points missed out in the documentation.

5.3 Version control

Due to the number of tasks and team-members involved in coding, version control was critical – in debugging and rolling back code. Initially the plan was to use a version control system. However since a majority of the team members were not familiar with version control systems this plan was not executed. Version control was carried out by use of folder structures (which was not a good way of managing code, but simple and easy to execute). However because a proper central repository was not present – many hours were wasted tracking a particular version of a code branch, also rolling back to a particular state was not easily possible. Figure 5.1 shows an example of how source code versioning was carried out.

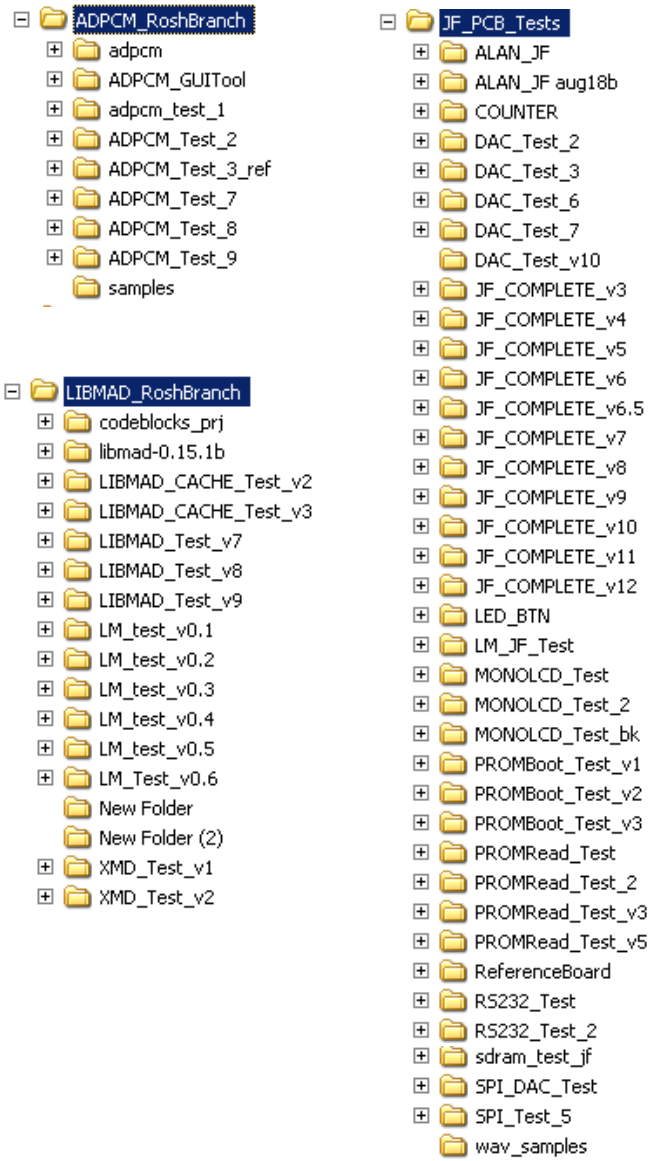


Figure 5.1 - Crude version control method (managing via folders)

5.4 Integration of different components

Figure 5.2 shows different strands of project work being integrated together. As planned in the design document (Tier-3: integration and testing) integration was done in different steps and each integration branch was tested before further integration was done. At first glance it is clear that integration was one of the main tasks in the project. It is a very difficult process to streamline and many bugs were discovered during the different integration stages. Branch (9) in Figure 5.2 is the final main release branch of the code – and all final updates were merged into this.

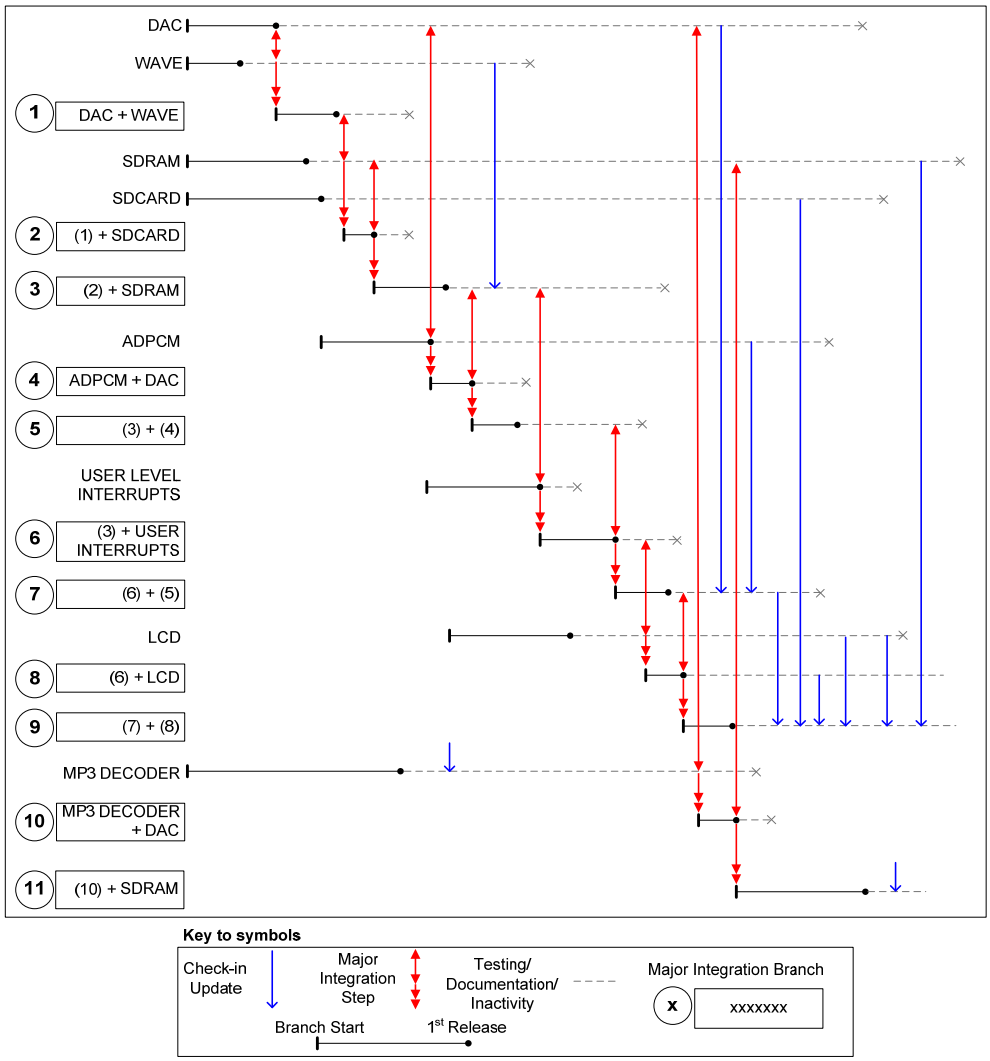
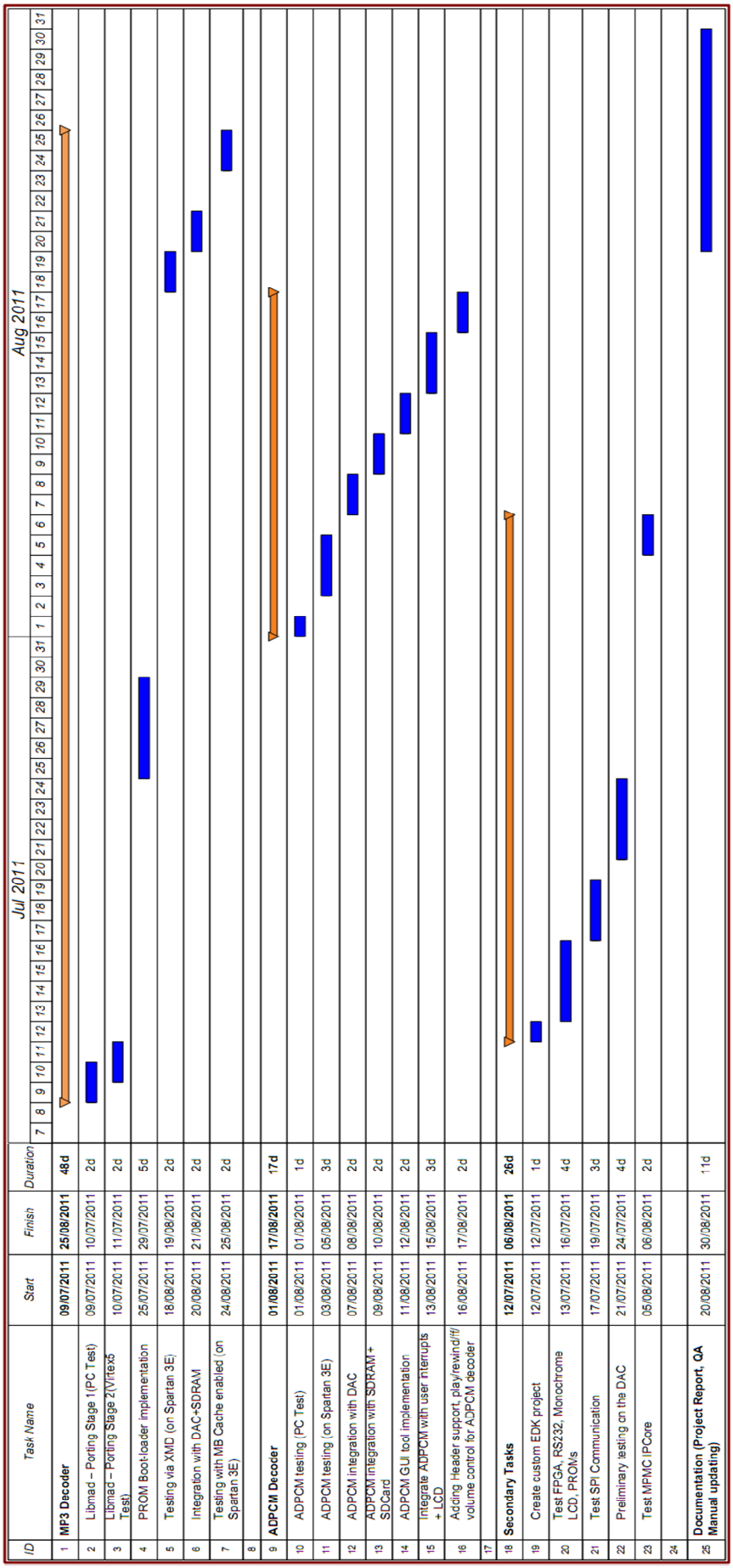


Figure 5.2 - Integration steps. Different branches of code merging together. Application of updates

After every integration step – progress and status was reported to the group leader and a new version of the code was backed-up. Any individual component updates were first tested independently before merging the update with the main branch

5.5 Time Management

Since there were many tasks to attend to – proper time management was crucial. Figure 5.3 shows the effort spent on different areas of the tasks during the project implementation period.



ID	Task Name	Start	Finish	Duration
1	**MP3 Decoder**	**09/07/2011**	**25/08/2011**	**48d**
2	Libmad – Porting Stage 1(PC Test)	09/07/2011	10/07/2011	2d
3	Libmad – Porting Stage 2(Virtex5 Test)	10/07/2011	11/07/2011	2d
4	PROM Bootloader implementation	25/07/2011	29/07/2011	5d
5	Testing via XMD (on Spartan 3E)	18/08/2011	19/08/2011	2d
6	Integration with DAC+SDRAM	20/08/2011	21/08/2011	2d
7	Testing with MB Cache enabled (on Spartan 3E)	24/08/2011	25/08/2011	2d
8				
9	**ADPCM Decoder**	**01/08/2011**	**17/08/2011**	**17d**
10	ADPCM testing (PC Test)	01/08/2011	01/08/2011	1d
11	ADPCM testing (on Spartan 3E)	03/08/2011	05/08/2011	3d
12	ADPCM integration with DAC	07/08/2011	08/08/2011	2d
13	ADPCM integration with SDRAM+ SDCard	09/08/2011	10/08/2011	2d
14	ADPCM GUI tool implementation	11/08/2011	12/08/2011	2d
15	Integrate ADPCM with user interrupts + LCD	13/08/2011	15/08/2011	3d
16	Adding Header support, play/rewind/ff/ volume control for ADPCM decoder	16/08/2011	17/08/2011	2d
17				
18	**Secondary Tasks**	**12/07/2011**	**06/08/2011**	**26d**
19	Create custom EDK project	12/07/2011	12/07/2011	1d
20	Test FPGA, RS232, Monochrome LCD, PROMs	13/07/2011	16/07/2011	4d
21	Test SPI Communication	17/07/2011	19/07/2011	3d
22	Preliminary testing on the DAC	21/07/2011	24/07/2011	4d
23	Test MPMC IPCore	05/08/2011	06/08/2011	2d
24				
25	**Documentation (Project Report, QA Manual updating)**	20/08/2011	30/08/2011	11d

Figure 5.3 - Project Timeline (for the Decoder tasks)

5.6 Resource repository

A majority of the resources were obtained from the internet. A vast community of software and hardware engineers exist and can be reached via forums (e.g.: Xilinx Community Forums, FPGA related forum, EDA Board forum etc). A large number of technical blogs also exist on the internet where scraps of useful information and source code were extracted. So it was important to keep track of all the resources found on a daily basis.

The following web page was set up for the group to store any relevant resources (web links), and to keep track of progress on the project. It was meant to be a method for collaborating more effectively.

https://sites.google.com/site/dsegroupproj2011/research/rosh

5.7 Financial considerations

During the tender presentation and the tender documentation – several financial aspects of the project were considered. Several design changes were made during the implementation stage – these changes will directly affect the cost of the music player.

- The object and primary specification was to implement a MP3 player. The prototype implementation of the device introduced several limitations in hardware resources – hence a MP3 player was not developed on time. This will reduce the cost of the product considerably
    - **Estimated 50% decrease in final price**
- The effort required for implementing the design was underestimated. Each engineer worked at least 10 hours, 6-7 days a week. This increase in labour cost will increase the cost of the product.
    - **Estimated 20% increase in final price**
- The final implementation uses a Colour LCD. This provides a better user experience and hence will increase the cost of the product.
    - **Estimated 10% increase in final price**
- The final design uses SPI transfer for communication with the SD Card. The SD mode communication requires a licensing fee. With SPI communication, the song is buffered to the SDRAM – hence the user has to 'wait' until the buffering is finished to listen to the song. This is a bad user-experience and hence will bring down the cost of the product further.
    - **Estimated 20% reduction in final price for – inadequate user experience.**
    - One-off payment (but very costly) to SD Association for licensing fee for SD Mode communication.

6 Future work

Due to time and resource constraints certain requirements from the 'decoder' section were not met. Bellow are some future improvements and design changes that can be made to the system to successfully accomplish all tasks.

6.1 Primary Task – MP3 decoding (Future work)

From the results obtained in Chapter 3 it was evident that real-time MP3 decoding 'is' possible on the Spartan 3E. However the following further modifications could have been explored if time permitted.

- Since decoding of 1 second MP3 is less than 1 second (latest results – Chapter 3, Table 3.1), the output can be directly sent to the DAC without post-decoder buffering into the SDRAM. If this passes, then an end to end test of MP3 can be performed – the SDRAM will only be used to store pre-decoded music, copied from SDCard. After this stage the LCD/SD Card/User interrupts can be added. The main application will then be needed to be updated so it can handle the new file type.
- The application will need to undergo regression tests to ensure all user-level interrupts are being services correctly.
- MicroBlaze uses software multiplication by default. Hardware multiplication can be enabled by a compiler option *"-mno-xl-soft-mul"* [19]. This would give some improvement to the decoding process.
- The current clock speed = 66MHz, a clock multiplier can be used to test of increasing the clock speed would result in any significant performance increase. Research into Xilinx clock managers (DCM IPCore) is needed.
- Bootloader issue : the work carried out on the bootloader is incomplete. Further investigation can be carried out on the SDRAM data contents 'after' coping the program from PROM to SDRAM. Check for 'byte-swapping' within the SDRAM. XMD can be used to investigate the boot-loader issue further by observing what happens 'after' the jump to the SDRAM base address. Once the bootloader is fully functional and tested – the device would be portable.
- Explore the option of using a third-party 'free' (from OpenCores.com) VHDL MP3 Decoder. The complexity would be to configure the IP core correctly and then 'integrate' the core into the overall system. If this option is explored – the Xilinx Fast Simplex Link (FSL) will need to be researched – to be able to incorporate custom IP. An example of a custom IDCT hardware accelerator is given in Xilinx XAPP529.

- Alternatively – another audio compression algorithm (less expensive) can be explored.

6.1.1 Hardware/Board level changes

From all the tests and analysis carried out- the ideal resources for an MP3 player is given below:

BRAM	Clock Speed	SD Card	LCD	DAC
250KB and above LIBMAD=205KB Others = 38KB (e.g.: Spartan 3A-DSP)	100Mhz or above	SD mode preferred (but costly!)	Monochrome LCD, for faster, real-time status updating.	24-bit for more precision

Table 6.1 - Recommended Hardware requirements for a MP3 Music Player

6.2 Secondary Task – ADPCM decoding (Future work)

- The ADPCM decoder is fully functional. The only issues existing are related to volume control (as mentioned in Section 4.4.2). The following steps could have been taken to troubleshoot (if time permitted):
  - Using a pure tone (e.g.: sine wave converted to a ADPCM file) observe the output of the DAC on an oscilloscope. Observe any distortions. Maybe the voltage goes too high ?
  - Check the output of the samples with volume set to 0.75.
- Update the ADPCM Encoder GUI tool.
  - Add options to decoder ADPCM to wave.
  - Add option to save previously used directory
  - Add ‘player’ functionality – able to play the songs within the tool after conversion.

7 Conclusion

The goal of the project was to investigate and implement an audio decoding algorithm on an FPGA. The decoder was a central sub-system to a larger embedded system – a Music Player. For these purposes, the MPEG-Layer 3 (MP3) algorithm was chosen and the implementation environment centred around the Xilinx Spartan 3E 1200E FPGA.

- **Ported the Libmad MP3 decoder software library to MicroBlaze**
  The Libmad decoder library was chosen as the software based MP3 decoder of choice. First certain features of it were reduced to decrease it's program size (Chapter 2). The Libmad MP3 decoder library was then successfully ported to the MicroBlaze platform and tested on the Xilinx Virtex 5 ML505 XUPv5 development board. A test mechanism was setup to further evaluate the performance of the library on the target platform – a Spartan 3E 1200E FPGA.

- **Tested the Libmad MP3 decoder on the target platform (Xilinx Spartan 3E 1200E)**
  Effort was put into implementing a bootloader to be able to load the Libmad library on the Spartan 3E FPGA which had a BRAM limitation. As this solution was unsuccessful – the library was tested on the target platform using *XMD*. Several tests were run on Libmad on the target platform, and the best achieved results were obtained by enabling 16KB of instruction-cache and 8KB of data-cache on the MicroBlaze system. The processor was then able to decode 1 second of MP3 under a second (0.9475 seconds). Which indicates with further optimisations – as mentioned in Chapter 6, it 'maybe' possible to decode MP3 in real-time on the target platform.

- **Implemented the backup decoder – ADPCM on the target platform**
  As a backup an ADPCM decoder was implemented, tested and integrated successfully with the final overall system. The main application now supports raw WAVE (uncompressed) audio file formats as well as compressed audio in the form of ADPCM. However still certain minor issues remain with the volume of the PCM samples produced – and is described in Section 6.2.

Lastly both the knowledge and experience gained during this project have been both exiting, valuable and extensible to many other fields, beside audio decoding applications.

8 References

[1] "Underbit Technologies - MAD Decoder library Web site", http://www.underbit.com/products/mad/, accessed 01/06/2011

[2] "memcpy and memmove implementations", http://en.wikibooks.org/wiki/C_Programming/Strings, accessed 13/07/2011

[3] "malloc, free" Xilinx implementation, http://www.xilinx.com/support/answers/23345.htm, accessed 11/07/2011

[4] "LibXil Standard C Libraries Document", EDK 9.1i December 12 2006, Xilinx Inc.

[5] "Virtex-5 Family Overview - DS100 v5.0", Xilinx inc. February 6, 2009.

[6] Bin2h - Binary file to C style Header converter, http://www.deadnode.org/sw/bin2h/, accessed 08/07/2011

[7] GoldWave Audio player, http://www.goldwave.com, accessed 10/07/2011

[8] S. Sheth "XAPP482 - MicroBlaze Platform Flash/PROM Boot Loader and Data Storage", Xilinx Inc, June 27th 2005.

[9] A. Khu, F. Shokoui, J. Hussein, A. Patel, "XAPP544 - Using Xilinx XCF02S/XCF04S JTAG PROMs for Data Storage Applications", Xilinx Inc, January 11th 2008.

[10] "XAPP 694 - Reading User Data from Configuration PROMs", Xilinx Inc, Nov. 19th 2007.

[11] "PROMGen - Description of PROM/EEPROM file formats: MCS, EXO, HEX and others", http://www.xilinx.com/support/answers/476.htm

[12] C. Agurto, XILINX MICROPROCESSOR DEBUGER (XMD) REFERENCE GUIDE, www.cosmiac.org/pdfs/T1_XMD_commands_REFERENCE_V.2.doc, accessed 25/07/2011

[13] "Loading code from DDR SDRAM", http://forums.xilinx.com/t5/Spartan-Family-FPGAs/Executing-MicroBlaze-code-from-the-ddr-sdram-on-Spartan-3e/td-p/5147, accessed 28/07/2011

[14] "EDK Profiling User Guide, UG448-EDK 11.2", Xilinx Inc. 2009.

[15] "ADPCM Decoding" http://wiki.multimedia.cx/index.php?title=IMA_ADPCM, accessed 01/08/2011

[16] "Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia Systems", IMA Digital Audio Focus and Technical Working Groups, 21 October 1992.

[17] J. Jansen, "IMA/Intel-DVI ADPCM Implementation" , Centre for Mathematics and Computer Science, Stichting Mathematisch Centrum, Amsterdam, The Netherlands, http://www.cwi.nl/ftp/audio/adpcm.zip, accessed 02/08/2011

[18] A. Tyrrel - "Embedded Computer Systems - Microprocessor Architectures, lecture notes", University of York, 2010/2011

[19] "OS and Libraries Document Collection, UG643" - Xilinx Inc, March 1, 2011.

9 Appendix

This section will contain the following items:

- Libmad test on Virtex-5 – mb-objdump, mb-size output
- Libmad test on Spartan 3E – XMD session.
- Libmad mb-gprof (code profiling) results.
- Libmad testing – rs232 screen capture example
- ADPCM testing – rs232 screen capture example
- Main Test Application for testing Libmad on Spartan 3E (with caching enabled)
- PROM Bootloader Test Application
- ADPCM decoder C-modules (adpcm.c, adpcm.h)
- ADPCM encoder wrapper (GUI tool – coded in wxPython)

Please Note: Full listing of other source material can be found on the CD [folder : ./ROSH_MENDIS/]

9.1 LIBMAD Test - on Virtex 5 (mb-objdump, mb-size)

```
executable.elf:     file format elf32-microblaze
Sections:
Idx Name                          Size      VMA       LMA       File off  Algn
  0 .vectors.reset                00000004  00000000  00000000  000000b4  2**2
  1 .vectors.sw_exception         00000008  00000008  00000008  000000b8  2**2
  2 .vectors.interrupt            00000008  00000010  00000010  000000c0  2**2
  3 .vectors.hw_exception         00000008  00000020  00000020  000000c8  2**2
  4 .text                         0000db08  00000050  00000050  000000d0  2**2
  5 .init                         00000028  0000db58  0000db58  0000dbd8  2**2
  6 .fini                         00000020  0000db80  0000db80  0000dc00  2**2
  7 .rodata                       0000ae32  0000dba0  0000dba0  0000dc20  2**2
  8 .sbss2                        00000000  000189d2  000189d2  00019d1c  2**0
  9 .data                         000012b0  000189d4  000189d4  00018a54  2**2
 10 .ctors                        00000008  00019c84  00019c84  00019d04  2**2
 11 .dtors                        00000008  00019c8c  00019c8c  00019d0c  2**2
 12 .eh_frame                     00000004  00019c94  00019c94  00019d14  2**2
 13 .jcr                          00000004  00019c98  00019c98  00019d18  2**2
 14 .sbss                         00000004  00019c9c  00019c9c  00019d1c  2**0
 15 .tdata                        00000000  00019ca0  00019ca0  00019d1c  2**0
 16 .tbss                         00000000  00019ca0  00019ca0  00019d1c  2**0
 17 .bss                          00000078  00019ca0  00019ca0  00019d1c  2**2
 18 .heap                         00019000  00019d18  00019d18  00019d1c  2**0
 19 .stack                        00000400  00032d18  00032d18  00019d1c  2**0
 20 .debug_abbrev                 00001ced  00000000  00000000  00019d1c  2**0
 21 .debug_info                   00008164  00000000  00000000  0001ba09  2**0
 22 .debug_line                   0000a9b8  00000000  00000000  00023b6d  2**0
 23 .debug_frame                  00000958  00000000  00000000  0002e528  2**2
 24 .debug_loc                    000080e5  00000000  00000000  0002ee80  2**0
 25 .debug_pubnames               00000767  00000000  00000000  00036f65  2**0
 26 .debug_aranges                000002e0  00000000  00000000  000376d0  2**3
 27 .debug_ranges                 00000b48  00000000  00000000  000379b0  2**0
 28 .debug_str                    000020ba  00000000  00000000  000384f8  2**0

mb-size TestApp/executable.elf
   text        data         bss         dec         hex     filename
 100766        4808      103548      209122      330e2     TestApp/executable.elf
```

.text	Text Section
.rodata	Read-Only Data Section
.sdata2	Small Read-Only Data Section
.sbss2	Small Read-Only Uninitialized Data Section
.data	Read-Write Data Section
.sdata	Small Read-Write Data Section
.sbss	Small Uninitialized Data Section
.bss	Uninitialized Data Section
.heap	Program Heap Memory Section
.stack	Program Stack Memory Section

9.2 LIBMAD Test on Spartan 3E – XMD session

```
Installing Cygwin from EDK installation area...
Added registry entries for c:\Xilinx\10.1\edk\cygwin
Xilinx Microprocessor Debug (XMD) Engine
Xilinx EDK 10.1.03 Build EDK_K_SP3.6
Copyright (c) 1995-2007 Xilinx, Inc.  All rights reserved.

Overriding IP level properties ...
Address Map for Processor microblaze_0
  (0000000000-0x00007fff) dlmb_cntlr    dlmb
  (0000000000-0x00007fff) ilmb_cntlr    ilmb
  (0x81800000-0x8180ffff) xps_intc_0    mb_plb
  (0x83400000-0x8340ffff) xps_spi_1     mb_plb
  (0x83420000-0x8342ffff) xps_spi_0     mb_plb
  (0x83c00000-0x83c0ffff) xps_timer_0   mb_plb
  (0x83c20000-0x83c2ffff) timer_1       mb_plb
  (0x84000000-0x8400ffff) RS232 mb_plb
  (0x84400000-0x8440ffff) debug_module  mb_plb
  (0x86000000-0x87ffffff) mpmc_0        mb_plb

XMD%
Info:AutoDetecting cable. Please wait.
Info:Connecting to cable (Parallel Port - LPT1).
Info:Checking cable driver.
Info: Driver windrvr6.sys version = 8.1.1.0.Info: WinDriver v8.11 Jungo (c) 1997
 - 2006 Build Date: Oct 16 2006 X86 32bit SYS 12:35:07, version = 811.
Info:Cable connection failed.
Info:Connecting to cable (Parallel Port - LPT2).
Info:Checking cable driver.
Info: Driver windrvr6.sys version = 8.1.1.0.Info: WinDriver v8.11 Jungo (c) 1997
 - 2006 Build Date: Oct 16 2006 X86 32bit SYS 12:35:07, version = 811.
Info:Cable connection failed.
Info:Connecting to cable (Parallel Port - LPT3).
Info:Checking cable driver.
Info: Driver windrvr6.sys version = 8.1.1.0.Info: WinDriver v8.11 Jungo (c) 1997
 - 2006 Build Date: Oct 16 2006 X86 32bit SYS 12:35:07, version = 811.
Info:Cable connection failed.
Info:Connecting to cable (Parallel Port - LPT4).
Info:Checking cable driver.
Info: Driver windrvr6.sys version = 8.1.1.0.Info: WinDriver v8.11 Jungo (c) 1997
 - 2006 Build Date: Oct 16 2006 X86 32bit SYS 12:35:07, version = 811.
Info:Cable connection failed.
Info:Connecting to cable (Usb Port - USB21).
Info:Checking cable driver.
Info: Driver file xusb_emb.sys found.
Info: Driver version: src=1029, dest=1029.
Info: Driver windrvr6.sys version = 8.1.1.0.Info: WinDriver v8.11 Jungo (c) 1997
 - 2006 Build Date: Oct 16 2006 X86 32bit SYS 12:35:07, version = 811.
Info: Cable PID = 0008.
Info: Max current requested during enumeration is 74 mA.
Info:Type = 0x0004.
Info: Cable Type = 3, Revision = 0.
Info: Setting cable speed to 6 MHz.
Info:Cable connection established.
Info:Firmware version = 1303.
Info:File version of c:/Xilinx/10.1/ise/data/xusb_xlp.hex = 1303.
Info:Firmware hex file version = 1303.
Info:PLD file version = 0012h.
Info: PLD version = 0012h.

JTAG chain configuration
```

```
-------------------------------------------------
Device   ID Code        IR Length    Part Name
 1       21c2e093           6        XC3S1200E
 2       f5046093           8        XCF04S
 3       f5046093           8        XCF04S

MicroBlaze Processor Configuration :
-------------------------------------
Version............................7.10.d
Optimization.......................Area
Interconnect.......................PLBv46
MMU Type...........................No_MMU
No of PC Breakpoints...............1
No of Read Addr/Data Watchpoints...0
No of Write Addr/Data Watchpoints..0
Instruction Cache Support..........off
Data Cache Support.................off
Exceptions  Support................off
FPU  Support.......................off
Hard Divider Support...............off
Hard Multiplier Support............on - (Mul32)
Barrel Shifter Support.............off
MSR clr/set Instruction Support....on
Compare Instruction Support........on

Connected to "mb" target. id = 0
Starting GDB server for "mb" target (id = 0) at TCP port no 1236
XMD% cd TestApp
XMD% dow executable.elf
System Reset .... DONE
Downloading Program -- executable.elf
        section, .vectors.reset: 0x00000000-0x00000007
        section, .vectors.sw_exception: 0x00000008-0x0000000f
        section, .vectors.interrupt: 0x00000010-0x00000017
        section, .vectors.hw_exception: 0x00000020-0x00000027
        section, .text: 0x86000000-0x860117a7
        section, .init: 0x860117a8-0x860117cf
        section, .fini: 0x860117d0-0x860117ef
        section, .ctors: 0x860117f0-0x860117f7
        section, .dtors: 0x860117f8-0x860117ff
        section, .rodata: 0x86011800-0x8601c775
        section, .data: 0x8601c778-0x8601da73
        section, .eh_frame: 0x8601da74-0x8601da77
        section, .jcr: 0x8601da78-0x8601da7b
        section, .bss: 0x8601da80-0x8601db57
        section, .heap: 0x8601db58-0x86036b57
        section, .stack: 0x86036b58-0x86038157
Setting PC with Program Start Address 0x00000000

XMD% run
Info:Processor started. Type "stop" to stop processor

RUNNING> XMD%
```

################ Hyper-Terminal Output shown below ##################

```
***** LIBMAD TEST v0.6 ******
****************************
Main::sizeof(speech_sample_24KHz_64kbps = 3456
Main::(void *)speech_sample_24KHz_64kbps = 0x8601C784
Main::going to decode now..
```

```
Main:: Initialise Timer
Decoding Started - Initial Timer value is 1716
==================================================

==================================================
Decoding Finished - Total decoding time is 776639075
File Size in SDRAM:  intSDRAMAddrOffset = 00004C80
AuTmr_StartTimer::enter
Finished Playing whole song

##########################################################################
```

9.3 Results from 'mb-gprof' (LIBMAD profiling on the Spartan 3E)

Note: the following tests were made without caching enabled.

```
XMD% profile -config sampling_freq_hz 10000 binsize 4 profile_mem 0x86F00000

mb-gprof -b TestApp/executable.elf gmon.out

Flat profile:

Each sample counts as 0.0001 seconds.
```

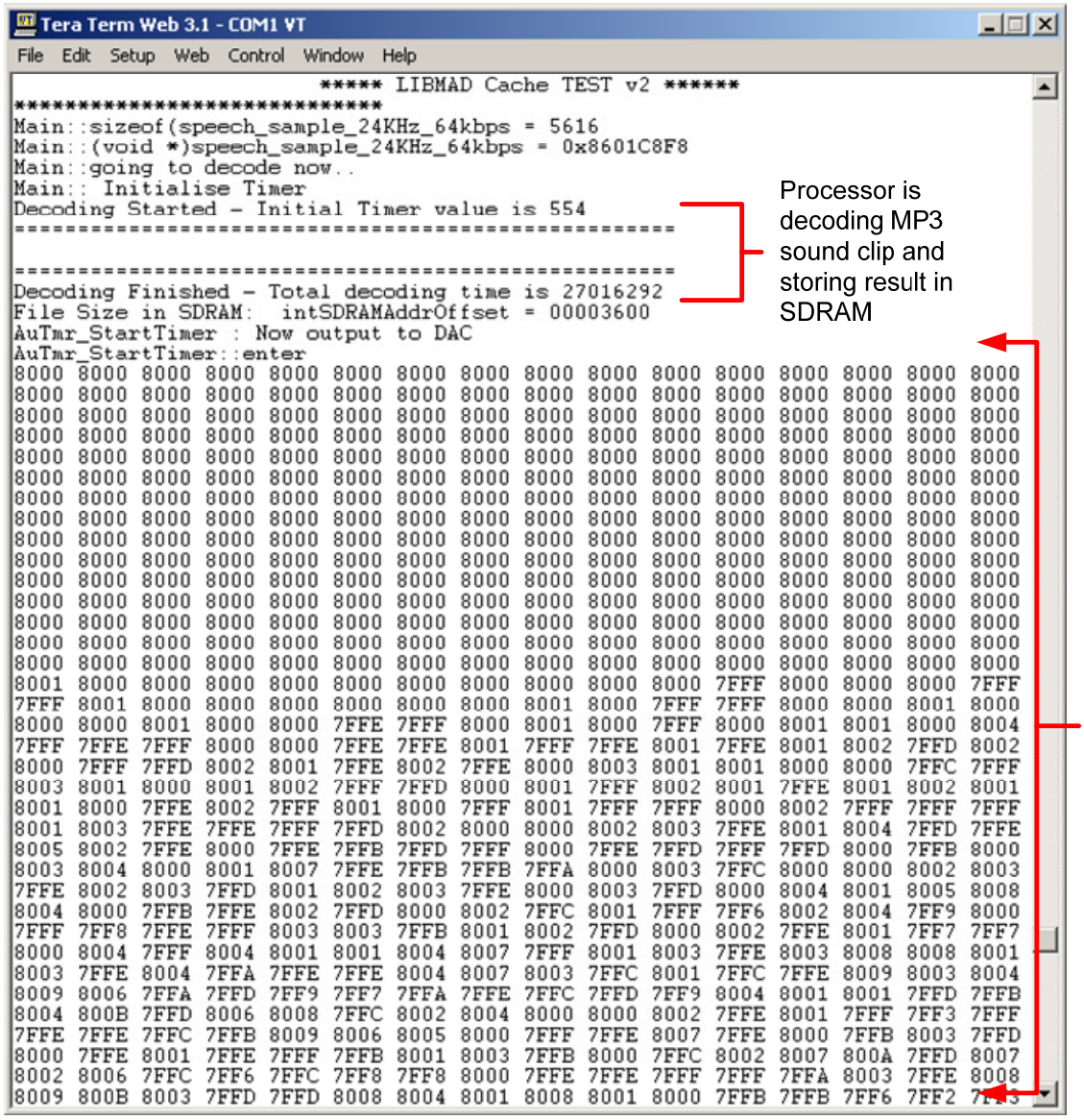
% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
45.43	11.66	11.66				__mulsi3
22.65	17.47	5.81				xil_printf
6.89	19.24	1.77	465	0.00	0.00	III_imdct_l
5.76	20.71	1.48	17	0.09	0.30	mad_layer_III
4.73	21.92	1.21	306	0.00	0.00	dct32
4.04	22.96	1.04	17	0.06	0.13	synth_full
1.71	23.40	0.44	17	0.03	0.03	output
1.64	23.82	0.42	15	0.03	0.03	III_aliasreduce
1.57	24.22	0.40	930	0.00	0.00	fastsdct
1.36	24.57	0.35	61	0.01	0.01	III_imdct_s
1.17	24.87	0.30	3032	0.00	0.00	mad_bit_read
0.74	25.06	0.19	1	0.19	0.19	DAC_Initialise
0.57	25.21	0.15				XUartLite_SendByte
0.56	25.35	0.14	526	0.00	0.00	III_overlap
0.54	25.49	0.14	738	0.00	0.00	III_requantize
0.20	25.54	0.05	26	0.00	0.00	memcpy
0.11	25.57	0.03	272	0.00	0.00	III_freqinver
0.06	25.59	0.02				__udivsi3
0.04	25.60	0.01				__modsi3
0.04	25.61	0.01				__umodsi3
0.03	25.61	0.01	35	0.00	0.00	mad_bit_skip
0.03	25.62	0.01	1	0.01	0.01	mad_frame_mute
0.01	25.63	0.00	9	0.00	0.00	memmove
0.01	25.63	0.00	18	0.00	0.00	decode_header
0.01	25.63	0.00	18	0.00	0.00	mad_header_decode
0.01	25.64	0.00				memset
0.01	25.64	0.00	41	0.00	0.00	mad_bit_nextbyte
0.01	25.64	0.00	1	0.00	0.00	mad_synth_mute
0.01	25.64	0.00	1	0.00	0.00	mad_frame_finish
0.01	25.64	0.00	18	0.00	0.00	mad_timer_set
0.01	25.65	0.00				outbyte
0.01	25.65	0.00	18	0.00	0.28	mad_frame_decode
0.01	25.65	0.00	54	0.00	0.00	mad_bit_init
0.01	25.65	0.00	17	0.00	0.13	mad_synth_frame
0.00	25.65	0.00	17	0.00	0.00	mad_bit_length
0.00	25.65	0.00				outnum
0.00	25.65	0.00				_malloc_r
0.00	25.66	0.00	1	0.00	7.81	run_sync
0.00	25.66	0.00	1	0.00	0.00	mad_synth_init
0.00	25.66	0.00				XSpi_Transfer
0.00	25.66	0.00				_free_r
0.00	25.66	0.00				_sbrk_r
0.00	25.66	0.00	1	0.00	0.00	mad_stream_init
0.00	25.66	0.00				XTmrCtr_GetValue
0.00	25.66	0.00				__divsi3
0.00	25.66	0.00				mad_timer_string
0.00	25.66	0.00				timer_int_handler
0.00	25.66	0.00	2	0.00	0.00	input

```
  0.00     25.66     0.00        1     0.00     0.00  mad_decoder_init
  0.00     25.66     0.00        1     0.00     0.00  mad_header_init
  0.00     25.66     0.00        1     0.00     8.00  main
  0.00     25.66     0.00                             XSpi_Initialize
  0.00     25.66     0.00                             XSpi_Reset
  0.00     25.66     0.00                             XSpi_SetOptions
  0.00     25.66     0.00                             XSpi_Start
  0.00     25.66     0.00                             XTmrCtr_LookupConfig
  0.00     25.66     0.00                             XTmrCtr_SetOptions
  0.00     25.66     0.00                             _profile_init
  0.00     25.66     0.00                             end_len
  0.00     25.66     0.00                             sbrk
  0.00     25.66     0.00        1     0.00     0.00  DAC_PowerDownDAC
  0.00     25.66     0.00                             free
  0.00     25.66     0.00                             XTmrCtr_SetResetValue
  0.00     25.66     0.00                             check_alignment
  0.00     25.66     0.00                             getnum
  0.00     25.66     0.00                             padding
  0.00     25.66     0.00                             synth_half
  0.00     25.66     0.00        1     0.00     0.00  mad_stream_finish
  0.00     25.66     0.00                             DAC_SendDataToLeftChannel
  0.00     25.66     0.00                             len_loop
  0.00     25.66     0.00                             malloc
  0.00     25.66     0.00                             strlen
  0.00     25.66     0.00        1     0.00     0.00  AuTmr_StartTimer
  0.00     25.66     0.00        1     0.00     0.00  mad_decoder_finish
  0.00     25.66     0.00        1     0.00     7.81  mad_decoder_run
  0.00     25.66     0.00        1     0.00     0.01  mad_frame_init
  0.00     25.66     0.00        1     0.00     0.00  mad_stream_buffer
```

Explanation of the fields(taken from the 'gprof' website: http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html#SEC5)

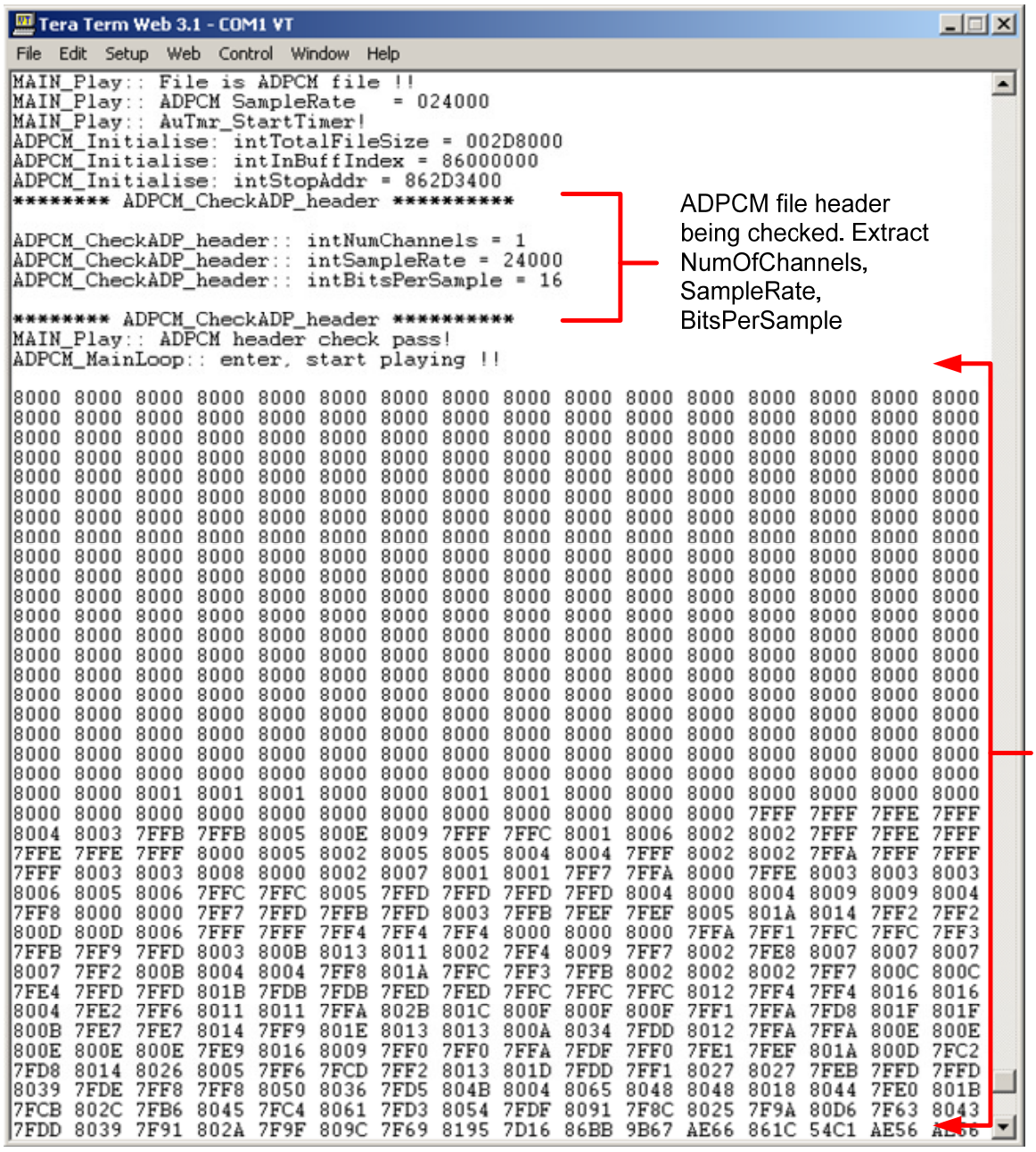
% time	This is the percentage of the total execution time your program spent in this function. These should all add up to 100%.
cumulative seconds	This is the cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.
self seconds	This is the number of seconds accounted for by this function alone. The flat profile listing is sorted first by this number.
calls	This is the total number of times the function was called. If the function was never called, or the number of times it was called cannot be determined (probably because the function was not compiled with profiling enabled), the calls field is blank.
self ms/call	This represents the average number of milliseconds spent in this function per call, if this function is profiled. Otherwise, this field is blank for this function.
total ms/call	This represents the average number of milliseconds spent in this function and its descendants per call, if this function is profiled. Otherwise, this field is blank for this function.
name	This is the name of the function. The flat profile is sorted by this field alphabetically after the self seconds field is sorted.

9.4 Libmad testing – rs232 output screen capture



Deocoded PCM samples were checked against the development version of the decoder running on the PC.

9.5 ADPCM testing – rs232 output screen capture



Deocoded PCM samples were checked against the development version of the decoder running on the PC.

9.6 Main Test application for testing Libmad on MicroBlaze platform

(Caching enabled, and timer to calculate decode latency)

```
/*****************************************************************
 * Name        : Libmad-TestApp (TestApp.c)
 * Modified by : Rosh Mendis
 * Revisions   : v2.0
 * Description : Test application written to test Libmad MP3
 *               Library on the MicroBlaze platform
 *               - Initial creation, mad.h, rs232 output
 *               - Large potion of the code taken from 'minimad.c'
 *               - Added timer to calculate decode delay
 *               - Added SDRAM r/w to store decoded data
 *               - Added DAC to output PCM samples
 *               - Added Cache software support
 *****************************************************************/

/** Xilinx dpecific includes **/
#include "xparameters.h"
#include "xbasic_types.h"
#include "xtmrctr.h"
#include "xspi.h"
#include <xintc_l.h>
#include "xio.h"

/** Local custom includes **/
#include "mad.h"    // api for LIBMAD library
//#include "speech_sample_24KHz_64kbps.h"
//#include "speech_sample_16KHz.h"
#include "chilll_32Kbps_16Khz.h"    // mp3 sample (7 seconds)
#include "COMMON.h"
#include "DAC.h"
#include "autmr.h"
#include "SDRAM.h"

/** Local defines  **/
// performance calculation timer
#define TIMER_COUNTER_1         1
// sdram offset of decoded data storage
#define SDRAM_DATA_START_LOC    (MEM_BASEADDR + 0x00100000)
// sample rate of mp3 file
#define SAMPLE_RATE 16000

/** Global variables  **/
XTmrCtr XPS_Timer0;
Xuint32 intDevStatus;
XSpi mySPI;
Xuint32 intSDRAMAddrOffset=0;
Xuint32 intDataRemaining=0;

/** Global function prototypes  **/
static int decode(unsigned char const *, unsigned long);



/*****************************************************************
 * @Name    : pause
 * @Desc    : Stalls processor in a loop
 * @Param   : None
 * @Return  : None
 *****************************************************************/
void pause(void){
    Xuint32 count;
    count=0x0000FFFF;
    while (count--) asm volatile ("nop");
}
```

```
/*****************************************************************
 * @Name    : timer_int_handler
 * @Desc    : Timer to provide correct sampling output rate (to DAC)
 * @Param   : baseaddr_p
 * @Return  : None
 *****************************************************************/
void timer_int_handler(void * baseaddr_p) {
    /* Add variable declarations here */
    unsigned int csr;

    // wav specific vars
    Xuint8 aaa=0;
    Xuint8 bbb=0;
    Xuint8 j=0;
    Xuint16 intCombinedSamples=0;

    static Xuint32 addr = SDRAM_DATA_START_LOC;

    // Read timer 0 CSR to see if it raised the interrupt
    csr = XTmrCtr_mGetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR,0);

    // If the interrupt occured, then increment a counter and set one_second_flag
    if (csr & XTC_CSR_INT_OCCURED_MASK)
    {
        // if zero is reached - it means finished playing song
        if(intDataRemaining > 0){

            aaa=XIo_In8(addr);  // LSBs
            addr++;
            bbb=XIo_In8(addr);  // MSBs
            addr++;

            // concat two 8-bit uint vars to form 16-bit uint var
            intCombinedSamples = aaa | (bbb << 8);

            //intCombinedSamples = bbb | (aaa << 8);
            intCombinedSamples = intCombinedSamples + 32768;
            intDataRemaining=intDataRemaining-2;    // decrement remaining counter

            //xil_printf("(%08X) = %04x\r\n",addr, intCombinedSamples);
            //xil_printf("%04x ", intCombinedSamples);

            // send data to both channels of DAC
            DAC_SendDataToBothChannel(intCombinedSamples,intCombinedSamples);

        }
        else{
            xil_printf("Finished Playing whole song \r\n");
            AuTmr_StopTimer();  // stop current timer (stops interrupt)
            DAC_PowerDownDAC(); // switch off DAC - gets rid of hissing noise

            // reset memory pointers
            addr=SDRAM_DATA_START_LOC;
            intDataRemaining=intSDRAMAddrOffset;

            // pause for a bit..
            pause();

            // start timer to repeat playout process
            AuTmr_StartTimer(SAMPLE_RATE);
        }
    }

    /* Clear the timer interrupt */
    XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR,0,csr);
}
```

```
/*****************************************************************
 * @Name    : main
 * @Desc    : Main function, peripheral initialisation, main loop etc
 * @Param   : None
 * @Return  : None
 *****************************************************************/
int main (void) {

  // local variables
  Xuint32 intInitialTimerVal=0;
  Xuint32 intCurTimerVal=0;
  Xuint32 intCaliberationVal=0;
  Xuint32 intDecoderRunTime=0;

  /*********************************************
   *    Enabling MicroBlaze I&D CACHE
   *********************************************/
  microblaze_disable_icache();
  microblaze_init_icache_range(0,XPAR_MICROBLAZE_0_CACHE_BYTE_SIZE);
  microblaze_enable_icache();

  microblaze_disable_dcache();
  microblaze_init_dcache_range(0,XPAR_MICROBLAZE_0_DCACHE_BYTE_SIZE);
  microblaze_enable_dcache();


  // Clear the screen
  xil_printf("%c[2J",27);

  xil_printf("***** LIBMAD TEST v0.6 ******\r\n");
  xil_printf("****************************\r\n");
  xil_printf("Main:: sizeof(speech_sample_24KHz_64kbps =
%d\r\n",sizeof(chilll_32Kbps_16Khz));
  xil_printf("Main:: (void *)speech_sample_24KHz_64kbps = 0x%X\r\n",(void
*)chilll_32Kbps_16Khz);
  xil_printf("Main:: going to decode now..\r\n");
  xil_printf("Main:: Initialise Timer \r\n");

  /*********************************************
   *    Initialise timer - performance measure
   *********************************************/
   //Initialize and configuring the timer
   intDevStatus = XTmrCtr_Initialize(&XPS_Timer0, XPAR_TIMER_1_DEVICE_ID);
    if ( intDevStatus != XST_SUCCESS ) {
            xil_printf("XTmrCtr_Initialize() failed.\r\n");
    }

   //Set options to the Control register and enable the timer
   XTmrCtr_SetOptions(&XPS_Timer0, TIMER_COUNTER_1, XTC_ENABLE_ALL_OPTION |
XTC_AUTO_RELOAD_OPTION);

   //Read TCR register
   intInitialTimerVal = XTmrCtr_GetValue(&XPS_Timer0, TIMER_COUNTER_1);
   //intCurTimerVal = XTmrCtr_GetValue(&XPS_Timer0, TIMER_COUNTER_1);
   //intCaliberationVal = intCurTimerVal - intInitialTimerVal;
   xil_printf("Decoding Started - Initial Timer value is %d \r\n", intInitialTimerVal);

   //Start timer
   XTmrCtr_Start(&XPS_Timer0, TIMER_COUNTER_1);

   // start the decode process. feed the entire mp3 sample to Libmad
   // decode function.
   xil_printf("=====================================================\r\n");
   decode((void *)chilll_32Kbps_16Khz, sizeof(chilll_32Kbps_16Khz));
   xil_printf("\r\n=====================================================\r\n");

   // Read the TCR register  - report latency
   intCurTimerVal = XTmrCtr_GetValue(&XPS_Timer0, TIMER_COUNTER_1);
```

```
    intDecoderRunTime = intCurTimerVal - intInitialTimerVal;
    xil_printf("Decoding Finished - Total decoding time is %d \r\n", intDecoderRunTime);


    /*********************************************
    *    Finished Decoding - now start playing
    *********************************************/
    intDataRemaining = intSDRAMAddrOffset;

    xil_printf("File Size in SDRAM:  intSDRAMAddrOffset = %08x\r\n", intSDRAMAddrOffset);

    DAC_Initialise();
    DAC_PowerDownDAC();

     /* now start playing the sound */
     /* Register the Timer interrupt handler in the vector table */
     XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,
                           XPAR_XPS_INTC_0_XPS_TIMER_0_INTERRUPT_INTR,
                           (XInterruptHandler) timer_int_handler,
                           (void *)XPAR_XPS_TIMER_0_BASEADDR);

    /* Start the interrupt controller */
    XIntc_mMasterEnable(XPAR_XPS_INTC_0_BASEADDR);

    /* Enable MB interrupts */
    microblaze_enable_interrupts();
    xil_printf("AuTmr_StartTimer : Now output to DAC\r\n");
    AuTmr_StartTimer(SAMPLE_RATE);


   // infinite loop
   while(1){
     // do nothing for now
     }

   return 0;
}

/*****************************************************************
 * This is a private message structure. A generic pointer to this structure
 * is passed to each of the callback functions. These data can be accessed
 * within the callbacks
 * [** Taken from LIBMAD>minimad.c]
 *****************************************************************/
struct buffer {
  unsigned char const *start;
  unsigned long length;
};


/*****************************************************************
 * @Name     : input
 * @Desc     :  This is the input callback. The purpose of this
 *              callback is to (re)fill the stream buffer which is
 *              to be decoded. In this example, an entire file has
 *              been mapped into an array, so we just call
 *              mad_stream_buffer() with the address and length of
 *              the array. When this callback is called a second
 *              time, we are finished decoding.
 *              [** Adapted from LIBMAD>minimad.c]
 * @Param    : void *data, struct mad_stream *stream
 * @Return   : enum mad_flow
 *****************************************************************/
static enum mad_flow input(void *data, struct mad_stream *stream){
  struct buffer *buffer = data;

  #if X_DEBUG
    xil_printf("input::Enter\r\n");
  #endif
```

```
  // if end of buffer length - then stop
  if (!buffer->length)
    return MAD_FLOW_STOP;

  mad_stream_buffer(stream, buffer->start, buffer->length);

  buffer->length = 0;

  return MAD_FLOW_CONTINUE;
}


/*****************************************************************
 * @Name    : scale
 * @Desc    : The following utility routine performs simple rounding,
 *            clipping, and scaling of MAD's high-resolution samples
 *            down to 16 bits. It does not perform any dithering or
 *            noise shaping, which would be recommended to obtain any
 *            exceptional audio quality. It is therefore not recommended
 *            to use this routine if high-quality output is desired.
 *             [** Adapted from LIBMAD>minimad.c]
 * @Param   : mad_fixed_t sample
 * @Return  : signed int (truncated sample)
 *****************************************************************/
static inline signed int scale(mad_fixed_t sample){

  #if X_DEBUG
   xil_printf("scale::Enter\r\n");
  #endif

  /* round */
  sample += (1L << (MAD_F_FRACBITS - 16));

  /* clip */
  if (sample >= MAD_F_ONE)
    sample = MAD_F_ONE - 1;
  else if (sample < -MAD_F_ONE)
    sample = -MAD_F_ONE;

  /* quantize */
  return sample >> (MAD_F_FRACBITS + 1 - 16);
}


/*****************************************************************
 * @Name    : output
 * @Desc    : This is the output callback function. It is called
 *            after each frame of MPEG audio data has been completely
 *            decoded. The purpose of this callback is to output
 *            the decoded PCM audio.
 *            [** Adapted from LIBMAD>minimad.c]
 * @Param   : void *data, struct mad_header const *header,
 *            struct mad_pcm *pcm
 * @Return  : mad_flow
 *****************************************************************/
static enum mad_flow output(void *data,
             struct mad_header const *header,
             struct mad_pcm *pcm)
{
  unsigned int nchannels, nsamples;
  mad_fixed_t const *left_ch, *right_ch;

  #if X_DEBUG
   xil_printf("output::Enter\r\n");
  #endif

  /* pcm->samplerate contains the sampling frequency */
```

```
  nchannels = pcm->channels;
  nsamples  = pcm->length;
  left_ch   = pcm->samples[0];
  right_ch  = pcm->samples[1];

  while (nsamples--) {
    signed int sample;

    /* output sample(s) in 16-bit signed little-endian PCM */

      sample = scale(*left_ch++);   // truncate
     //xil_printf("%02X",(sample >> 0) & 0xff);
     //xil_printf("%02X",(sample >> 8) & 0xff);

     /* write the values to SDRAM increment mem loc after every byte*/
     XIo_Out8(SDRAM_DATA_START_LOC+intSDRAMAddrOffset, (sample>>0)&0xFF);
     intSDRAMAddrOffset++;
     XIo_Out8(SDRAM_DATA_START_LOC+intSDRAMAddrOffset, (sample>>8)&0xFF);
     intSDRAMAddrOffset++;

    /* condition for two channel audio */
    if (nchannels == 2) {
      sample = scale(*right_ch++);
        //xil_printf("%02X",(sample >> 0) & 0xff);
        //xil_printf("%02X",(sample >> 8) & 0xff);
    }
  }

  return MAD_FLOW_CONTINUE;
}


/*****************************************************************
 * @Name    : output
 * @Desc    : This is the error callback function. It is called
 *            whenever a decoding error occurs. The error is indicated
 *            by stream->error; the list of possible MAD_ERROR_
 *            errors can be found in the mad.h (or stream.h) header file.
 *            [** Adapted from LIBMAD>minimad.c]
 * @Param   : void *data, struct mad_stream *stream,struct mad_frame *frame
 * @Return  : mad_flow
 *****************************************************************/
static enum mad_flow error(void *data,
            struct mad_stream *stream,
            struct mad_frame *frame)
{
  struct buffer *buffer = data;
    #if X_DEBUG
        xil_printf("error::Enter\r\n");
    #endif

  xil_printf("decoding error 0x%04x (%s) at byte offset %u\n",
      stream->error, mad_stream_errorstr(stream),
      stream->this_frame - buffer->start);

  /* return MAD_FLOW_BREAK here to stop decoding (and propagate an error) */

  return MAD_FLOW_CONTINUE;
}

/*****************************************************************
 * @Name    : output
 * @Desc    : This is the function called by main() above to perform
 *            all the decoding. It instantiates a decoder object and
 *            configures it with the input, output, and error callback
 *            functions above. A single call to mad_decoder_run()
 *            continues until a callback function returns MAD_FLOW_STOP
 *            (to stop decoding) or MAD_FLOW_BREAK (to stop decoding and
 *            signal an error).
```

```
 *            [** Adapted from LIBMAD>minimad.c]
 * @Param    : unsigned char const *start, unsigned long length
 * @Return   : int (result)
 *****************************************************************/
static int decode(unsigned char const *start, unsigned long length)
{
  struct buffer buffer;
  struct mad_decoder decoder;
  int result;


    #if X_DEBUG
        xil_printf("decode::Enter\r\n");
    #endif

  /* initialize our private message structure */

  buffer.start  = start;
  buffer.length = length;

  /* configure input, output, and error functions */

  mad_decoder_init(&decoder, &buffer,
           input, 0 /* header */, 0 /* filter */, output,
           error, 0 /* message */);

  /* start decoding */

  result = mad_decoder_run(&decoder, MAD_DECODER_MODE_SYNC);

  /* release the decoder */

  mad_decoder_finish(&decoder);

  return result;
}
```

9.7 PROM Bootloader test application

```
/*****************************************************************************
 * Name : PROM Boot-loader (Main Application)
 * FileName: TeatApp.c
 * Author : Rosh Mendis
 * Date Created : 10/08/2011
 * Description : First Disables PROM-1, then starts copying to SDRAM from PROM2,
 *               Then jump to start of SDRAM
 *****************************************************************************/
#include "xparameters.h"
#include "xio.h"
#include "xbasic_types.h"
#include "xgpio.h"
#include "xmpmc.h"
#include "promread.h"


// GPIO to disable PROM-1
XGpio GpioOutputPROM1;

/* program counter setting */
#define PROG_START_ADDR     XPAR_MPMC_0_MPMC_BASEADDR
// Function point that is used at the end of the program to jump to the
// address location stated by PROG_START_ADDR
int (*func_ptr) ();


int main (void) {


    Xuint32 LED_GPIOStatus;
    Xuint32 PROM1_GPIOStatus;

    // Clear the screen
    xil_printf("%c[2J",27);

    // Hello World
    xil_printf("** PROM READ TEST - v2.0 **\r\n");
    xil_printf("**************************\r\n");

    //------------------------------------------------------
    // Set the OE of the PROM-1 to LOW
    //------------------------------------------------------
    // Initialize the GPIO driver
    PROM1_GPIOStatus = XGpio_Initialize(&GpioOutputPROM1,
                    XPAR_PROM1_OE_DEVICE_ID);

    if (PROM1_GPIOStatus != XST_SUCCESS){
    xil_printf("ERROR: PROM1_GPIOStatus failed to initialise\r\n");
    return XST_FAILURE;
    }

    // Set the direction for all signals to be outputs
    XGpio_SetDataDirection(&GpioOutputPROM1, 1, 0x0);

    // set the OE pin low
    XGpio_DiscreteWrite(&GpioOutputPROM1, 1, 0x0);

    //------------------------------------------------------
    // INITIALISE PROMREAD-2
    //------------------------------------------------------
    promread_init();

    //------------------------------------------------------
    // START READING FROM PROM
    //------------------------------------------------------
    promread_start();
```

```
    xil_printf("------------>> Finished reading/writing !\r\n");

    xil_printf("------------>> changing program counter and jumping !\r\n");

    // function point that is set to point to the address of PROG_START_ADDR
    func_ptr = PROG_START_ADDR;
    // jump to start execution code at the address PROG_START_ADDR
    func_ptr();

    xil_printf("------------>> Oops jump didn't work !\r\n");

  while(1){
  }
  return 0;

}
```

```
/*****************************************************************************
 * Name : PROM Boot-loader
 * FileName: promread.c
 * Author : Rosh Mendis
 * Date Created : 10/08/2011
 * Description : Contains functions to:
 *               - Initialise GPIO
 *               - Read from PROM (bit-toggling)
 *              (Code adapted from XAPP482 - Xilinx Inc)
 *
 *****************************************************************************/
#include "xparameters.h"
#include "xgpio.h"
#include "xbasic_types.h"
#include <promread.h>
#include "xio.h"

/* global vars */
XGpio GpioInputPROMREAD_DIN;    // Data in GPIO
XGpio GpioOutputPROMREAD_CNTRL; // PROM2 control GPIO
Xuint32 intSDRAMAddr = XPAR_MPMC_0_MPMC_BASEADDR;   // base address of SDRAM

/*****************************************************************************
* Function name    :  pause (void)
* returns          :  void
* parameters       :  void
* Description      :  Stalls processor for some time
*****************************************************************************/
void pause(void){
    Xuint32 count;
   count=0x00FFFF;
    while (count--) asm volatile ("nop");

}

/*****************************************************************************
* Function name    :  PROMREAD_GetAddrPointer (void)
* returns          :  Xuint32
* parameters       :  void
* Description      :  returns SDRAM end address (getter)
*****************************************************************************/
Xuint32 PROMREAD_GetAddrPointer(){
    return intSDRAMAddr;
}

/*****************************************************************************
* Function name    :  read_helper (void)
* returns          :  Void
* parameters       :  data read word size
* Description      :  Reads Platform FLASH  device serially 32 time and puts
*                     it into 32-bit word
*****************************************************************************/
```

```
Xuint32 prom_read_helper( Xuint8 num)
{
    Xuint8 i;
    Xuint32 data32=0;

    for( i=0;i<num;i++){

        //single bit is shifted into the 32-bit word.
        data32 = (data32 << 1) + XGpio_DiscreteRead(&GpioInputPROMREAD_DIN, 1);

        //clock the PROM to output data - (toggle CCLK)
        XGpio_DiscreteWrite(&GpioOutputPROMREAD_CNTRL, 1, OE_HIGH | CCLK_HIGH | CE_LOW);
        XGpio_DiscreteWrite(&GpioOutputPROMREAD_CNTRL, 1, OE_HIGH | CCLK_LOW | CE_LOW);
    }

    return data32;
}

/*****************************************************************************
* Function name    :  promread_init(void)
* returns          :  int (success/fail)
* parameters       :  None
* Description      :  Initialises the GPIO IP Cores
*****************************************************************************/
int promread_init(void){


    Xuint32 PROMREAD_DIN_status;
    Xuint32 PROMREAD_CNTRL_status;

    //-----------------------------------------------------
    // INITIALISE PROM DIN Connections (GPIO) - DIN
    //-----------------------------------------------------

    xil_printf("  - Setting up the PROMREAD-DIN connections\r\n");

    // Initialize the GPIO driver
    PROMREAD_DIN_status = XGpio_Initialize(&GpioInputPROMREAD_DIN,
                          XPAR_PROMREAD_DIN_GPIO_DEVICE_ID);

    if (PROMREAD_DIN_status != XST_SUCCESS){
    xil_printf("ERROR: PROMREAD_DIN failed to Initialise\r\n");
    return XST_FAILURE;
    }

    // Set the direction for all signals to be input
    XGpio_SetDataDirection(&GpioInputPROMREAD_DIN, 1, 0x1);

    //-----------------------------------------------------
    // INITIALISE PROM CNTRL Connections (GPIO) - CCLK, CE, OE
    //-----------------------------------------------------

    xil_printf("  - Setting up the PROMREAD-CNTRL connections\r\n");

    // Initialize the GPIO driver
    PROMREAD_CNTRL_status = XGpio_Initialize(&GpioOutputPROMREAD_CNTRL,
                          XPAR_PROMREAD_CNTRL_GPIO_DEVICE_ID);

    if (PROMREAD_CNTRL_status != XST_SUCCESS){
    xil_printf("ERROR: PROMREAD_CNTRL failed to Initialise\r\n");
    return XST_FAILURE;
    }

    // Set the direction for all signals to be outputs
    XGpio_SetDataDirection(&GpioOutputPROMREAD_CNTRL, 1, 0x0);

    //sets the INIT and CCLK bits high  (3 bits)
    XGpio_DiscreteWrite(&GpioOutputPROMREAD_CNTRL, 1, (OE_HIGH | CCLK_LOW | CE_LOW));
```

```
    return 0;

}

/*****************************************************************************
* Function name    :  promread_start()
* returns          :  int
* Created by       :  HRM
* Date Created     :  21/07/2011
* Description      :  Start reading and reporting the readout from prom
*****************************************************************************/
int promread_start(void){

    Xuint32 data    = 0;
    Xuint32 count   = 0;

    pause();

    xil_printf("promread_start:: enter\r\n");

    /* search prom until APPLICATION market is found */
    while(data != ADDRSYNC){
        data = prom_read_helper(32);
        //xil_printf("%08X ", data);
    }

    xil_printf("addrsync found!\r\n");
    xil_printf("now getting application data!\r\n");

    /* copy remaining data into SDRAM */
    while(data != 0xFFFFFFFF){  // read until END Marker
        data = prom_read_helper(32);
        //xil_printf("%08X ", data);

        XIo_Out32(intSDRAMAddr, data);      // write out data to SDRAM

        /* for every write check the read */
        if(XIo_In32(intSDRAMAddr) != data){
            xil_printf("SDRAM write unsuccesful!\r\n");
            return 0;
        }
        intSDRAMAddr = intSDRAMAddr+4; // increment sdram address pointer
    }

    xil_printf("Finished copying data to SDRAM, end addr = 0x%08X\r\n", intSDRAMAddr);

    return 0;
}
```

```
/*****************************************************************************
 * Name : PROM Boot-loader
 * FileName: promread.h
 * Author : Rosh Mendis
 * Date Created : 10/08/2011
 * Description : Contains defines and function prototypes
 *****************************************************************************/

#define ADDRSYNC 0x9F8FAFBF
#define END_PROM 0xFFFFFFFF

#define OE_HIGH     0x2   // 0010
#define OE_LOW      0x0
#define CCLK_HIGH   0x4
#define CCLK_LOW    0x0
#define CE_HIGH     0x1
#define CE_LOW      0x0

#define DATAREAD 0x0
```

```
#define ADDRREAD 0x1

//to enable debugging to UART uncomment the following
#define DEBUG

#ifdef DEBUG
#define INFO(format,args...) xil_printf(format, ##args)
#else
#define INFO(format,args...)
#endif


// prototypes
int promread_init(void);
Xuint32 prom_read_helper( Xuint8 num);
int promread_start(void);
Xuint32 PROMREAD_GetAddrPointer();
```

9.8 ADPCM decoder C-modules (adpcm.c, adpcm.h)

```
/***********************************************************
 * Name        : ADPCM - Decoder (adpcm.c)
 * Modified by : Rosh Mendis
 * Revisions   : v2.0
 * Description : Original Algorithm taken from Stichting Mathematisch
 *               Centrum, Amsterdam.
 *               Added:
 *               - Getters, Setters for globals
 *               - Added Header support
 *               - Added support for Rewind/FF
 *               - Added Volume control
 *               - Added Get Progress of song
 ***********************************************************/

/************************************************************
Copyright 1992 by Stichting Mathematisch Centrum, Amsterdam, The
Netherlands.

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the names of Stichting Mathematisch
Centrum or CWI not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

************************************************************/
/*
** Intel/DVI ADPCM coder/decoder.
**
** The algorithm for this coder was taken from the IMA Compatability Project
** proceedings, Vol 2, Number 2; May 1992.
**
** Version 1.0, 7-Jul-92.
*/

/* xilinx includes */
#include "xio.h"
#include "xstatus.h"

/* local application includes */
#include "adpcm.h"
#include "DAC.h"
#include "autmr.h"


#ifndef __STDC__
#define signed
#endif

// this is the multiplication format (fast/slow mult)
#define NODIVMUL


/* globals used by ADPCM decoder */
static struct adpcm_state state;
static char   inbuf[ADPCM_NSAMPLES/2]={0};
```

```
static short  outbuf[ADPCM_NSAMPLES]={0};
static int    intADPCMSampleCount=0;
static int    intOutBuffIndex=0;
static int    intInBuffIndex=0;
static int    intTotalFileSize=0;
static int    intFileStartAddr=0;
static int    intStopAddr=0;
static int    intSampleRate=0;
static int    intNumChannels=0;
static int    intBitsPerSample=0;
static Xuint8 intVolume=15;


/* local prototypes */
void ADPCM_Pause(void);


/* Intel ADPCM step variation table */
static long indexTable[16] = {
    -1, -1, -1, -1, 2, 4, 6, 8,
    -1, -1, -1, -1, 2, 4, 6, 8,
};

// stepsize table used to quantize the difference of the Original vs. Predicted
// sample value. Hence max value is = (2^16)-1
static long stepsizeTable[89] = {
    7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
    19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
    50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
    130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
    337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
    876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
    2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
    5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
    15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767
};

/*****************************************************************
 * @Name    : ADPCM_Decoder
 * @Desc    : Adaptive Differential Pulse Code Modulation-Decoder
 * @Param   : char indata[] - input data buffer
 *            short outdata[] - output data buffer
 *            int len - size of input buffer
 *            struct adpcm_state *state - previous state info
 * @Return  : None
 *****************************************************************/
void ADPCM_Decoder(char indata[], short outdata[], int len, struct adpcm_state *state){
    signed char *inp;          /* Input buffer pointer */
    short *outp;               /* output buffer pointer */
    long sign;                  /* Current adpcm sign bit */
    long delta;                 /* Current adpcm output value (difference value) */
    long step;                  /* Stepsize */
    long valprev;               /* virtual previous output value */
    long vpdiff;                /* Current change to valprev */
    long index;                 /* Current step change index */
    long inputbuffer;           /* place to keep next 4-bit value */
    long bufferstep;            /* toggle between inputbuffer/input */

    outp = outdata;
    inp = (signed char *)indata;

    valprev = state->valprev;
    index = state->index;
    step = stepsizeTable[index];

    bufferstep = 0;

    len *=2;
```

```
    for ( ; len > 0 ; len-- ) {

        /* Step 1 - get the delta value and compute next index */
        if ( bufferstep ) {
            delta = inputbuffer & 0xf;           // get MSB 4 bits
        } else {
            inputbuffer = *inp++;                // get next data value (8bit)
            delta = (inputbuffer >> 4) & 0xf;    // get LSB 4 bits
        }
        bufferstep = !bufferstep;    // invert the variable

         /* Step 2 - Find new index value (for later) */
         index += indexTable[delta];
         // index must lie between 0 and 88 (else buffer overflow!)
         if ( index < 0 ){
             index = 0;
         }
         else if ( index > 88 ){
             index = 88;
         }

        /* Step 3 - Separate sign and magnitude */
        sign = delta & 8;         // 8 = 1000     (get MSB bit)
        delta = delta & 7;        // 7 = 0111     (get LSB 3 bits)

        /* Step 4 - update output value */
        // check what multiplication method to use (fast/slow)
#ifdef NODIVMUL
         // fast multiplication (shifting)
         vpdiff = 0;
         if ( delta & 4 ){
             vpdiff  = (step << 2);
         }
         if ( delta & 2 ){
             vpdiff += (step << 1);
         }
         if ( delta & 1 ){
             vpdiff += step;
         }
         vpdiff >>= 2;
#else
         // slow multiplication (expensive)
         vpdiff = (delta*step) >> 2;
#endif

        if ( sign )
          valprev -= vpdiff; // subtract the difference if sign is 1 (negative)
        else
          valprev += vpdiff; // add back the difference if sign is 0 (positive)

        /* Step 5 - clamp output value */
        if ( valprev > 32767 )
          valprev = 32767;
        else if ( valprev < -32768 )
          valprev = -32768;

        /* Step 6 - Update step value */
        step = stepsizeTable[index];

        /* Step 7 - Output value */
        *outp++ = valprev;
    }

    // save state info for next block
    state->valprev = valprev;
    state->index = index;
}

/*****************************************************************
```

```
 * @Name    : ADPCM_CheckADP_header
 * @Desc    : Checks the ADPCM file header, and returns false if
 *            'ADPCM' tag is not found. If true, extracts SampleRate,
 *            NumChannels, BitsPerSample
 * @Param   : None
 * @Return  : True/False
 *****************************************************************/
Xuint8 ADPCM_CheckADP_header(void){

    int ix=0;

    ix=intInBuffIndex;

    // check if file has the 'ADPCM' tag
    if ((XIo_In8(ix) == 0x41) && (XIo_In8(ix+1) == 0x44) &&
        (XIo_In8(ix+2) == 0x50) && (XIo_In8(ix+3) == 0x43) &&
        (XIo_In8(ix+4) == 0x4D)){

           intNumChannels = XIo_In8(ix+5);
           xil_printf("\r\n");
           xil_printf("ADPCM_CheckADP_header:: intNumChannels = %d\r\n", intNumChannels);

           intSampleRate = (Xuint32)XIo_In8(ix+6)+((Xuint32)XIo_In8(ix+7)<<8) +\
                           ((Xuint32)XIo_In8(ix+8)<<16) + ((Xuint32)XIo_In8(ix+9)<<24);

           xil_printf("ADPCM_CheckADP_header:: intSampleRate = %d\r\n", intSampleRate);

           intBitsPerSample = (Xuint32)XIo_In8(ix+10)+((Xuint32)XIo_In8(ix+11)<<8);
           xil_printf("ADPCM_CheckADP_header:: intBitsPerSample = %d\r\n",
intBitsPerSample);

           xil_printf("\r\n");

           // change the start buffer index
           intInBuffIndex=ix+12;

           return 1;

        }
    else{
             return 0;
    }
}



/*****************************************************************
 * @Name    : ADPCM_Pause
 * @Desc    : Stalls processor in a loop
 * @Param   : None
 * @Return  : None
 *****************************************************************/
void ADPCM_Pause(void){
    Xuint32 count;
    count=0x00000003;
    while (count--) asm volatile ("nop");
}

/*****************************************************************
 * @Name    : ADPCM_MainLoop
 * @Desc    : ADPCM has it's own loop, decodes the block of data
 *            within the loop. After every decode the timer is started
 *            to output the PCM samples
 * @Param   : None
 * @Return  : None
 *****************************************************************/
void ADPCM_MainLoop(void){

    int j=0;
```

```
    //-----------------------------------------------------
    // ADPCM MAIN LOOP - goes on until end of file
    //-----------------------------------------------------
    //xil_printf("sizeof(inbuf) = %d\r\n",sizeof(inbuf));
    //xil_printf("sizeof(outbuf) = %d\r\n",sizeof(outbuf));
    xil_printf("ADPCM_MainLoop:: enter\r\n");

    /* loop while the pointer in memory is lower than total filesize */
    while(intInBuffIndex<intStopAddr){

        //xil_printf("ADPCM_MainLoop:: inside loop\r\n");

        ADPCM_Pause();

        /* only start to decode the next batch of adpcm samples
           only when the previous samples have been completely
           decoded.
         */
        if((intADPCMSampleCount == 0) && (intInBuffIndex<intTotalFileSize)){

            // disable all interrupts - critical section
            microblaze_disable_interrupts();

            // -- start decoding ADPCM_NSAMPLES of the adpcm file --
            // fils in input buffer
            for (j=0;j<(ADPCM_NSAMPLES/2);j++){
                inbuf[j]=XIo_In8(intInBuffIndex+j); // get data from SDRAM, byte by byte

                //xil_printf("ADPCM_MainLoop:: inbuff fill [%08X] =
0x%02X\r\n",intInBuffIndex+j,inbuf[j]);
            }

            // call the adpcm decoder to decode the NSAMPLES
            ADPCM_Decoder(inbuf, outbuf, sizeof(inbuf), &state);

            // reset the counter again to count down
            intADPCMSampleCount=ADPCM_NSAMPLES;

            //increment the adpcm file pointer (get next chunk of data from SDRAM)
            intInBuffIndex=intInBuffIndex+(ADPCM_NSAMPLES/2);

            //reset the outbuff index
            intOutBuffIndex=0;

            // enable the interrupt - so the samples can be output
            AuTmr_StartTimer(intSampleRate);

            // enable all interrupts - critical section finished
            microblaze_enable_interrupts();
        }
        else{
            // print reason why it didnt decode
            //xil_printf("main:: intADPCMSampleCount = %d\r\n", intADPCMSampleCount);
        }
    }
    xil_printf("ADPCM_MainLoop:: Finished ADPCM file playing !!\r\n");
    AuTmr_StopTimer();
    //xil_printf("ADPCM_MainLoop::exit intInBuffIndex = %08x\r\n", intInBuffIndex);

}



/*****************************************************************
 * @Name    : ADPCM_PlayOut
 * @Desc    : The outbuf contains the decoded PCM. Everytime the
 *            interrupt kicks in this function is called to output
 *            data to the DAC.
```

```
 * @Param    : None
 * @Return   : None
 ***************************************************************/
void ADPCM_PlayOut(void){

    // adpcm specific
    Xuint8 intLSBs=0;
    Xuint8 intMSBs=0;
    Xuint16 intDACdata=0;


    //xil_printf("ADPCM_PlayOut::enter \r\n");

    if(intOutBuffIndex<(ADPCM_NSAMPLES+1)){

        // split into 8 bit groups
        intLSBs = outbuf[intOutBuffIndex];
        intMSBs = outbuf[intOutBuffIndex] >> 8;

        // reverse bit order and adjust unsigned scalling
        intDACdata = intLSBs | (intMSBs << 8);  // swap bit around
        intDACdata = intDACdata + 32768;

        // set the appropriate volume (0->15)
        intDACdata = intDACdata*intVolume/15;

        // send data to dac (mono)
        DAC_SendDataToBothChannel(intDACdata,intDACdata);

        //xil_printf("ADPCM_PlayOut:: Sent to DAC=, 0x%04X \r\n", intDACdata);

        // decrement sample counter: when this gets to zero
        // the next batch of samples will be decoded.
        intADPCMSampleCount--;
        intOutBuffIndex++;
    }
    else{

        DAC_PowerDownDAC();

        intADPCMSampleCount=0;
        intOutBuffIndex=0;

        // disable the interrupt - so the next batch can be decoded
        AuTmr_StopTimer();
        //XIntc_mDisableIntr(XPAR_XPS_INTC_0_BASEADDR, XPAR_XPS_TIMER_0_INTERRUPT_MASK);
    }
}

/***************************************************************
 * @Name     : ADPCM_FastForward
 * @Desc     : adds 1 to the input buffer pointer
 * @Param    : None
 * @Return   : None
 ***************************************************************/
void ADPCM_FastForward(void){
    //intInBuffIndex = intInBuffIndex+(ADPCM_NSAMPLES/2);

    if((intInBuffIndex+1) > intStopAddr){
        intInBuffIndex = intInBuffIndex;
    }
    else{
        //intInBuffIndex = intInBuffIndex + (ADPCM_NSAMPLES/2);
        intInBuffIndex++;
    }
}

/***************************************************************
 * @Name     : ADPCM_Rewind
```

```
 * @Desc    : subtracts 1 from the input buffer pointer
 * @Param   : None
 * @Return  : None
 ***************************************************************/
void ADPCM_Rewind(void){
    //intInBuffIndex = intInBuffIndex-(ADPCM_NSAMPLES/2);

    if((intInBuffIndex-1) < intFileStartAddr){
        intInBuffIndex = intInBuffIndex;
    }
    else{
        //intInBuffIndex = intInBuffIndex - (ADPCM_NSAMPLES/2);
        intInBuffIndex--;
    }
}

/***************************************************************
 * @Name    : ADPCM_SetVolume
 * @Desc    : sets the volume (0->15). this value is then scalled to
 *            a fraction and multiplied with the DAC output sample.
 * @Param   : intV (volume)
 * @Return  : None
 ***************************************************************/
 void ADPCM_SetVolume(Xuint8 intV){
    if(intV < 16){
        intVolume = intV;
    }
    else{
    // max volume level reached for ADPCM
        intVolume = intVolume;
    }
}


/***************************************************************
 * GETTERS AND SETTERS for GLOBAL Vars
 ***************************************************************/
void ADPCM_SetOutBuffIndex(int intO){
    intOutBuffIndex = intO;
}

void ADPCM_SetInBuffIndex(int intI){
    intInBuffIndex = intI;
}

void ADPCM_SetSampleCount(int intSC){
    intADPCMSampleCount = intSC;
}

int ADPCM_GetSampleCount(void){
    return intADPCMSampleCount;
}

int ADPCM_GetInBuffIndex(void){
    return intInBuffIndex;
}

int ADPCM_GetOutBuffIndex(void){
    return intOutBuffIndex;
}

int ADPCM_GetSampleRate(void){
    return intSampleRate;
}


/***************************************************************
 * @Name    : ADPCM_GetSongProgress
 * @Desc    : Calculates song progress.
```

```
 *           (input buffer pointer/file end position) *100%
 * @Param    : None
 * @Return   : Xuint8 intProgress (%)
 ***************************************************************/
Xuint8 ADPCM_GetSongProgress(void){

    Xuint8 intProgress = 0;
    intProgress = ((intInBuffIndex/intStopAddr)*100);

    return intProgress;
}

/*************************************************************
 * @Name     : ADPCM_Initialise
 * @Desc     : Initialises the Global variables. Has to be called
 *             befoe playing a new song. The decoder must know the
 *             memory end address and start address.
 * @Param    : int intSDRAMOffset (end addr)
               int intFSAdr       (start addr)
 * @Return   : None
 ***************************************************************/
void ADPCM_Initialise(int intSDRAMOffset, int intFSAdr){

    intADPCMSampleCount = 0;
    intOutBuffIndex     = 0;
    intInBuffIndex      = intFSAdr;
    intTotalFileSize    = intSDRAMOffset;
    intFileStartAddr    = intFSAdr; // file start address (in SDRAM)

    // have to deduct 19KB (0x4c00) from file (for some unknown reason)
    //intStopAddr           = (intFSAdr+intSDRAMOffset)-(33*ADPCM_NSAMPLES);
    intStopAddr         = (intFSAdr+intSDRAMOffset)-0x4C00;
    //intStopAddr           = (intFSAdr+intSDRAMOffset);

    intSampleRate=0;
    intNumChannels=0;
    intBitsPerSample=0;

    xil_printf("ADPCM_Initialise: intTotalFileSize = %08X\r\n", intTotalFileSize);
    xil_printf("ADPCM_Initialise: intInBuffIndex = %08X\r\n", intInBuffIndex);
    xil_printf("ADPCM_Initialise: intStopAddr = %08X\r\n", intStopAddr);

}
```

```
/*************************************************************
 * Name         : ADPCM - Decoder (Header) (adpcm.h)
 * Modified by  : Rosh Mendis
 * Revisions    : v2.0
 * Description  : Original Algorithm taken from Stichting Mathematisch
 *                Centrum, Amsterdam.
 *                Added:
 *                - Getters, Setters for globals
 *                - Added Header support
 *                - Added support for Rewind/FF
 *                - Added Volume control
 *                - Added Get Progress of song
 ***************************************************************/
#ifndef ADPCM__H
#define ADPCM__H



#define ADPCM_NSAMPLES          50
//#define ADPCM_FIXED_SAMPLERATE    16000

struct adpcm_state {
    short      valprev;        /* Previous output value */
```

```
    char       index;          /* Index into stepsize table */
};

// main decode logic
extern void ADPCM_Decoder(char [], short [], int, struct adpcm_state *);

// getters and setters
extern void ADPCM_SetOutBuffIndex(int intO);
extern void ADPCM_SetInBuffIndex(int intI);
extern void ADPCM_SetSampleCount(int intSC);
extern int ADPCM_GetSampleCount(void);
extern int ADPCM_GetInBuffIndex(void);
extern int ADPCM_GetOutBuffIndex(void);
extern int ADPCM_GetSampleRate(void);
extern void ADPCM_SetVolume(Xuint8 intV);

// initialise - set ram address etc.
extern void ADPCM_Initialise(int intSDRAMOffset, int intFSAdr);

// play out the decoded samples
extern void ADPCM_PlayOut(void);

// decode the next chunk of data - do this in a loop.
extern void ADPCM_MainLoop(void);

// extract the header info from the song
extern Xuint8 ADPCM_CheckADP_header(void);

// rewind and ffwd
extern void ADPCM_FastForward(void);
extern void ADPCM_Rewind(void);

// calculate progress
extern Xuint8 ADPCM_GetSongProgress(void);

#endif
```

9.9 ADPCM encoder wrapper GUI tool (written in wxPython)

WAV2ADPCM.py – Main Contianer application

```
import wx

import MainFrame
import Buttons


app = MainFrame.App(0)
app.MainLoop()
```

MainFrame.py – Primary Frame window

```
###########################################################################
#   MainFrame.py
#   Contains 3 buttons inside a toolbar object, 2 list control objects and
#   splitters. Outputs encoded files to ADPCM_OUTPUT. Maintains log file
#
#   Author: Rosh Mendis (adapted from http://zetcode.com/wxpython/draganddrop/)
###########################################################################
import wx
import os
import subprocess
from time import gmtime, strftime

from pprint import pprint
from TextDropTarget import TextDropTarget

class MainFrame(wx.Frame):


    def __init__(self, parent, id, title):

        # ==== global vars ====
        self.FileList = [];   # list of files the user selected
        self.CurrentDir='';

        # ==== draw the toolbar Buttons ====
        wx.Frame.__init__(self, parent, id, title, wx.DefaultPosition, wx.Size(900, 400))
        toolbar = wx.ToolBar(self, 1, style=wx.TB_HORIZONTAL | wx.NO_BORDER)
        toolbar.AddControl(wx.Button( toolbar, 1, "Encode", (1, 1), wx.DefaultSize ) )
        wx.EVT_BUTTON( self, 1, self.MnbEncodeEvent )
        toolbar.AddControl(wx.Button( toolbar, 2, "Select-All", (80, 1), wx.DefaultSize ) )
        wx.EVT_BUTTON( self, 2, self.MnbSelectAllEvent )
        toolbar.AddControl(wx.Button( toolbar, 3, "Unselect-All", (160, 1), wx.DefaultSize
) )
        wx.EVT_BUTTON( self, 3, self.MnbUnSelectAllEvent )
        self.SetToolBar(toolbar)

        # ==== draw the file lists, splitters etc ====
        splitter1 = wx.SplitterWindow(self, -1, style=wx.SP_3D)
        splitter2 = wx.SplitterWindow(splitter1, -1, style=wx.SP_3D)
        self.dir = wx.GenericDirCtrl(splitter1, -1, dir='D:/cygwin/home/DSE/adpcm',
style=wx.DIRCTRL_DIR_ONLY)
        self.lc1 = wx.ListCtrl(splitter2, -1, style=wx.LC_LIST)
        self.lc2 = wx.ListCtrl(splitter2, -1, style=wx.LC_LIST)
        dt = TextDropTarget(self.lc2)
        self.lc2.SetDropTarget(dt)
        wx.EVT_LIST_BEGIN_DRAG(self, self.lc1.GetId(), self.OnDragInit)
        wx.EVT_LIST_ITEM_RIGHT_CLICK(self, self.lc2.GetId(), self.OnLC2RC)
```

```
        tree = self.dir.GetTreeCtrl()
        splitter2.SplitHorizontally(self.lc1, self.lc2)
        splitter1.SplitVertically(self.dir, splitter2)
        wx.EVT_TREE_SEL_CHANGED(self, tree.GetId(), self.OnSelect)
        self.OnSelect(0)
        self.Centre()

    def OnLC2RC(self, event):
        #remove from logical list
        text = self.lc2.GetItemText(event.GetIndex())
        self.FileList.remove(text)
        # remove from graphical list
        self.lc2.DeleteItem(event.GetIndex())


    def OnSelect(self, event):
        self.CurrentDir = self.dir.GetPath(); # update current dir
        list = os.listdir(self.dir.GetPath())

        self.lc1.ClearAll()
        self.lc2.ClearAll()
        for i in range(len(list)):
            if list[i][0] != '.':
                if '.wav' in list[i]:   # filter by only 'wav' files
                    self.lc1.InsertStringItem(0, list[i])

    def OnDragInit(self, event):
        text = self.lc1.GetItemText(event.GetIndex())
        tdo = wx.PyTextDataObject(text)

        # only add to the list if not already in list
        if tdo.GetText() not in self.FileList:
            tds = wx.DropSource(self.lc1)
            tds.SetData(tdo)
            tds.DoDragDrop(True)
            self.FileList.append(tdo.GetText());


    def MnbEncodeEvent(self, event):
        FileList_w_fullpath=[]

        # construct the file list (with full path)
        for item in self.FileList:
            FileList_w_fullpath.append(self.CurrentDir+'\\'+item)

        # call the ADPCM Encoder for each file and save result in special folder
        OutputFolderName = 'ADPCM_OUTPUT'

        # if output folder does not exist create it
        if(os.path.exists(self.CurrentDir+'\\'+OutputFolderName)==False):
            os.mkdir(self.CurrentDir+'\\'+OutputFolderName)

        # --- write to log file ---
        LOGFILE = open("WAV2ADPCM.log","a")
        LOGFILE.write(" "); #empty line
        LOGFILE.write("************************ <Batch Started>
************************\r\n")
        LOGFILE.write(strftime("%Y-%m-%d %H:%M:%S", gmtime())+"\r\n")

        i=0
        for item in FileList_w_fullpath:
            ext_command = 'ADPCM_Enc.exe ' + "\""+item+"\" "+"\""+ self.CurrentDir + "\\" +
OutputFolderName + "\\" + self.FileList[i].replace('wav','adp')+"\""
            process = subprocess.Popen(ext_command,shell=True, stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
            #process.wait()
            err_output = process.communicate()[1]
            std_output = process.communicate()[0]
```

```
            # --- write to log file ---
            LOGFILE.write(ext_command+"\r\n")
            LOGFILE.write(std_output+"\r\n")
            LOGFILE.write(err_output+"\r\n")

            i=i+1; #increment counter to get next filename

        LOGFILE.write("************************ <!Batch Ended!>
************************\r\n")
        LOGFILE.close() # close log file

        #pprint(FileList_w_fullpath)


    def MnbSelectAllEvent(self, event):
        list = os.listdir(self.dir.GetPath())

        # reset the second list control
        self.FileList =[]
        self.lc2.ClearAll()

        # populate the second list control with the items in the first one
        for idx in range(self.lc1.GetItemCount()):
            item = self.lc1.GetItem(idx)
            self.lc2.InsertStringItem(0, item.GetText()) # add to physical list
            self.FileList.append(item.GetText()) # add to logical list


    def MnbUnSelectAllEvent(self, event):
        # remove all logical items from list
        self.FileList =[];

        # remove all physical items
        self.lc2.ClearAll()
        self.lc2.DeleteAllItems()


class App(wx.App):
    def OnInit(self):
        frame = MainFrame(None, -1, "WAV-to-ADPCM-Converter")
        frame.Show(True)
        self.SetTopWindow(frame)
        return True
```

TextDropTarget.py – Drag and Drop component

```
import wx

class TextDropTarget(wx.TextDropTarget):
    def __init__(self, object):
        wx.TextDropTarget.__init__(self)
        self.object = object

    def OnDropText(self, x, y, data):
        self.object.InsertStringItem(0, data)
```