CHALMERS

DAT095 Project Renewal

Implementation of a MP3 Player on a FPGA

Master of Science Thesis in the Programme Integrated Electronic System Design

RECEP GÖKHAN ASLAN
CEMIL CAGLAR BÖKE

Chalmers University of Technology / University Of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, August 2011

DAT095 Project Renewal
Implementation of MP3 player on FPGA

Recep Gökhan ASLAN
Cemil Caglar BÖKE

Examiner: Sven Knutsson

Chalmers University of Technology / University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden August 2011

Abstract

This thesis has the intention to create a base for renewal of the DAT095 (Electronic System Design Project) course. As a basis for the new project, implementation of a LEON3 processor on a FPGA board was done and a MP3 player application was run on it. The MPG123 [15] application's source code was used and modified according to the system and by using hardware/software co-design techniques a complete system was designed. The audio interface hardware core was designed according to the requirements of the digital to analog converter MCP4288 [3]. Necessary interfaces were implemented according to the AMBA bus. A demonstrator was built on the Digilent Spartan3 xc3s1000 board [4]. During the analysis of the MP3 decoder, it was seen that the Inverse Discrete Cosine Transform (IDCT) part of the decoder algorithm was too computation-intensive and a hardware implementation for that part was made and attached to the processor's AMBA bus as a slave. The MP3 decoder software and the IDCT hardware were working together to decode the data.

Finally the development platform was changed to Digilent Atlys Spartan6 FPGA development board [21] that gave a more flexible usage for future works. The LEON3 processors template design was modified according to the needs of the new development platform and the MPG123 application was run on it.

Acknowledgements

We would like to give our special thanks to our supervisor Sven Knutsson for his supervision and support of this thesis and to Doctor Magnus Själander for his support and comments to us.

Abbreviations

AHB : Advanced High Performance Bus

AMBA : Advanced Microcontroller Bus Architecture

APB : Advanced Peripheral Bus

ASB : Advanced System Bus

DAC: Digital to Analog Converter

DCT : Discrete Cosine Transform

FPGA : Field Programmable Gate Array

FPU : Floating Point Unit

IDCT : Inverse Discrete Cosine Transform

MMU: Memory Management Unit

OS : Operating System

PCM : Pulse Code Modulation

SPARC: Scalable Processor Architecture

VHDL : Very High Speed Integrated Circuit (VHSIC) Hardware Description Language

Table of Contents

List of Figures and Tables

1 Introduction

This chapter explains the background of this thesis project, purpose, objective and thesis tasks which lead the studies for developing the project and methods used on thesis project to achieve set goals.

1.1 Background

Today is the world of embedded systems and processors as they have real time performance designed for a specific purpose and they offer a simplified system hardware which reduces the manufacturing costs. Lots of embedded systems are designed according to needs of technology and processors are the main parts of those embedded systems. In an embedded system, processors can be used for running applications and suitable hardware interface designs can be used for getting outputs.
The usage of embedded systems are increasing day by day and lot more work on embedded systems needs to be done. Therefore renewal of projects according to new processors and applications become requisite. These new projects will help students understand the basics of embedded systems; giving them the taste of working with embedded processors combined with hardware and giving them the experience of how hardware and software co-design can be done. For example, an embedded system which has a processor that reads inputs from an interface and performs some form of signal processing on these, before passing them on to an output interface can be realized. Moreover in this embedded system, from input to output, some part of the software can be implemented as an accelerator in hardware for accelerating the system to meet the performance constraints. Accelerator design can achieve this by realizing excessive computations with extra hardware and working in parallel with the software.

1.2 Objective and Tasks

The purpose of this master's thesis is to implement the LEON3 [1] processor on a FPGA board and run a suitable application on it. This master's thesis will be used later as a reference design to develop a project that can be used as the base for renewal of the DAT095 (Electronic System Design Project) course. The new project is intended to be

more system oriented and to achieve this low level software and general purpose processors should be introduced. To achieve the objective of this thesis, five tasks are introduced at the beginning of the studies. These tasks are:

1. Identifying an application that has a suitable computational kernel, preferably more than one; that can be isolated and improved by a hardware accelerator.

2. Based on the selected application, identify required input and output interfaces and the overall system design. This mainly consists of selecting suitable IP cores from GRLIB (refer to section 1.4.3) and to assemble and configure the system.

3. Considering the runtime system. Should the application be executed directly on LEON3 or should an operating system be used.

4. Adapting the application to achieve a suitable interface for hardware acceleration. Algorithmic changes might also be required to allow for efficient hardware implementations, e.g., removal of floating point operations.

5. Implementing a suitable kernel in hardware that is accessible over the AMBA bus (refer to section 1.4.4) and interfacing it to the software.

1.3 Method

Thesis work starts with research on GRLIB [1], the LEON processor and software for a suitable application. A FPGA board needs to be selected as development platform according to the requirements of selected application and it should preferably be supported by GRLIB. Reconfiguration of the software to be used in a LEON based embedded system is required. Analyzing the application for computation excessive parts and implementing those parts in hardware to accelerate embedded system is a critical part to achieve.

1.4 Project Materials

1.4.1 Development Platform

The development platform used in this project is the Digilent Spartan3 FPGA board [4]. It includes the FPGA XC3S1000-FT256 with on-board I/O devices and 1MB fast asynchronous SRAM. It is supported by the GRLIB IP library and it has 1000K gates and it is large enough for the LEON3 processor to be implemented. Moreover, it has 40-pin expansion connectors which are used for attaching a DAC card to get audio outputs. The board has these significant features;

- Xilinx Spartan-3 FPGA with twelve 18-bit multipliers, 216Kbits of block RAM, and up to 500MHz internal clock speeds
- -200 and -1000 versions available
- On-board 2Mbit Platform Flash (XCF02S)
- 8 slide switches, 4 pushbuttons, 9 LEDs, and 4-digit seven-segment display
- Serial port, VGA port, and PS/2 mouse/keyboard port
- Three 40-pin expansion connectors
- Three high-current voltage regulators (3.3V, 2.5V, and 1.2V)
- Works with Digilent's JTAG3, JTAG USB, and JTAG USB Full Speed cables, as well as P4 & MultiPRO cables from Xilinx
- 1Mbyte on-board 10ns SRAM (256Kb x 32)

1.4.2 LEON3 Processor

The LEON3 is the IP core of a 32-bit processor compliant with the SPARC V8 architecture [9] and it is distributed as a part of the GRLIB IP library provided by Aeroflex Gaisler [1] with the GNU GPL license. The LEON3 processor has the following features:

- SPARC V8 instruction set with V8e extensions
- Advanced 7-stage pipeline
- Hardware multiply, divide and MAC units
- Separate instruction and data cache (Harvard architecture) with snooping

- Configurable caches: 1 - 4 ways, 1 - 256 Kbytes/way. Random, LRR or LRU replacement
- Local instruction and data scratch pad RAM, 1 - 512 Kbytes
- SPARC Reference MMU (SRMMU) with configurable TLB
- AMBA-2.0 AHB bus interface
- Advanced on-chip debug support with instruction and data trace buffer
- Large range of software tools: compilers, kernels, simulators and debug monitors

1.4.3 GRLIB IP Library

The Gaisler Research IP Library (GRLIB) is a library for system-on-chip development, provided by Aeroflex Gaisler Company. It includes various IP cores and AMBA AHB/APB (refer to section 1.4.4), a common on-chip bus. In this project, the IP cores of the LEON3 SPARC processor, 32-bit SRAM controller, 32-bit DDR2 controller and serial debug link are used. These cores are placed around AMBA AHB/APB after they are configured by using Xconfig GUI tool (refer to section 3.1). The library can be used easily with different CAD tools for simulation and synthesis purposes and it has support for various FPGA platforms. The library is provided under the GNU GPL license.

1.4.4 Advanced Microcontroller Bus Architecture (AMBA)

In GRLIB, AMBA 2.0 [2] with AHB/APB is included which is introduced and supported by ARM Limited Corporation. The AMBA is an on-chip bus used in the System-on-Chip (SoC) designs to connect different functional blocks like processors, memories and peripherals. The AMBA Advanced High-Performance Bus (AHB) is a multi-master bus which interconnects the blocks with high data rates like the LEON3 processor and memory unit. Other units like the universal asynchronous receiver/transmitter (UART) which requires low data rates are connected to the system via the AMBA Advanced Peripheral Bus (APB). An APB is connected to an AHB/APB bridge which is the only APB master on the bus. The figure 1.1 is an example of a LEON3 system.

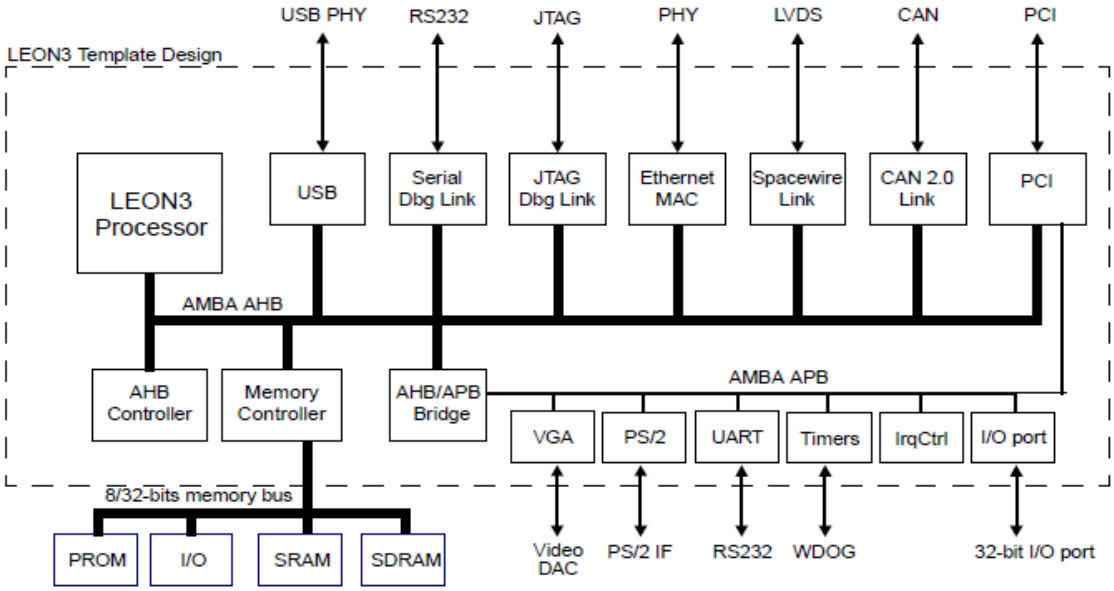


Figure 1.1 Block Diagram of LEON3 System [1]

1.4.5 RTEMS Cross Compilation System (RCC)

Real-Time Executive for Multiprocessor Systems (RTEMS) is a free cross compilation system provided by Aeroflex Gaisler which is used for embedded systems and has been ported to different processor architectures including SPARC. Applications cross-compiled with RTEMS Cross Compiler (RCC) [13] can be debugged on the TSIM LEON simulator [16] and on hardware using GRMON debug monitor [12]. In this project RTEMS 4.10 version is used.

1.4.6 TSIM

During the development process, it is highly important to simulate target applications using simulators before debugging on real hardware to shorten development time. Aeroflex Gaisler provides a licence free, evaluation version of TSIM 2.0 which is capable of simulating LEON processors and it has an important role in the testing of the application after it is cross-compiled. TSIM can be run in stand-alone mode which has various debugging commands to allow the user to change the contents of memory and registers, insert breakpoints and measure performance. Simulation time is measured according to integer unit (IU) and floating point unit (FPU) instruction timing. TSIM can also simulate user defined I/O devices which are loaded as modules written in C

language. TSIM has a simulation performance of more than 1MIPS/100 MHz (host CPU frequency) allowing the user to significantly decrease development process time.

1.4.7 GRMON

GRMON is a debug monitor for LEON processors which is provided by Aeroflex-Gaisler. By using GRMON, applications can be downloaded to a FPGA board and executed, the application process can be profiled for performance testing and moreover all registers and memory are accessible for debug and verification purposes. In this project, an evaluation version of GRMON provided by Aeroflex Gaisler is used as it is a license free version.

1.5 Development Environment

The project has been developed on Windows XP and Linux operating systems. Hardware has been synthesized and downloaded in Windows by using Cygwin [10] which is a Unix-like environment for Microsoft Windows. In Linux environment application software has been coded and compiled.

2 Software

This chapter explains the structure of the softwares that were used in this thesis project.

2.1 MPEG – LAYER III (MP3)

2.1.1 Introduction

MP3 [17] is the most commonly used audio file format. Actually it refers to MPEG-1 or MPEG-2 Audio Layer III, a patented digital audio encoding format which uses a form of lossy data compression and it was designed by the Moving Picture Experts Group (MPEG). MP3 is not license free; it is based on patents from Fraunhofer IIS, University of Hannover, AT&T-Bell Labs, Thomson-Brandt, CCETT and other engineers in the former MPEG group. The standard was finalized in the mid 1990's and MP3 files began to be widely used on the Internet.

The MP3 standard reduces the amount of data required and still sound like uncompressed audio for most listeners. If a 16-bit, 2-channel uncompressed audio file with 44.1 KHz sampling frequency is compressed as a MP3 file at 128kbit/s bit rate, the compression ratio is 11.025. Audio files can also be compressed at higher or lower bit rates, with resulting higher or lower quality MP3 files. There are different sampling frequencies and bit rates defined for MPEG Audio. MPEG-1 defines audio compression at 32 KHz, 44.1 KHz and 48 KHz. The standard also defines a range of bit rates from 8kbit/s to 320kbit/s.

2.1.2 MP3 File Structure

A MP3 file consists of multiple MP3 frames in which a header and data blocks exist [15]. A frame header block has a length of 32 bits (4bytes). The first 11 bits are all set to '1' to for frame sync. Other important information that can be gathered from the header are MPEG Audio version, layer description, bit rate, sampling frequency and channel mode. Data blocks contain compressed audio data in terms of frequencies and amplitudes. Nowadays, MP3 files have also ID3 metadata placed before or after the MP3 frames. An ID3 metadata block has information like artist, title and album and the format which describes the process of including additional information about audio file

is called tag format. However, the MP3 standard does not define tag formats and different MP3 decoders either read information from tags or just treat them as junk data.

2.1.3 Encoding

The MPEG-1 standard does not have an exact specification for encoding, however it gives examples of psychoacoustic models. Implementers of the standard are supposed to design their own algorithms to remove parts of the information from the audio input. This leads to the development of many different encoders each having different performance and quality [18]. A block diagram of a typical MPEG Layer 3 encoder can be seen in figure 2.1. As the project is mainly on implementing of a MP3 decoder, the MP3 encoder is only discussed briefly in the report.

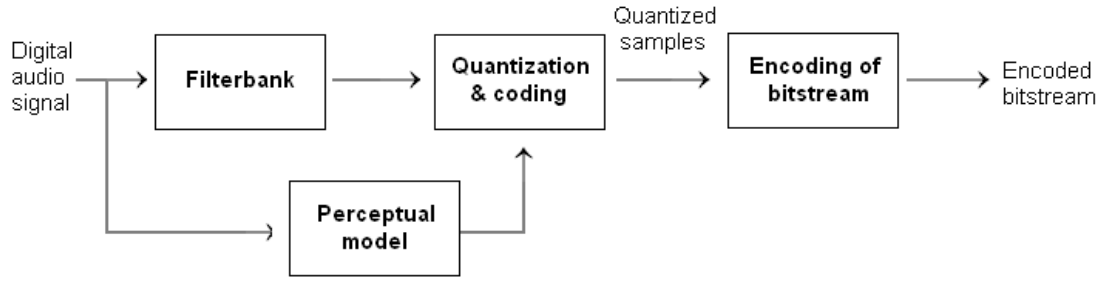


Figure 2.1 A typical MPEG Layer 3 Encoder

2.1.3.1 Filter bank

The filter bank consists of a polyphase filter bank and a Modified Discrete Cosine Transform (MDCT). The polyphase filter bank includes 32 bandpass filters of equal width. Digital audio signals taken as input to the filter bank are in frames of 1152 PCM coded samples. Each frame is divided into two granules of 576 samples. Filter bank converts these 576 samples in the time domain to frequency components with a downsampling ratio of 32. As a result, the filter bank block outputs 18 samples for each of the 32 bandpass filters.

2.1.3.2 Perceptual model

According to psychoacoustics rules, the perceptual model calculates an estimate of the actual masking threshold or allowed noise for each coder partition by using time domain input signal and/or the output of the filter bank. These masking thresholds are used by the quantization and coding block to determine how many bits that are needed to encode

each sample. If the quantization noise is lower than the masking threshold, then the quality of the compressed audio signal is the same as the original audio signal. As a result, the perceptual model is an important block which determines the quality of the encoder.

2.1.3.3 Quantization & coding

This block quantizes and codes the spectral components to keep the quantization noise which is introduced by quantizing below the masking threshold. After quantization, Huffman coding [23] is used to code the quantized values. Because the music signals have different local statistics, different Huffman code tables are used for different parts of the spectrum.

2.1.3.4 Encoding of bitstream

In this block, a bitstream formatter constructs the bitstream which consists of quantized and coded spectral coefficients and some side information e.g. bit allocation information.

2.1.4 Decoding

Compared to MP3 encoding, the MPEG-1 standard defines MP3 decoding more specifically. Most decoders will give the same output within a small rounding tolerance. In further sections, detailed information about MP3 decoding will be given [14].

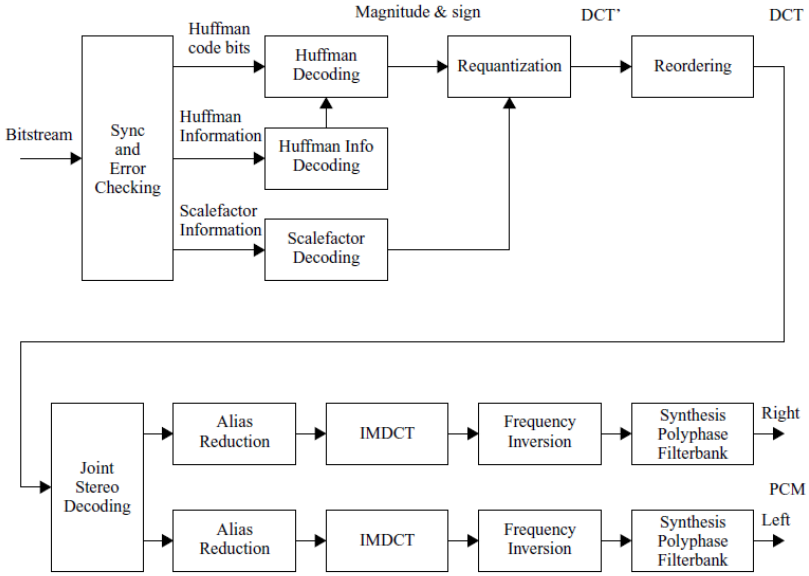


Figure 2.2 MP3 decoder structure [14]

2.1.4.1 Frame data block format

Each frame consists of one or two granules with 576 samples depending on the layer type and the decoder processes one granule at a time. If it is MPEG Layer I, there are two granules and 1152 mono or stereo frequency domain samples and if it is MPEG Layer II, there is only one granule. Each granule is also divided in 32 sub-blocks each having 18 frequency lines. So the length of a frame for a fixed bit rate is usually constant with the exception of a possible deviation of one byte to maintain an exact bit rate.

A frame is composed of a header block and a data block. In the data block, there are side info, main data and ancillary data. The side info section has the information needed to decode the main data such as Huffman table selection, scale factors, requantization parameters and window selection. It is 17 bytes long in single channel and 32 bytes in dual channel mode. The main data section has the coded scale factor values and the Huffman coded frequency lines. For a fixed bit rate, the size depends on the ancillary data section. The ancillary data is not needed to decode audio data, but it includes the specific information between the encoder and the decoder of the same software.

The frequency spectrum is from 0 to $F_s/2$ Hz and this spectrum is divided into 32 equal sub-bands.

2.1.4.2 Huffman decoding

Huffman decoding is used to decode the amplitude of 576 spectral lines. It is expected that low spectral frequencies have large amplitudes while high spectral frequencies have low amplitudes and even zeros. According to this expectation, these 576 spectral lines are located in five regions; *rzero* for "0" values, *count1* for small values between -1 and 1 and three *bigvalue* regions for high values.

2.1.4.3 Requantization

A requantization block is used after the Huffman decoding block to convert the Huffman decoded output to spectral values by using scale factors.

$$xr_i = is_i^{\frac{4}{3}} * 2^{(0.25*C)}$$

EQ 1. Requantization of samples

The equation used in the requantization block can be seen in eq. (1), the C factor in the equation is gathered from side information and the scale factors.

2.1.4.4 Reordering

In the encoding process, short windows are used to make Huffman coding process more efficient assuming that samples with closer frequencies have similar values. Therefore after the samples are requantized, they must be reordered for the scale factor bands, in other words frequency lines are sorted by subbands, then by frequency. This block is used only for short blocks, not for long blocks which are only sorted by frequency. Short blocks have better time resolution than long blocks and are used for transients where there is large difference between consecutive frames.

2.1.4.5 Alias reduction

An alias reduction block includes eight butterfly calculations for each sub-band and is used to reduce the aliasing effects of the polyphase filter bank in the encoding process.

2.1.4.6 IMDCT

The IMDCT (Inverse Modified Discrete Cosine Transform) is used to transform the sub-band samples from frequency domain to time domain.

$$x_i = \sum_{k=0}^{\frac{n}{2}-1} X_k \cos\left[\frac{\pi}{2n}\left[2i+1+\frac{n}{2}\right](2k+1)\right], \text{ for i=0 to n-1}$$

EQ 2. IMDCT Transform

In the equation, the n value is 12 for short blocks and 36 for long blocks which is received from the side info in the data block of each frame. If it is a short block, three transforms are realized for every 6 input values and three vectors of 12 output values are overlapped with each other. Then 6 zeros are added to the both ends of the vector of 24 output values resulting from overlapping. As a result, a vector of 36 output values is generated. If it is a long block, an output of 36 values is generated for every 18 input values. The first half of the current block and second half of the previously saved block are overlapped. The overlapped addition of the long blocks is shown in the following figure:

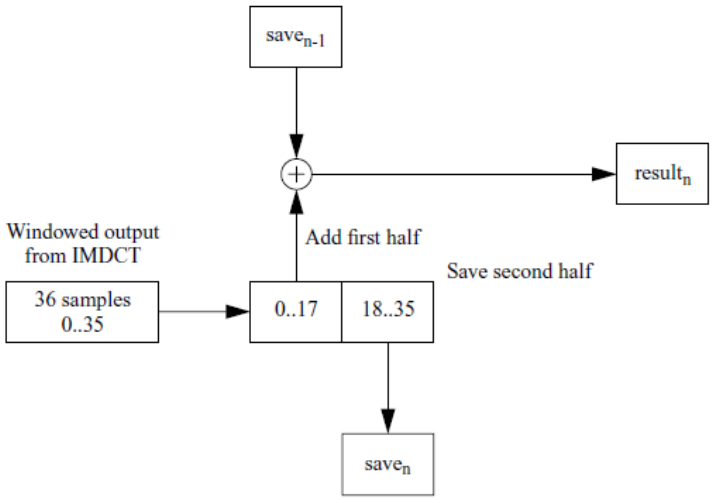


Figure 2.3 Overlapped addition operation of the long blocks

As a result, the IMDCT block outputs 18 time-domain samples for each of the 32 sub-band blocks. After the IMDCT block, a 36 point windowing function is used to smoothen frame transition. If there is little difference between consecutive frames, a long window for 36 values is used. If there is large difference, a short window is used to for better time resolution.

2.1.4.7 Frequency inversion

The frequency inversion block multiplies every odd time sample of every odd sub-band with -1 to correct for the effect of frequency inversions in the synthesis polyphase filter bank.

2.1.4.8 Synthesis polyphase filter bank

The synthesis polyphase filter bank uses 32 samples, one from each sub-band block of 18 time-domain samples in each granule and converts it to 18 blocks of 32 PCM samples.

2.2 MPG123 Library

After a research among different MPEG Audio player and decoder libraries, the MPG123 library with version 1.13.2 [15] is selected for implementation as it provides non-floating point execution, compatibility with the RTEMS compiler and is a license-free, fast real-time library.

The MPG123 library provides various features like HTTP support which enables getting MP3 audio files from a WWW server, outputting different audio file formats like WAV, SUN audio and CDR file instead of playing the audio file, down sampling, playlist and equalizer, however many properties are excluded due to the small RAM size of the FPGA board. Before RTEMS cross-compilation, the MPG123 application is tested in Linux. In figure 2.4, it is observed that only 680 KB memory and max 1.3% of the 1.86GHz CPU are used.

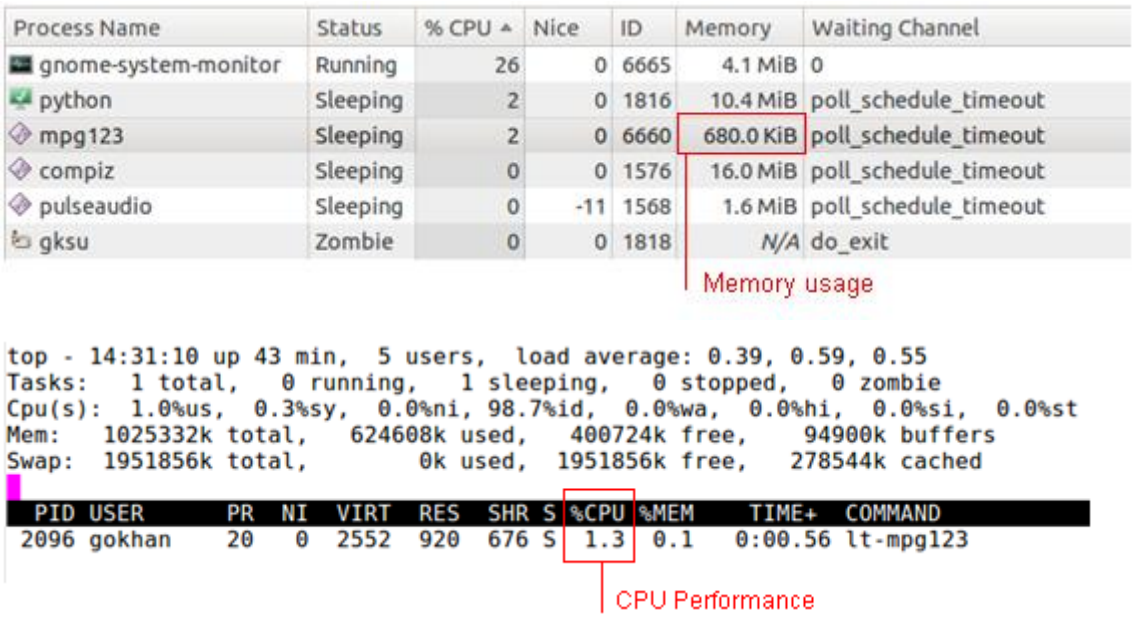


Figure 2.4 MPG123 system performance in Linux

The CPU frequency needed is calculated as 24 MHz and the working frequency of the LEON3 processor on Spartan3 is 40 MHz which is generated from the 50 MHz main clock to meet the timing constraint of the design. It is expected that the MPG123 application can run on the LEON3 processor. However, when memory capacity is considered, the FPGA board has 1 MB capacity whereas the MPG123 application is 700 KB and uses 680 KB of memory. If unnecessary parts of the application are removed, it is expected that the application can run with 1 MB RAM.

RTEMS is chosen for cross-compilation of the MPG123 application source code because it is provided by Gaisler and is supported by the MPG123 library with configuration options. To use the RTEMS cross-compiler, the source files are downloaded and added to the path.

export PATH=/opt/rtems-4.10/bin:$PATH

Then, the RTEMS options *-msoft-float* for no floating point operation, *-g* for debugging with gdb and *mcpu=v8* for generating SPARC V8 mul/div instructions are changed in the configure file of the library by the following commands and with the *make* command the application is cross-compiled.

export CC=sparc-rtems-gcc
export CXX=sparc-rtems-gcc
export CFLAGS='-msoft-float -g -mcpu=v8'
./configure --target=sparc-rtems --host=sparc-rtems --disable-shared
--with-cpu=generic_nofpu
make

After that, a few changes are done to use the application with the LEON3 processor. When it is used in a Linux terminal, the application uses the *getopt* library to get options and filename written in command line in the terminal. However, GRMON does not accept additional commands, but only the run command. So, the following modifications in *mpg123.c* and *compat.c* are done to have the application select a user-located MP3 file.

Change in mpg123.c;

```
while ((fname = "test.mp3")) => while ((fname = get_next_file()))
```

Instead of getting files with the *get_next_file* function which uses the *getopt* library, the filename is given as *test.mp3* and stored in the "fname" char string. The "fname" char string is used as "filename" char string in compat.c and it is located in *compat.c* by adding following lines,

```
int *ptr;
int yuk;

ptr=0x400d0000;
ret =creat(filename ,S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
yuk=write(ret,ptr,40000);
close(ret);
```

Before the following line in compat.c;

```
ret=open(filename,flags,S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH);
```

After the file is opened in compat.c, MP3 data is loaded from the file descriptor (fd) by the mpg123_handle. Instead of changing the mpg123_handle, a pointer is defined starting from the address of 0x400d0000 and a file is created just before the file is opened by the application. The address range of the RAM is from 0x40000000 to 0x40100000 which corresponds to 1 Mbyte. Because the application is loaded from

0x40000000 to 0x400b0000, MP3 data is loaded to 0x400d0000 which is the free part of the RAM. To load the MP3 file into the LEON3 processor, it is changed to SREC format by using the BIN2SREC program which is described in the section 2.3.

Before testing the application on the FPGA board with GRMON, it is tested with TSIM which simulates the LEON3 processor successfully, loads files and runs faster. It makes us find software errors before moving to FPGA; as a result it decreases the development time significantly. The simulation report for the application on TSIM is as follows.

```
tsim> load test.srec
section: HDR at 0x400d0000, size 40818 bytes
entry point: 0x400d0000
tsim> load mpg123.exe
section: .text, addr: 0x40000000, size 703408 bytes
section: .data, addr: 0x400abbb0, size 17328 bytes
section: .jcr, addr: 0x400aff60, size 4 bytes
read 1957 symbols
tsim> run
starting at 0x40000000
High Performance MPEG 1.0/2.0/2.5 Audio Player for Layers 1, 2 and 3
        version 1.13.2; written and copyright by Michael Hipp and others
        free software (LGPL/GPL) without any warranty but with best wishes
Playing MPEG stream 0 of 0: test.mp3 ...

Title:   Exponential Sweep 16Hz-20kHz, 1/f^2 power spectrum
MPEG 1.0 layer III, 64 kbit/s, 44100 Hz mono
0:00] Decoding of test.mp3 finished.
```

Figure 2.5 TSIM simulation result of modified MPG123 sofware

As it can be seen in figure 2.5, the application is run on TSIM successfully and ends without any errors or traps. By simulating on TSIM, the functionality of the application is verified. In other words it can be run on the LEON3 processor without any software errors. After that the application is loaded into the LEON3 processor and it also works successfully on the GRMON as expected. Further modification before getting the application to run is done to minimize its size. *Http, playlist, streamdump, wav, control-generic header* and *C* files which do not affect program functionality and lines for parameters that are not used are removed.

2.3 BIN2SREC

Elf-sparc [13] and s-record [20] are the only supported file formats when using GRMON to load files into the LEON3 processor's memory. Executable and linkable format for SPARC architecture (ELF-Sparc) is the file format for executable files

created by using cross-compilers. As MP3 files are binary data files, which cannot be compiled with cross-compilers, changing MP3 to s-record format is the only option to load in the memory. SREC (s-record) is a hexadecimal text encoding for binary data which was created for the Motorola 6800 processor.

The SREC format has the following structure;

Start Code (S) + Record Type + Byte Count + Address + Data + Checksum

```
S3 15 40000000 4944330200000000003A545432000034 A0
```

S3 : *Record type*: Data sequence with 4 address bytes

15 : *Byte count*

40000000 : *Address*

A0: *Checksum*

The BIN2SREC library [19] is used to convert the MP3 file to a s-record file with the following options;

```
./bin2srec -o 400d0000 -a 4 -l 16 test.mp3 >test.srec
```

GRMON needs the SREC files to have the address in 4 byte and 16 byte data in a line. The data start address is selected to be after the last program address and in the range of memory capacity.

3 Hardware

This chapter explains the LEON3 processor structure and the hardware implementations that were made in this thesis project. In addition hardware-software co-simulation and new development platform are mentioned.

3.1 LEON3 Processor

The starting point of the thesis project is implementation of the LEON3 system using GRLIB. The implementation is typically done in three steps which are configuration of the design using xconfig GUI tool, simulation of the design and synthesis and place&route. The template design of the LEON3 processor is available in GRLIB and based on three files : config.vhd, leon3mp.vhd and testbench.vhd. The config.vhd file is containing the configurations that are given from the xconfig GUI tool and the leon3mp.vhd is the top module of the leon3 processor. The testbench.vhd is a testbench with external memory and emulating the FPGA board.

Before synthesis and place&route, the LEON3 processor needs to be configured according to the requirements of the thesis project and the leon3mp.vhd file needs to be modified according to the configurations. The configuration of the LEON3 processor is done with the xconfig GUI tool. The xconfig GUI tool is launched by the command "make xconfig" in the cygwin shell(windows). The GUI window is popping up like in the figure below.

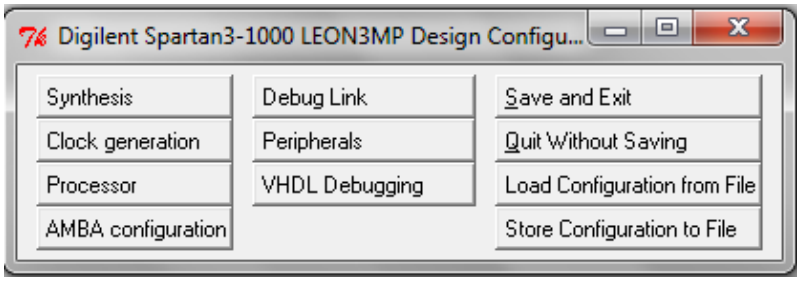


Figure 3.1 Xconfig GUI Window

From the xconfig GUI tool, the options need to be configured according to the requirements of the thesis project. In the synthesis part the target technology is selected as Xilinx-Spartan3 and other options are all disabled. In the Clock generation part, the clock generator is selected as Xilinx-DCM and the clock multiply is selected as 4 and

the clock divider is selected as 5 which are the default values. The processor clock which is 40 MHz is generated from the 50 MHz main clock by multiplying 4 and dividing by 5 to satisfy the timing constraint of the design. In the Processor options, the number of processors is selected as 1 which is the LEON3 SPARC V8 processor. The options for the integer unit is left as default. The floating point unit is disabled, cache systems is left as default, MMU and Debug Support Unit is enabled. The AMBA configuration option is also left as default. For Debug Link option there is just one alternative which is JTAG Debug link but in the thesis project a serial connection is used for debugging so changes are made in the config.vhd and leon3mp.vhd file for using serial connection. The RS232 standard[11] is used for the serial connection between the computer and the FPGA board. In the leon3mp.vhd file the AHB uart connection part is modified. The *ahbjtag0* component is deleted and only the component *dcom0* is used. With this small modification serial connection with the board is achieved. In the Peripherals part, Memory Controller, On-chip Ram/ROM, UART options are left as default and the keyboard option is disabled. After the settings are made the configurations are saved. After the configuration of the LEON3 processor, pin assignment in the user constraint file (UCF) of the design is arranged. Synthesis and place&route is made in Xilinx ISE[6]. Finally the programming file is generated and loaded to the FPGA board.

In section 1.4.7 information about the debugging monitor GRMON is given. GRMON can be used for read/write access to all system registers and memory downloading and execution of LEON3 applications. It has built-in disassembler and trace buffer management. The applications that are used in this thesis project are loaded with GRMON using serial communication. In the windows command shell, GRMON can be started with command line options. The most important options for this thesis project are “-baud”, “-u”,”-uart com#” and “-nb”. The first option “-baud” is used for setting the baud rate for the DSU serial link. In this thesis project the baud used is 115200 which is the default baud rate value. The option “-u” is putting UART in FIFO debug mode and it is enabling both reading and writing to the UART from the monitor console. In addition “-uart com#” is used for communicating with the target using a specific port of the host. By default GRMON tries to communicate with the first uart port of the host. Finally the option “-nb” is used for telling the debugger not to stop on

error traps. After starting GRMON with these options, GRMON tries to connect with the target and when it achieves the connection then it scans the system to detect which IP cores are present.

This is done by reading the plug&play information which is normally located at address 0xfffff000 on the AHB bus. A debug driver for each recognized IP core is then initialized, and performs a core-specific initialization sequence if required. For a memory controller, the initialization sequence would typically consist of a memory probe operation to detect the amount of attached RAM. After the initialization is complete, the system configuration is printed:

```
C:\Users\caca\Desktop\win32>grmon-eval.exe -u -nb -nosram -uart com5

 GRMON LEON debug monitor v1.1.49 evaluation version

 Copyright (C) 2004-2011 Aeroflex Gaisler - all rights reserved.
 For latest updates, go to http://www.gaisler.com/
 Comments or bug-reports to support@gaisler.com

 This evaluation version will expire on 20/1/2012
try open device //./com5
###opened device //./com5

 Device ID: : 0x601
 GRLIB build version: 4108

 initialising ...........
 detected frequency:  50 MHz
 SRAM waitstates: 1

 Component                            Vendor
 LEON3 SPARC V8 Processor             Gaisler Research
 AHB Debug UART                       Gaisler Research
 GR Ethernet MAC                      Gaisler Research
 AHB/APB Bridge                       Gaisler Research
 LEON3 Debug Support Unit             Gaisler Research
 Xilinx MIG DDR2 controller           Gaisler Research
 LEON2 Memory Controller              European Space Agency
 Generic APB UART                     Gaisler Research
 Multi-processor Interrupt Ctrl       Gaisler Research
 Modular Timer Unit                   Gaisler Research
 General purpose I/O port             Gaisler Research

 Use command 'info sys' to print a detailed report of attached cores

grlib>
```

Figure 3.2 GRMON Initialization on Command Window

After the connection more detailed system information can be printed and the attached IP cores can be seen by the command “info sys”.

```
grlib> info sys
00.01:003   Gaisler Research  LEON3 SPARC V8 Processor (ver 0x0)
             ahb master 0
01.01:007   Gaisler Research  AHB Debug UART (ver 0x0)
             ahb master 1
             apb: 80000700 - 80000800
             baud rate 115200, ahb frequency 50.00
02.01:01d   Gaisler Research  GR Ethernet MAC (ver 0x0)
             ahb master 2, irq 12
             apb: 80000f00 - 80001000
             Device index: dev0
01.01:006   Gaisler Research  AHB/APB Bridge (ver 0x0)
             ahb: 80000000 - 80100000
02.01:004   Gaisler Research  LEON3 Debug Support Unit (ver 0x1)
             ahb: 90000000 - a0000000
             AHB trace 128 lines, 32-bit bus, stack pointer 0x47fffff0
             CPU#0 win 8, itrace 128, V8 mul/div, srmmu, lddel 1
                   icache 1 * 8 kbyte, 32 byte/line
                   dcache 1 * 4 kbyte, 16 byte/line
04.01:06b   Gaisler Research  Xilinx MIG DDR2 controller (ver 0x0)
             ahb: 40000000 - 48000000
             apb: 80000500 - 80000600
             DDR2: 128 Mbyte
05.04:00f   European Space Agency  LEON2 Memory Controller (ver 0x1)
             ahb: 00000000 - 20000000
             ahb: 20000000 - 40000000
             apb: 80000000 - 80000100
             8-bit prom @ 0x00000000
01.01:00c   Gaisler Research  Generic APB UART (ver 0x1)
             irq 2
             apb: 80000100 - 80000200
             baud rate 38343, DSU mode (FIFO debug)
02.01:00d   Gaisler Research  Multi-processor Interrupt Ctrl (ver 0x3)
             apb: 80000200 - 80000300
03.01:011   Gaisler Research  Modular Timer Unit (ver 0x0)
             irq 8
             apb: 80000300 - 80000400
             8-bit scaler, 2 * 32-bit timers, divisor 50
0b.01:01a   Gaisler Research  General purpose I/O port (ver 0x1)
             apb: 80000b00 - 80000c00
```

Figure 3.3 Info sys Result on Command Window

As can be seen from the detailed system information, the memory address is starting at 0x40000000 and when there is a load of any application, which does not contain any specific address information, loading starts from the initial adress of the memory by default. Before an application is loaded, cross-compilation of the source code of the application is needed because the sparc processor only runs sparc binaries. For the cross-compilation, the RTEMS LEON/ERC32 Cross-Compiler System is used. In section 1.4.5 general information about the RTEMS Cross-Compiler System is given and according to the information and the user's manual, the compilations of the source codes of the applications are made. During the cross-compilation, some useful *sparc-rtems-gcc* compiler options are used. The first option is "*-msoft-float*" which must be used if there is no FPU in the system. This option is emulating floating point. The other option used during the cross-compilation is "*-mcpu=v8*". This option generates SPARC V8 mul/div instructions for LEON with hardware multiply and divide configured system. Moreover option "*O2*" is used to optimize the code. This option should be used for optimum performance and minimal code size. After the cross-compilation options

are selected, the cross-compilation of the test C code is done. The test code and the command line to compile the code are shown below.

```
main ()
{
     printf("\n\nHELLO!!\n");
     return 0;
}
```

sparc-rtems-gcc –msoft-float –mcpu=v8 -O2 hello.c -o hello.exe

After the compilation an "*exe*" format file is generated and ready for loading to the system. The loading process is done using GRMON. In the GRMON console the command "*load hello.exe*" is written and the application is loaded to the board. After load a "*run*" command runs the code and the "HELLO" word is written on console and this verifies that the processor is running correctly and is ready to be used for the audio core and player applications.

3.2 Audio Core

3.2.1 Introduction

In order to use the decoded values from the MP3 player, an audio core is implemented. The audio core that is used in this project addresses a circuit board which contains a digital to analog converter. The converter is followed by a smoothing circuit which is removing the clock from the output signal. The circuit board is connected to one of the expansion connectors on the Xilinx Spartan 3 board. The digital to analog converter on the circuit board is MCP4822 from the vendor Microchip. MCP4822 is a dual channel 12-bit DAC which has Serial Peripheral Interface (SPI™) supporting 20 MHz clock frequency. In order to communicate with the DAC, a hardware interface needs to be implemented in VHDL according to the specifications of the DAC and needs to be attached on the LEON3 processors AMBA/APB Bus as a slave. In the following figure a simple block schematic of the system is given.

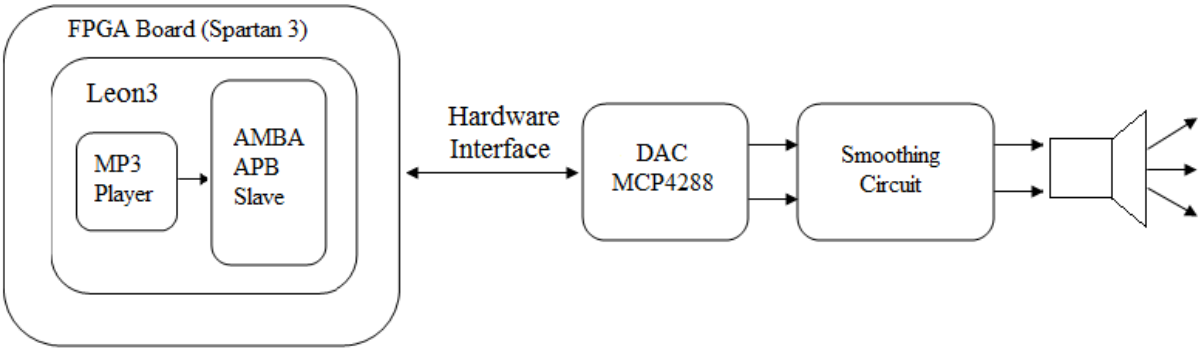


Figure 3.4 Block Schematic of the Entire System

3.2.2 Implementation

3.2.2.1 Hardware Interface Design

According to the datasheet of the MCP4288 digital to analog converter, the device is designed to interface directly with the SPI port of the microcontroller. The converter has 8 pins which are Vdd, CS, SCK, SDI, $Vout_A$,$Vout_B$, AV_{SS} and LDAC.

Communication with the DAC is unidirectional so data cannot be read at the output of the device. The configuration and data bits are sent to the device via the SDI pin, where data is being clocked-in at the rising edge of the SCK. The clock data which is sent to the SCK pin needs to be at most 20 MHz. The CS pin needs to be low during the write command. The write command of the device is a 16-bit word and the four most significant bits of the 16-bit word are used for configuring the device and the other 12 bits are data . The four most significant bits of the 16-bit word are used for selecting DAC_A or DAC_B, don't care, output gain selection and output power down control respectively. After each write command, the LDAC pin needs to be low for transfering the input latch registers to the output latches. In the following figure the timing diagram for a write command is shown.

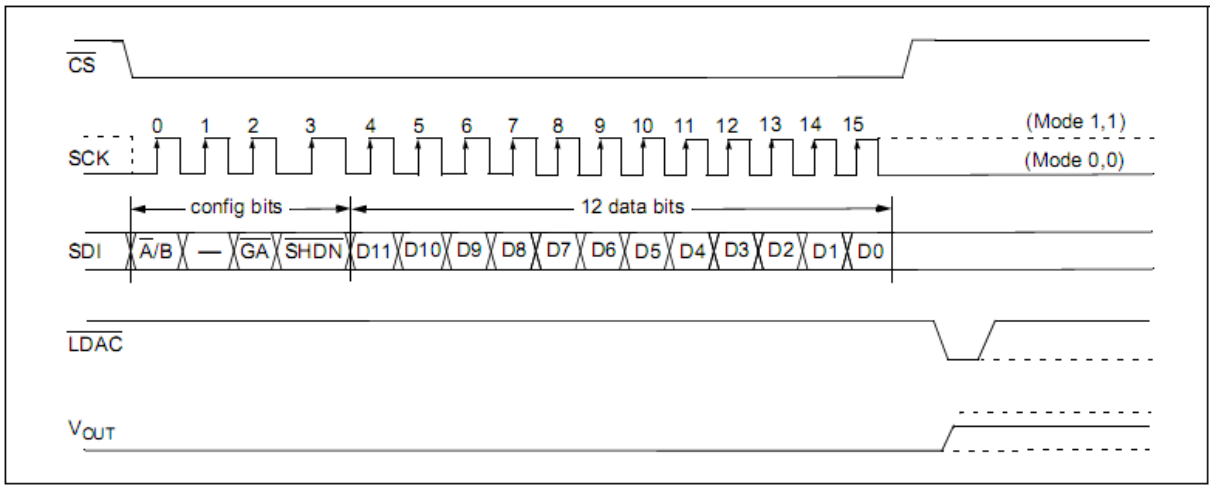


Figure 3.5 Timing Diagram for Write Command

With the guidance of the information from the data sheet of the DAC, VHDL code for the interface is written and simulated in Modelsim[5]. The interface has three inputs which are 12 bit data, main clock and reset. In addition to that there is four outputs which are SCK, LDAC, CS and DOUT. The main clock of the LEON3 processor is 40 MHz and in the implementation it is used as SCK for the DAC and so it needs to be slowed down to a frequency higher than 16x44 kHz to send all data bits and generate the analog value at 44 kHz. When reset is high, new 12 bit datas are taken and the CS pin goes low. After 16 clock cycles the CS pin goes high and the LDAC pin goes low for a short time. As the device triggers on the positive edge on the SCK signal, the bits that are sent have to be placed on DOUT before the positive edge so the data are stable when the positive edge comes.

3.2.2.2 Simulation and Testing

After the implementation of the VHDL code, the design is simulated in Modelsim for debugging and then it is ready for testing. During the testing the inputs are given from the switches of the FPGA board which are 8-bits. The remaining 4-bits are set to ‘0’. The VHDL code is synthesized using Xilinx ISE and a programming file is generated. The generated file is loaded to the FPGA. The output of the circuit board is connected to an oscilloscope and for each 8-bit wide digital input value, an analog value is seen on the oscilloscope. With that stage the testing of the implemented code is done and next step is attaching the audio core to the AMBA bus of the LEON3 processor.

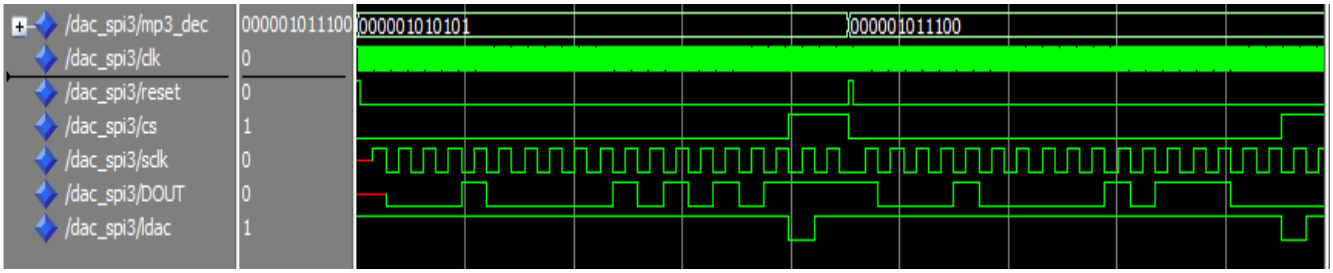


Figure 3.6 Simulation output of Hardware Interface

3.2.2.3 Bus Selection

The LEON3 processor is using AMBA AHB and APB for connection to the memory and peripherals. The LEON3 processor needs to be connected to the audio core through AHB or APB. The first thing to be done is selecting one of these buses.

AMBA AHB is used for high clock frequency and high performance system modules. It is suitable to use as the main system bus. On the other hand APB is suitable for low power peripherals and the interface complexity is reduced. Using the audio core as a master on AHB can increase the bus loading and may decrease the performance of the processor. According to the AMBA specifications the APB interface is recommended for simple register-mapped slave devices. Because of these facts, attaching the audio core to the AMBA APB as a slave is decided.

3.2.2.4 Attaching the Audio Core to AMBA/APB as a Slave

In the section above, selection of the bus is made and using APB for communicating with LEON3 processor is decided. To attach the audio core to the APB as a slave, an interface implementation in VHDL is needed.

Before the implementation of the interface, understanding of the specifications of the APB is important. The APB has eight signals and each name starts with the single letter 'P' indicating that they are APB signals. The signals of the APB are:

PCLK	Bus clock and rising edge is used for all transfers on APB.
PRESETn	APB bus reset signal is active LOW and can be connected to system reset.
PADRR[31:0]	APB adress bus with 32-bits wide and driven by peripheral bus bridge unit.
PSELx	Indicates that the slave device is selected and transfer of data required. For each slave unit there is a PSELx signal.

Table 1. Signals of APB Bus

PENABLE	Used to time all accesses on the peripheral bus. It is used to indicating the second cycle of an APB transfer. In the middle of the APB transfer the rising edge of the PENABLE signal occurs.
PWRITE	Indicates the direction of the transfer. When HIGH, there is an APB write access and when LOW there is an APB read access
PRDATA	When PWRITE is LOW, read data bus is driven and can be up to 32-bits wide.
PWDATA	When PWRITE is HIGH, write data bus is driven and can be up to 32-bits wide.

Table 1. Signals of APB Bus (cont)

According to the AMBA/APB specifications, the APB has three states which are IDLE, SETUP and ENABLE as in figure 3.7. The IDLE state is a default state for the peripheral bus and PSELx and PENABLE signals are '0' at that time. When a transfer is required, the bus is moving to SETUP state and the select signal PSELx becomes '1'. The bus only stays in that state for one clock cycle and always moves to the ENABLE state on the rising edge of the clock.

In the ENABLE state the PSELx and PENABLE signals are asserted. This state only lasts for one clock cycle and the bus returns to IDLE state if there is no more transfers required. On the other hand if there is another transfer required the bus is returning to the SETUP state.

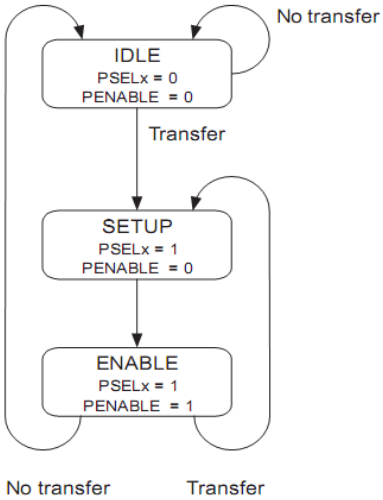


Figure 3.7 State Machine for APB Bus

There are two transfers that can be done on APB and they are write and read transfers. In the write transfer, after the rising edge of the clock, adress, write data, write signal and select signal are changing. The first clock cycle is the SETUP cycle and at the

following clock edge the PENABLE signal is asserted and the ENABLE cycle is taking place. The adress, data and control signals all remains valid in the ENABLE cycle and the write transfer is completed at the end of this cycle. At the end of the write transfer the PENABLE signal is deasserted and the select signal goes LOW except when there is another transfer to the peripheral. For the read transfer the timing of write, adress, select and strobe signals are the same as in the write transfer. The slave must provide the data during the ENABLE cycle and at the end of the ENABLE cycle the data must be sampled on the rising edge of the clock. For the write transfer the data is read from PWDATA signal and for read transfer the data is read from PRDATA signal. In the figure 3.8, the timing diagram for write and read transfers are shown. Between T1-T2 and T4-T5 the IDLE state takes place and between T2-T3 the SETUP state takes place and between T3-T4 the ENABLE state takes place.

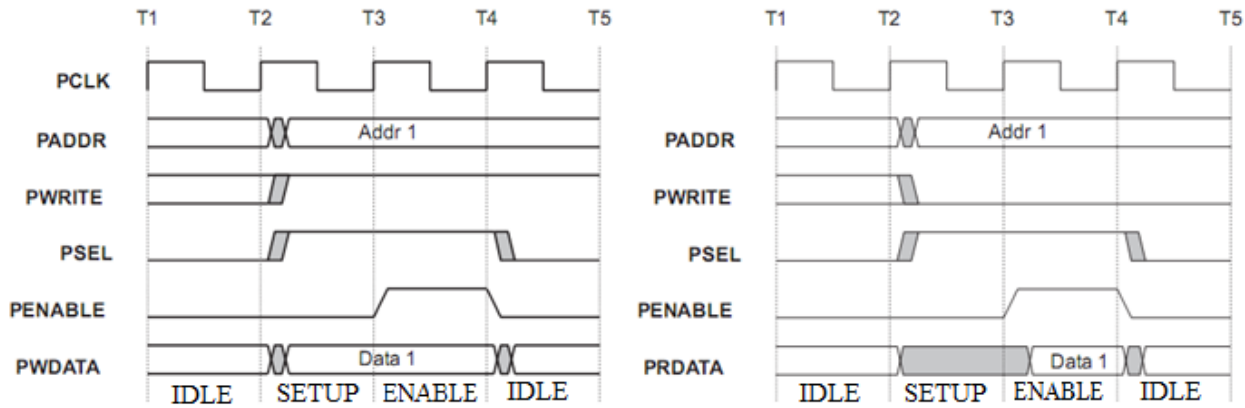


Figure 3.8 Timing diagram for write and read transfers respectively

One of the reasons for using APB is that it has a simple slave interface and in addition to that it is flexible with many possible options. In the case of write transfer, the data can be latched on the rising edge of the PCLK, when PSEL is HIGH or on the rising edge of PENABLE when PSEL is HIGH.

The APB slave interface that is implemented is written according to GRLIB and it has three inputs which are clock, reset and APBI. The APB master is driving a set of signals grouped into a VHDL record called APBI. This APBI is sent to all APB slaves. The bus multiplexer and adress decoder is controlling which slave that is going to be used. The input record APBI includes the select signals for all slaves in the vector APBI.PSEL and therefore the APB slave must use a generic called PINDEX ,which is of type integer, for

specifiying which PSEL element to use. Beside the inputs, the slave interface has 5 outputs which are APBO and outputs for the audio core which are described in the hardware interface design section. The APBO is the output record of the active APB slave and it is forwarded to AHB slave output. In the APB slave interface, the hardware interface that is implemented to communicate with the audio core is used as a component. When apbi.psel(pindex), apbi.penable and apbi.pwrite are HIGH the input data is taken from apbi.pwdata and sent to the component. The apbi.pwdata is the data coming from the APB and it is 32 bits. For the DAC device, only 12 bits of the apbi.pwdata can be used and the decision of using the LSBs or MSBs of the data needs to be taken according to the music data. This will be discussed in the further sections of this chapter. Besides the data coming from the bus, a spi_reset signal is sent to the hardware interface for setting the new coming data values and other instances of the interface. In the write transfer of the APB slave, the spi_reset signal becomes HIGH and LOW for other cases. When the spi_reset is LOW, the hardware interface starts sending the values according to the timings and specifications that are mentioned in the hardware interface design chapter. For the read transfer, an output named ‘flag’ is added to the hardware interface and after the whole 16 bit data is sent to the DAC, the flag signal becomes ‘1’ which is indicating that the hardware can get new values from the APB bus and it becomes ‘0’, when a new 16 bit data set comes. The flag signal is then used in the APB slave interface. When apbi.psel(pindex) is HIGH, then the flag is sent to the register to let the software know that new data can be sent. If the flag is LOW then the software is not sending any values to the bus. A detailed information about the usage of flag will be given in the futher parts of the chapter.

The slave interface and hardware interface are added to the opencores library. In the leon3mp.vhd which is the top module of the leon3 processor, the APB slave interface is added as a component and the signals are connected as in the VHDL code below.

```
mp3_hardware1 : mp3_amba_interface
    generic map (pindex  => 9, paddr => 9, pmask => 16#FFF#)
    port map (rst => rstn, clk => clkm, apbi => apbi, apbo => apbo(9),
  dac_clk2=> dac_clk_fin, dac_in =>dac_in_fin,dac_ldac=> dac_ldac_fin,
  dac_cs=> dac_cs_fin );
```

The pindex is 9 and it's telling that when the APB want to reach the audio core it is going to select the PSEL's 9th element. There are 100 hexadecimal addresses per device. The address of the AMBA APB is 0x80000000 and the audio core is placed in the address 0x80000900.

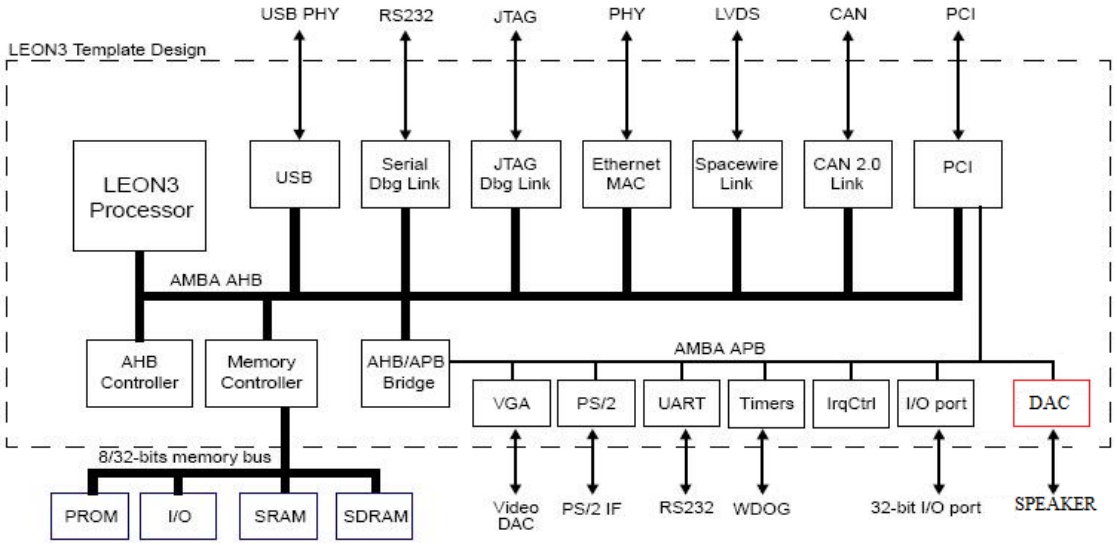


Figure 3.9 LEON3 Processor System with Audio Core Attached

3.2.3 Embedded System Simulation and Testing

The audio core is attached to AMBA APB as a slave and testing needs to be done before using the MP3 player. First a simple test program is used for testing the communication between the LEON3 processor and the audio core and then software is used for playing a WAVE file. After those tests the MPG123.exe program is used for playing MP3 files.

3.2.3.1 Simple Test Program

This program is sending integer values between 1 to 255. At the output of the audio core the analog values according to these integer values need to be seen if everything works correctly.

In the implementation of the APB slave interface, it is mentioned that small changes in the interface are needed according to the input values. The DAC in the audio core is just using 12 bits as data and the values coming from the APB bus are 32 bits. The least significant 12 bits of the 32 bit data needs to be used to get the values correctly.

The simple test program is sending values between 1-255 which corresponds to 8 bits which are the most significant bits of 12 bit data. The other 4 bits are assumed '0'.

According to the input value little changes are done in the slave interface design. For example the spi_in is the input values which are sent to the hardware interface and the apbi.pwdata is 32 bits and just 8 bits of the apbi.pwdata is used for getting the values.

After these changes are made in the APB slave interface, the software program is written in C programming language. The C code of the test program is as below:

```
struct data_regs_t {
  volatile int flag_send_data;
};
struct data_regs_t  * data_regs = (struct data_regs_t  *)0x80000900;
main()
 {
    int i;
    i=1;
  while(1) {
    if(i<255){
     data_regs->flag_send_data=i;
     i++;
            }
    else

        i=1;

 }
}
```

Figure 3.10 C code implementation of the Test Program

In the code above, flag_send_data is declared as volatile integer because it is a memory mapped peripheral register which can be changed asynchronously to the program flow. If it is not declared volatile, during optimized compilation, it is compiled as not changeable by peripheral.

Volatile integer flag_send_data is used for sending the value of the integer i which is changing from 1 to 255 to the address of the register. The address is 0x80000900 where the audio core is attached. The AMBA APB is writing the values to apbi.pwdata during the write process and the hardware interface is just taking the least significant 8 bits of the 32 bit data and sending them to the audio core. The output of the audio core is changing according to the integer values and the analog values can be seen. This simple test program verifies that the communication between the LEON3 processor and audio core is done correctly. After the test stage, the project can be taken to the next step where a wave file is played.

3.2.3.2 Simple Wave Player

After testing the communication between the LEON3 processor and the audio core, the next step is to play a wave file. First loading of a wave file to the memory is needed. For loading the wave file to memory, the BIN2SREC program, which is converting binary files to Srecord format, is used. In chapter 2.3 information about the Srecord format is

given and according to the information, the generated Srec file is changed. The starting adress of the data is selected as 0x40010000 and the wave player is getting values starting from that address. Before writing the C code of the wave player, some research is done about the wave file format.

3.2.3.2.1 Wave File Format

The waveform audio file format which is a Microsoft standard is used for storage of multimedia files. It is a subset of the RIFF bitstream format [7] which starts with a header followed by data chunks. The wave file contains a RIFF chunk descriptor and 2 other sub-chunks which are fmt and data chunk. The fmt sub-chunk[8] is containing information about the format of the sound and the data sub-chunk is indicating the size of the sound information and contains the raw sound data. The general name for this format is Canonical Wave file format and the figure below shows the structure of this file format.

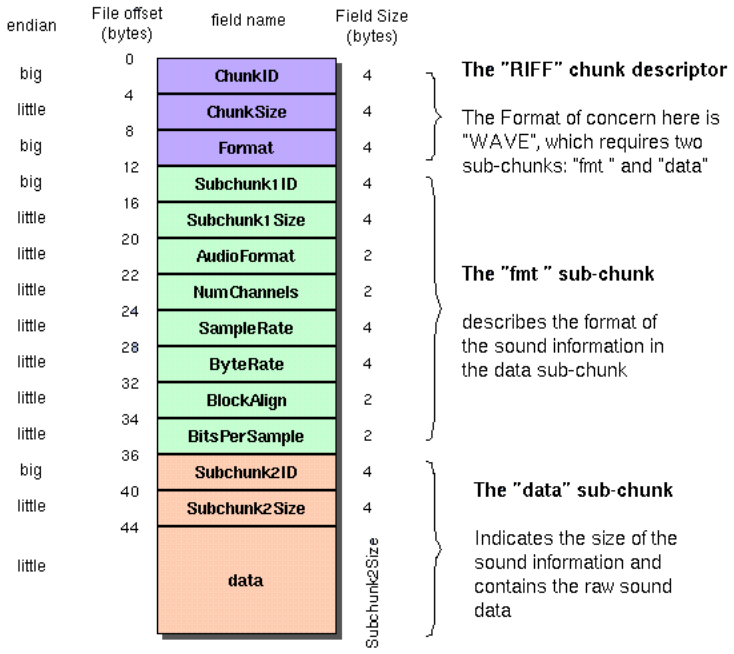


Figure 3.11 Structure of the Wave File Format [8]

The first 4 bytes contains the letter RIFF in ASCI form, after that the 4 bytes are chunk size which indicates the size of the rest of the chunk. 8th byte to 11th byte contains the letters WAVE in ASCI form. From 12th to 15th the letters “fmt “ are contained and from 16th to 19th the size of the subchunk is contained. The 20th and 21st bytes are containing information about the form of compression and if that value is 1 than it indicates that the audio is PCM. 22nd and 23rd bytes are indicating the number of channels like mono

or stereo. From byte 24 to 35 sample rate, byte rate, block align and bits per sample informations are contained respectively. From the 36th byte the data sub-chunk starts and it contains the size of the data and actual sound data. The default byte ordering for a wave file is little-endian and 16 bit samples are stored as 2's complement signed integers which are ranging between -32768 and 32767. Before continuing with the C code implementation understanding of the little-endian byte ordering is important.

3.2.3.2.2 Little endian byte ordering

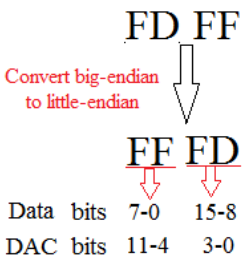
According to the wave format specifications, the default byte ordering is little-endian. In little endian byte ordering the least significant byte is stored in lowest address and the most significant byte is stored in the highest address. For example if there is a 32 bit quantity written as $FAFBFCFD_{16}$. Each hexadecimal represents 4 bits and there is a need of 8 hexadecimal digits. The bytes are FA, FB, FC and FD. The adresses to store these bytes are adr0,adr1,adr2 and adr3. According to little-endian byte ordering FA byte needs to be stored in adr3, FB byte is stored in adr2, FC byte is stored in adr1 and FD byte is stored in adr0.

The LEON3 processor is using SPARC V8 architecture and the SPARC is a big-endian architecture which means the most significant byte is stored in the lowest address and the least significant byte is stored at highest adress. Because of the endianess of the SPARC architecture changes are made in the AMBA APB slave interface implementation which will be explained in the next section.

3.2.3.2.3 Changes in APB Slave Interface for playing wave file

After examining the characteristics of wave files, changes are made in the APB Slave interface. The input wave sound that is used in the test section of the wave player is 16 bit, mono with 44 kHz sampling frequency. The audio data is 16 bit which are stored as 2's complement signed integers. The sound is mono channel so every 16 bit set is containing sound data and when a write transfer is required the APB is sending 32 bits which contains two 16 bit sets. In the C code implementation the 32 bits need to be divided into two sections and the 16 bit set which is at the lowest adress is sent to the APB slave interface first and then the other 16 bit set is sent. In the APB slave interface, the arrangements of the bits are made according to little-endian issue. Also the 2's complement signed bits are changed to unsigned by adding '1' to most significant bit of the spi_in. The most significant 12 bit of the data is used because of the DAC

specifications. The VHDL code and the figure below shows how the arrangements are made.




```
if(apbi.pwdata(7)='1') then
   spi_in(11)<='0';
else
   spi_in(11)<='1';
end if;
   spi_in(10)<=apbi.pwdata(6);
   spi_in(9)<=apbi.pwdata(5);
   spi_in(8)<=apbi.pwdata(4);
   spi_in(7)<=apbi.pwdata(3);
   spi_in(6)<=apbi.pwdata(2);
   spi_in(5)<=apbi.pwdata(1);
   spi_in(4)<=apbi.pwdata(0);
   spi_in(3)<=apbi.pwdata(15);
   spi_in(2)<=apbi.pwdata(14);
   spi_in(1)<=apbi.pwdata(13);
   spi_in(0)<=apbi.pwdata(12);
```

Figure 3.12 Little Endian Ordering example and Vhdl code implementation

3.2.3.2.4 C Code Implementation of Wave Player

After the changes are made in the APB slave interface, the C code implementation is made.

```
struct data_regs_t {
  volatile unsigned int flag_send_data;    // Can the driver send a data to Audio Core?
  volatile unsigned int flag_read_data;    // Can the driver receive a data from Audio Core?
};
struct data_regs_t  * data_regs = (struct data_regs_t  *)0x80000900;

main()
{
    unsigned int *i;
    unsigned int index,a,b,data;
    i=0x40010000;

  while(1) {

    while(data_regs->flag_read_data){

            data_regs->flag_send_data=data;

    if (index==0){
        a=*i;
        b=a>>16;
        data=b;
        index++;
        }
    else if (index==1){
        data=a;
        index=0;
        i++;
        }
    else
        index=0;

    }
    }
}
```

Figure 3.13 C code implementation of Wave Player

The simple program is modified to play a wave file. Volatile integer flag_read_data is added to the code, this checks if the DAC processed previous data by waiting for the flag output of the DAC to be set. The sound data which are 32 bit started to be taken from address 0x40010000. The 32 bit data is divided in two 16 bit sets and sent according to the issues that are mentioned in section 3.2.2.1 to address 0x80000900 where the audio core is attached. After the implementation of the wave player in C programming language, it is compiled with sparc-rtems-gcc and the generated sparc object is loaded to the board by GRMON. In addition to that, the sound data is loaded to address 0x40010000. The outputs of the audio core is connected to a 50 Ohm speaker. When the wave player program is run in GRMON, the player starts sending the sound data to the audio core and the music can be heard from the speaker. The wave sound file that is loaded to the board only contains 10 seconds of music data. The memory on the FPGA board has 1 MB capacity and if that capacity is increased then longer music data can be played.

3.2.3.3 MP3 player on chip

In the final step, after verifying audio core functionality with a simple test program and a simple wave player, the MPG123 software and the AMBA interface are modified to use the MPG123 application and play MP3 files.

3.2.3.3.1 Changes in MPG123 Software and AMBA Interface

The application has options for different audio outputs. When the application starts to run, it checks if there is an audio device and if there is not, it flushes the output of the frames to a dummy buffer. The *flush_output* function in audio.c is where the output is sent to the buffer. This function is modified to send output data to DAC by using the mp3_amba_interface.

```
struct mpg123_regs_t {
  volatile unsigned int flag_send_data;  //Data sent to mp3_amba_interface
  volatile unsigned int flag_read_data; //Can mp3_amba_int. receive data?
  };

struct mpg123_regs_t  * mpg123_regs = (struct mpg123_regs_t *)0x80000900;

int flush_output(audio_output_t *ao, unsigned char *bytes, size_t count)
{
    unsigned char *i;
    unsigned int a,b,data,countd;
    i=bytes;
    data=*i;
    countd=count;

  while(countd>0) {
    while(mpg123_regs->flag_read_data && countd){
            countd=countd-2;
        a=*i;
        b=a<<8;
        i++;
        a=*i;
        data=b+a+32768;
        i++;
        mpg123_regs->flag_send_data=data;
        }
    }
    return (int)count;
}
```

Figure 3.14 Modifications in audio.c to use audio core

The modified flush_output function gets its input in *bytes* buffer with the size of *count*. The function is modified only for 16-bit, mono MP3 files. In the code above, data in char declared *bytes* buffer are taken in to the integer declared *data* register byte by byte and then summed with 32768 to change from two's compliment to unsigned integer because DAC accepts unsigned integer input.

Moreover, the mp3_amba_interface is modified as shown below because unlike in the simple wave player case, the output is in big-endian format.

```
if (apbi.psel(pindex) and apbi.penable and apbi.pwrite) = '1' then
        spi_in <= (apbi.pwdata(15 downto 4));
```

After the modifications in software and hardware, MPG123 application with audio core is run successfully.

3.3 64-Point IDCT Core

In section 2.1.4.6 information about the IDCT function of the MP3 player is mentioned. One of the tasks of the thesis project is selecting a process consuming part of the software and make a hardware accelerator and an interface for communication with the AMBA bus. During the profiling of the MP3 player application, it is seen that the 64 point IDCT function is consuming 13.13% of the process and it has less complex

computations compared to the 36 point IDCT function. For those reasons, the 64 point IDCT function is selected and implemented in VHDL as a hardware core and attached to the LEON3 processor's APB bus as a slave. The profiling output of the design without IDCT core which is taken from the GRMON debug monitor is as follows.

```
Playing MPEG stream 0 of 0: test.mp3 ...
MPEG 1.0 layer III, 128 kbit/s, 44100 Hz mono
Title:   Forever                          Artist: Papa Roach
Album:   The Paramour Sessions
Year:    2006-09-12                       Genre:  Rock

[0:44] Decoding of test.mp3 finished.

Program exited normally.
grlib> prof
function                                  samples     ratio(%),
 INT123_synth_1to1                            758       47.16
 INT123_dct36                                 360       22.40
 INT123_dct64                                 211       13.13
 INT123_do_layer3                             104        6.47
 play_frame                                    35        2.17
 memcpy                                        22        1.36
 INT123_synth_1to1_mono                        14        0.87
 .rem                                          11        0.68
 IMFS_memfile_get_block_pointer                 6        0.37
```

Figure 3.15 Profiling output of MP3 Player in GRMON

The dct64 function used in the mpg123 library is named as 64-point as it uses a buffer array of 64 which stores input and output data and no further information is provided about it by developers. The implementation of the IDCT core starts with understanding of the C code of the 64 point IDCT function and a block diagram is generated according to the C code and the VHDL code implementation is made according to that block diagram. As can be seen from the figures 3.16,3.17 and 3.18, the 64 point IDCT has 5 calculation points. Each point contains additions, subtractions and multiplications. The IDCT core is taking 32 data values which are 32 bits each and giving 32 data values after IDCT calculations as output. The multiplications are made with cosine values and for those values cosine tables are generated and used. The cosine table used for the multiplications is as follows.

costab_1(0)	00802785	**costab_1(8)**	00be99ee	**costab_2(0)**	00809e8d	**costab_3(0)**	02901b3a
costab_1(1)	0081668b	**costab_1(9)**	00d6df9e	**costab_2(1)**	0085c278	**costab_3(1)**	00e664d7
costab_1(2)	0083f45b	**costab_1(10)**	00f8fa3b	**costab_2(2)**	0091233f	**costab_3(2)**	0099f1bd
costab_1(3)	0087f268	**costab_1(11)**	012b606a	**costab_2(3)**	00a5961d	**costab_3(3)**	008281f7
costab_1(4)	008d9838	**costab_1(12)**	017bf236	**costab_2(4)**	00c9c480	**costab_4(0)**	014e7ae9
costab_1(5)	00953b3a	**costab_1(13)**	020ecabc	**costab_2(5)**	010f8893	**costab_4(1)**	008a8bd4
costab_1(6)	009f5c6e	**costab_1(14)**	3685906	**costab_2(6)**	01b8f24b	**costab_4(2)**	014e7ae9
costab_1(7)	00acc03d	**costab_1(15)**	0a30a45f	**costab_2(7)**	0519e4e0	**costab_4(3)**	008a8bd4
costab_5							00b504f3

Table 2. Cosine Table used in IDCT Calculations

For the calculations simple vector adder, subtractor and multipliers are implemented and numeric_std libraries +, - and * operators are used. The simple vector adder makes addition of two 32 bit signed vectors, the vector subtractor takes one of the input's 2's complement and adds it with the other 32 bit signed input vector and in the multiplier, two 32 bit inputs are multiplied and the resulting 64 bits are shifted to the left 8 bits to match with the results of C code which uses its own "*REAL_MUL*" function written in assembly and its own cosine table. In addition the MSB 32 bits are taken as multiplication result. When the vector adder, subtractor and multiplier are implemented, the implementation of the top module is done. The top module contains 4 inputs and 2 outputs which are clk, rst, write, and in_a, read and idct_64 respectively. The write input is a 1-bit input which is used for checking if the APB bus has finished writing the 32 data values or not. The in_a input is a 1024 bit vector which contains 32 data values coming from the APB bus. The clk and rst inputs are system clock and reset. The output read is also a 1-bit value which is used for telling the APB bus that the calculation of the 64 point IDCT is finished. The idct_64 output vector is a 1024 bit vector which contains the calculated values.

When write input becomes '1' then the inputs are taken to an 32 column 32 row array and the signal for starting the calcultaion is set to '1'. The Spartan3 board has enough adders and subtractors so 16 addition and subtraction are made in one cycle but for multiplier case the Spartan3 board supports twelve 18x18 multipliers so in one cycle using 4 multiplication is selected and in each point of the IDCT calculation 16 multiplications are needed so for getting the result of the whole multiplications in one point, 4 clock cycles are needed. The inputs of the multiplier in each point is a value from the cosine table and the result of the subtraction so the multiplier waits for the subtraction operation and then starts doing the multiplication. For that reason after each subtraction finishes, the "sub_fin" signals are set to '1' and the multiplier starts calculation according to that signal's value. Each point is connected to each other because in each point the results coming from the previous point is used and for that reason calc_fin signals are used for telling the next point if the previous point is finished or not. When the previous point finishes the calculation than the next point uses the results coming from there. In the figures 3.16, 3.17 and 3.18 the block diagram of the 64 point IDCT can be seen.

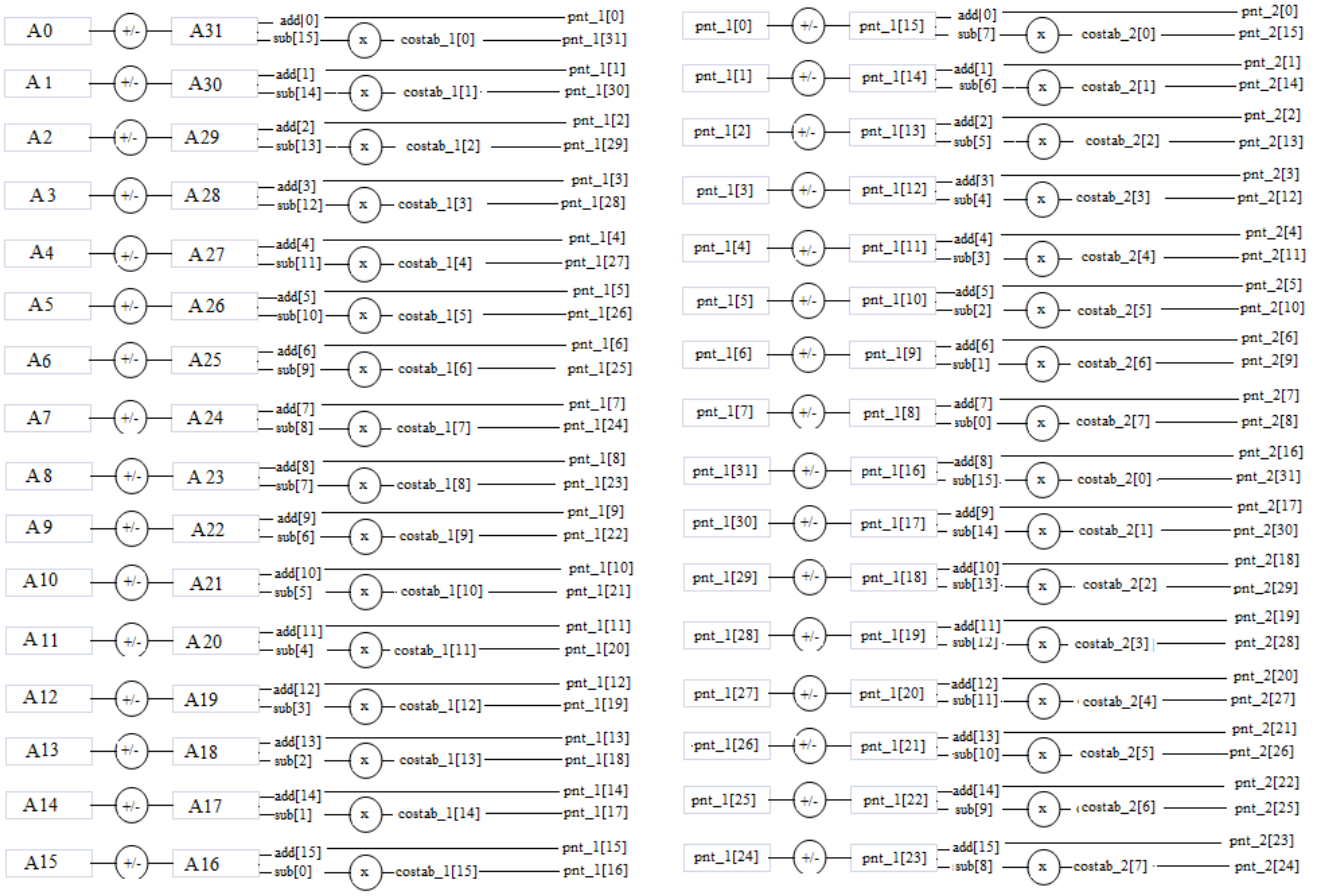


Figure 3.16 Block Diagram of first 2 calculation points of IDCT

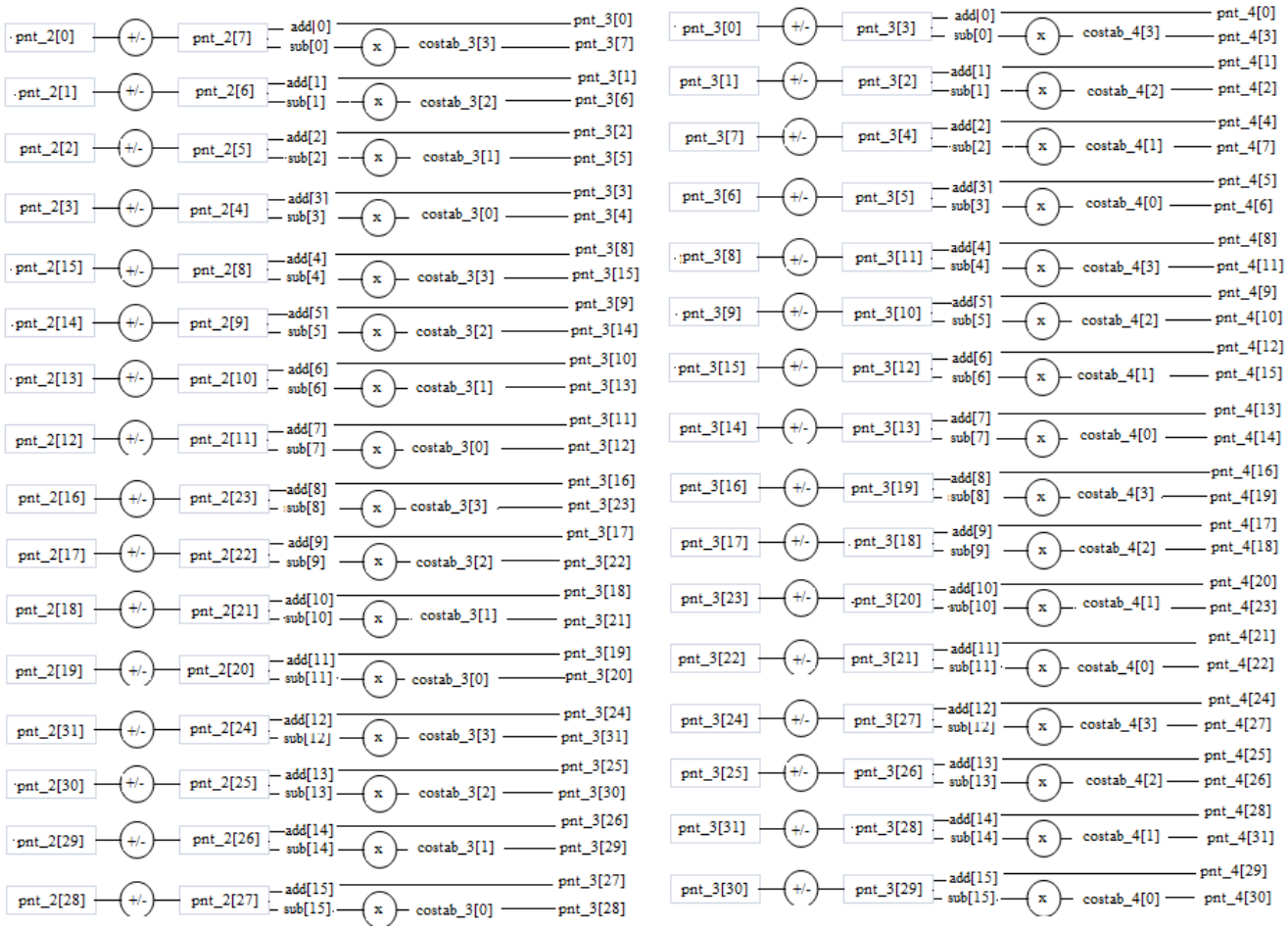


Figure 3.17 Block Diagram of 3rd and 4th calculation points of IDCT

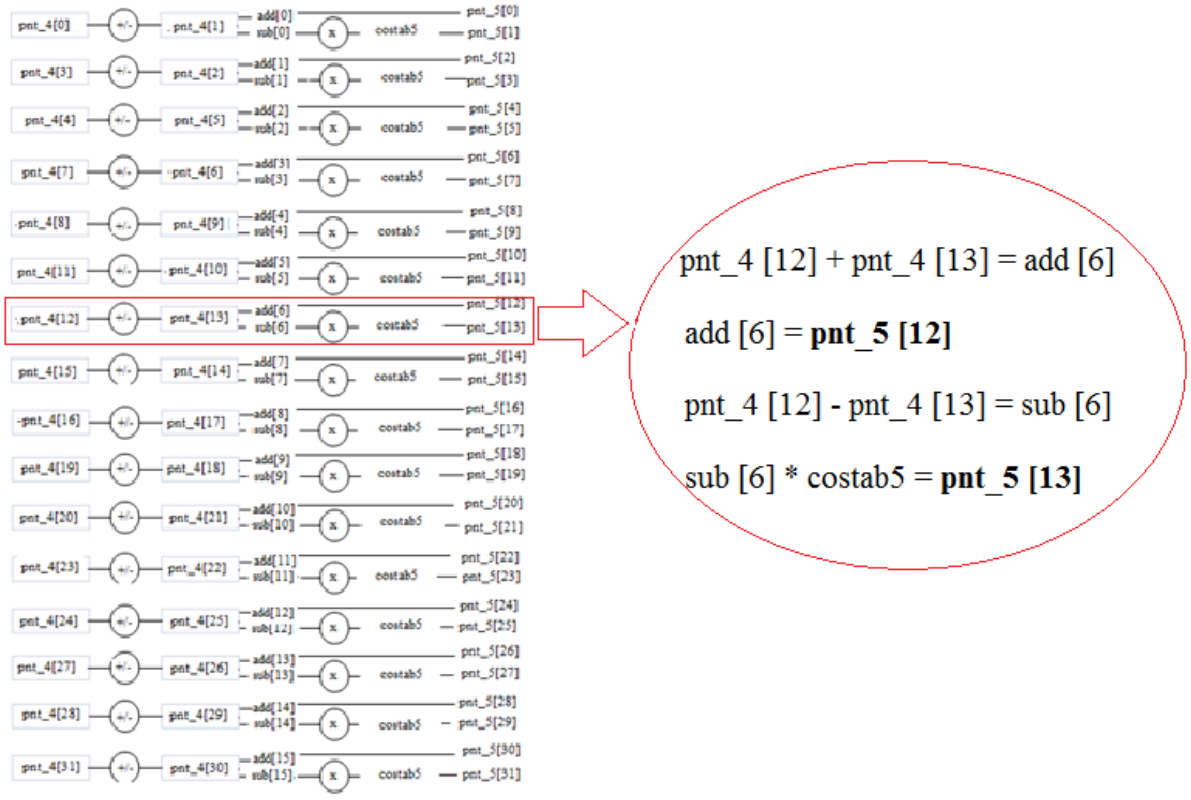


Figure 3.18 Block Diagram of last calculation point of IDCT

After the implementation of the IDCT core, simulation in Modelsim is done. The 32 data input values are taken from the MP3 player software and given to the simulation. The figure below shows the entire simulation of the 64 point IDCT calculation.

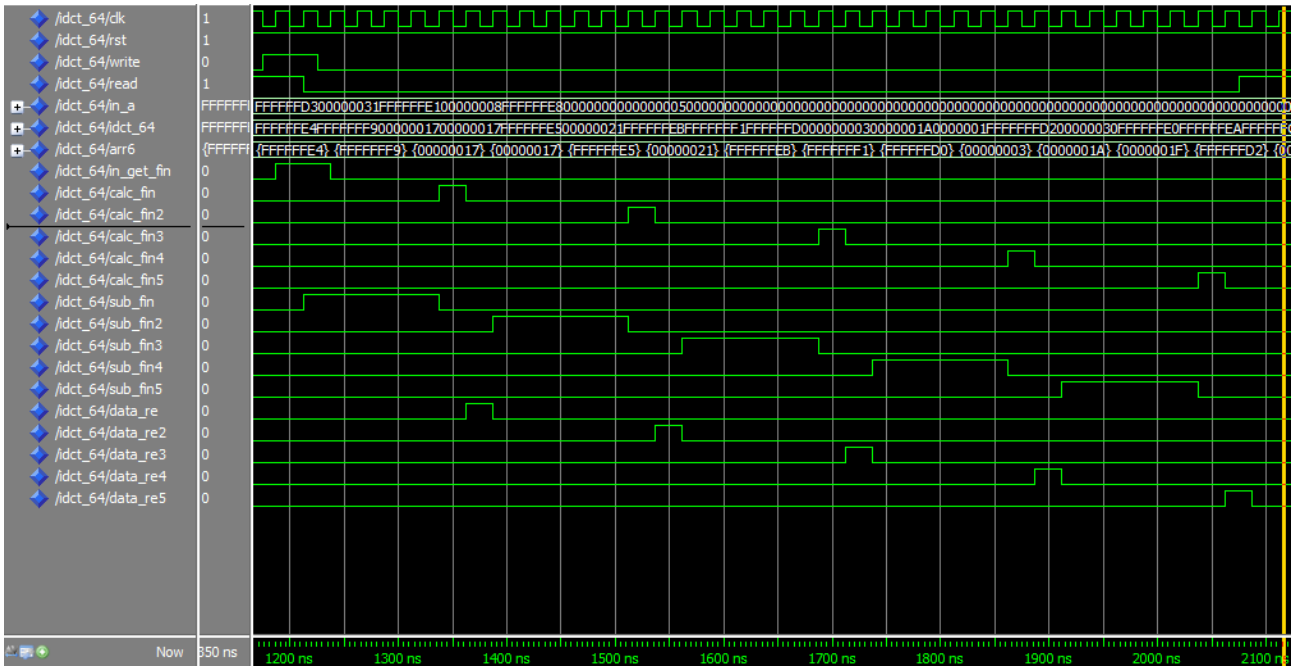


Figure 3.19 Entire Simulaton of 64 point IDCT Calculation

During the run of the MP3 player software the input values to the 64 point IDCT function are printed and taken for use in the simulation. Also the output of the calculation is taken from the software. First point by point comparison is made. After each calculation point the results are verified and the errors are fixed. Then at the end the final result coming from the software and hardware is compared. The test input values and the output values that are compared can be seen in figure 3.20. The table on the left shows the values taken from the software and the values on the right shows the values of the implemented core's output.

	Input	Output
0	ffffffd3	ffffffda
1	31	ffffffc4
2	ffffffe1	0000005f
3	8	ffffffa6
4	ffffffe8	39
5	0	0000002b
6	5	0000000d
7	0	ffffffa6
8	0	ffffffea
9	0	ffffffdb
10	0	0000003d
11	0	ffffffcd
12	0	0000002a
13	0	27
14	0	fffffffb
15	0	ffffffca
16	0	ffffffea
17	0	ffffffe0
18	0	30
19	0	ffffffd2
20	0	0000001f
21	0	0000001a
22	0	3
23	0	ffffffd0
24	0	fffffff1
25	0	ffffffeb
26	0	21
27	0	ffffffe5
28	0	17
29	0	17
30	0	fffffff9
31	0	ffffffe4

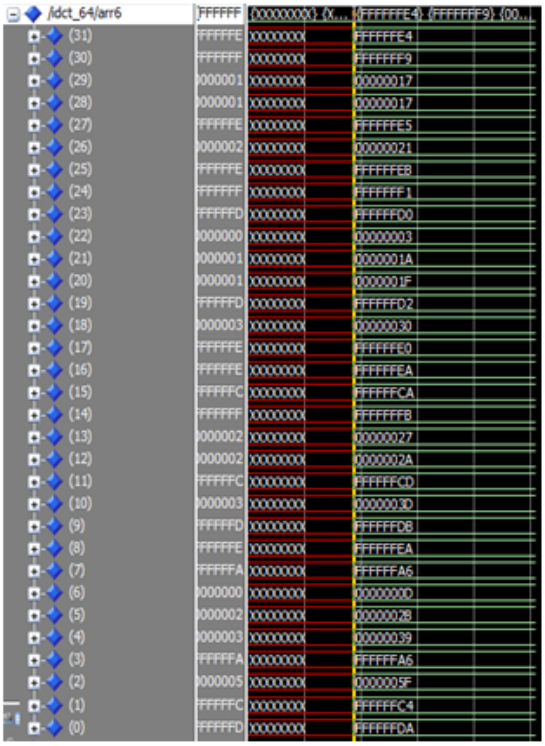


Figure 3.20 Input and output values taken from software and Outputs taken from Simulation

After the simulations are done in Modelsim, implementation of the APB slave interface is done. Attaching a peripheral as a APB slave is mentioned in section 3.2.2.4. The steps that are mentioned in that section is followed. The IDCT AMBA interface uses the IDCT core which is described in previous section. The IDCT core starts to calculate

from 32 given input when the *idct_write* signal goes high. When the calculation is done, *idct_read* signal goes high. The core needs to wait until 32 input data arrives, so the IDCT interface is the part where each 32-bit input is stored in an input array and sent to the core. It also gets the output array and sends it to the software one of the 32-bit outputs at a time.

```
-- APB BUS Write
  if (apbi.psel(pindex) and apbi.penable and apbi.pwrite) = '1' then
     if (in_cnt < 31 and (idct_read = '1' or flag= '1') then
        idct_in(in_cnt*32+31 downto in_cnt*32) <= apbi.pwdata;
        in_cnt <= in_cnt+1;
        idct_write <= '0';              ------becomes '1' if getting 32 data finish
        flag<='0';
       elsif (in_cnt = 31 and (idct_read = '1' or flag='1' )then
        flag<= '0';
        idct_in(in_cnt*32+31 downto in_cnt*32) <= apbi.pwdata;
        in_cnt <= 0;
        idct_write <= '1';              -----32 data were taken

    end if;

  end if;
```

Figure 3.21 APB BUS Write part of the IDCT AMBA Interface

Software sends the input array to the IDCT interface by using APB Bus Write part as in figure 3.21. If the IDCT core is ready to get data from the interface, *idct_read* or *flag* signals are high. When the last input is sent by the software, *idct_write* goes high and the calculation starts. After the inputs are sent, the software starts to read status of the IDCT core by checking the *idct_read* signal with the APB Bus Read part of the interface as in figure 3.22. When *idct_read* gets high, 32-bit outputs are taken one by one.

```
   -- APB BUS Read
if (apbi.psel(pindex)) = '1' then
 case apbi.paddr(4 downto 2) is
   when "000" =>  -- flag read stat
     rdata(1 downto 0) <= flag & idct_read
   when "001" => --  flag read stat
     rdata(1 downto 0) <= flag & idct_read
   when "010" => --  flag send finished
     rdata(1 downto 0) <= '0' & not idct_write;
   when "011" => --  idct write reset
     idct_write <= '0';
   when "100" => --  flag idct finished
      rdata(1 downto 0) <='0' & not idct_read;
   when others => -- read data

     if (read_cnt < 31) then
       rdata <= idct_out(read_cnt*32+31 downto read_cnt*32)
       read_cnt <= read_cnt+1;
      elsif (read_cnt = 31)then
       rdata <= idct_out(read_cnt*32+31 downto read_cnt*32)
       read_cnt <= 0;
     end if;
   end case;
end if;
```

Figure 3.22 APB BUS Read part in IDCT AMBA Interface

The IDCT AMBA interface is located on the APB Bus with pindex 10 which is not used by any other core in the design. Below is the added part in leon3mp.vhd for the interface.

```
idct_hardware1 : idct_amba_interface
      generic map (pindex  => 10, paddr => 10, pmask => 16#FFF#)
      port map (rst => rstn, clk => clkm, apbi => apbi, apbo => apbo(10) );
```

Figure 3.23 Addition of IDCT AMBA interface to leon3mp.vhd

```
struct idct_regs_t {
  volatile unsigned int flag_send_data;     // to send data to IDCT hardware
  volatile unsigned int flag_read_stat;     // to read status of IDCT HW
  volatile unsigned int flag_send_finished; // to check if IDCT HW received data
  volatile unsigned int idct_write_reset;   // to reset IDCT write flag
  volatile unsigned int flag_idct_finished; // to check if IDCT calculations done
  volatile unsigned int flag_read_data;     // to get data from IDCT hardware
};

struct idct_regs_t  * idct_regs = (struct idct_regs_t  *)0x80000a00;
```

Figure 3.24 Registers of the IDCT Hardware

After implementing the IDCT core and attaching the peripheral to the APB Slave, necessary modifications in MPG123 source code is done to use the IDCT core. 64 point IDCT calculations are done by the *dct64.c* in the source code. Because the 5 points of calculations in the code has time consuming multiplications with additions and

subtractions, they are all migrated into the IDCT core. The modified C code has 6 registers which are used to send data to the IDCT core, read data from the IDCT core and check the IDCT core status which can be observed in figure 3.24.

```
count=32;
stat=idct_regs->flag_read_stat;
if (stat=2 || stat=1)
  while (count){
          idct_regs->flag_send_data=*b1;
          count--;
          b1++;}

while(idct_regs->flag_send_finished);
idct_regs->idct_write_reset;

while(idct_regs->flag_idct_finished);
count=32;

while (count){
          *bufs=idct_regs->flag_read_data;
          count--;
          bufs++;}
```

Figure 3.25 Modifications in the C code of MPG123 source code

Sending of the input array starts after checking the IDCT core status. If the IDCT core is in reset mode, *flag_read_stat* is 2 and if it is in ready mode after a previous calculation, *flag_read_stat* is 1. Input array located by the *b1* pointer are sent to the IDCT core and *flag_send_finished* goes low, if *idct_write* signal in the core goes high which states that the calculation has started. The *idct_write* signal is reseted after the calculations has started by the *idct_write_reset* register until next data transfer. Then C code waits until the IDCT core finishes its calculations by checking *flag_idct_finished*. When the calculations are done, the output buffer pointed by *bufs* is loaded with output data from the IDCT core.

Simulation of the interface is done similar to the core simulations with Modelsim. Outputs of the software with the IDCT core are compared with outputs without the IDCT core. Outputs match exactly for different inputs.

3.4 Migrating the Design to Atlys FPGA board

Up to this chapter in the thesis project, the development platform used is Digilent Spartan3 FPGA board with the FPGA Xilinx XC3S1000-FT256. It has 1000K gates and 1MB fast asynchronous SRAM. The reason for starting the thesis project with this

development platform is the support of the GRLIB IP library and the familiarity with the development platform.

On the other hand during the process of the thesis project, it was seen that the chosen platform was not efficient and flexible for future uses. The 1MB SRAM was not enough for storing long music data and running big applications that need much more memory. Instead of a development platform that has a small memory and a small number of gates, Digilent Atlys Spartan6 FPGA development board [21] was a good alternative platform. It includes a Xilinx Spartan-6 LX45 FPGA which is optimized for high performance logic and offers 6,822 slices, each containing four 6-input LUTs and eight flip-flops, 2.1Mbits of fast block RAM, four clock tiles (eight DCMs & four PLLs) and 500MHz+ clock speeds. The board has these significant features;

• Xilinx Spartan-6 LX45 FPGA, 324-pin BGA package
• 128Mbyte DDR2 with 16-bit wide data
• 10/100/1000 Ethernet PHY
• on-board USB2 ports for programming and data transfer
• USB-UART and USB-HID port (for mouse/keyboard)
• AC-97 Codec with line-in, line-out, mic, and headphone
• real-time power monitors on all power rails
• 16Mbyte x4 SPI Flash for configuration and data storage
• 100MHz CMOS oscillator
• 48 I/O's routed to expansion connectors
• GPIO includes eight LEDs, six buttons, and eight slide switches

The development platform contains its own AC-97 codec which eliminates the need for a DAC card. In addition it has 128Mbyte DDR2 memory which is much more flexible and suitable for future uses. Besides these features, there is one disadvantage of the development platform. It is not supported by GRLIB IP library so changes and modifications need to be done in the template design of a similar development platform. In the next section those changes and modifications will be mentioned.

3.4.1 Implementation of LEON3 Processor

The Xilinx Spartan 6 design in GRLIB is used as a reference design and modified for Digilent Atlys Spartan 6 board. In the configuration of the LEON3 processor, the following modifications are done to use the Atlys board. In the clock generation

configuration, clock division and multiplication factors are selected as 12 and 6 respectively to get a 50 MHz processor operating frequency. The floating point unit is closed and MMU is selected. As the Atlys board has a USB-Uart cable, Serial Debug Link is selected as debug link. In the memory controllers section, the DDR2 SDRAM controller is closed and the Xilinx MIG DDR2 Controller is selected.

In the new release of GRLIB, there is also the Xilinx Memory Interface Generator to use DDR2 memory on the board. However, the Atlys board has different clock properties than the Xilinx board. The main clock in the Xilinx board is 27 MHz and there is a differential 200 MHz clock to drive the DDR2 memory, whereas the main clock in the Atlys board is 100 MHz and a 200 MHz clock for DDR2 memory is not available. To use DDR2 memory, a few changes are done in the design. The DCM (Digital Clock Manager) core is added to generate the 200 MHz single-ended clock from the processor clock and the Coregen project is modified to use single-ended clock instead of differential clock.

```
---DCM clk_gen---------
inst_dcm0 : DCM_CLKGEN
   generic map (
      --CLKFXDV_DIVIDE  => 5,        -- modify if other CLK than 50 MHz is desired
      CLKFX_DIVIDE    => 3,
      CLKFX_MULTIPLY  => 12,
      CLKFX_MD_MAX    => 0.0,
      CLKIN_PERIOD    => 20.0,
      SPREAD_SPECTRUM => "NONE",
      STARTUP_WAIT    => FALSE
   )
   port map (
      CLKFX     => clk200,         -- 200 MHz = 50 MHz / CLKFX_DIVIDE * CLKFX_MULTIPLY
      CLKFX180  => open,
      CLKFXDV   => open,          -- can be used as system clock, 50 MHz = MEM_CLK / CLKFXDV_DIVIDE
      LOCKED    => DCM0_LOCKED,
      PROGDONE  => open,
      STATUS    => DCM0_CLK_STATUS,
      CLKIN     => clkm,
      FREEZEDCM => '0',
      PROGCLK   => '0',
      PROGDATA  => '0',
      PROGEN    => '0',
      RST => not rstraw7
   );
```

Figure 3.26 DCM code added to leon3mp.vhd

The DCM generates the clk200 signal at 200 MHz from the 50 MHz main clock with 12/3 multiplication and division factors. The DCM works with *rstraw7* input reset which is delayed with seven gates. The DCM needs this delay to wait until the 50 MHz clock has settled. The DCM outputs *DCM0_LOCKED* signal which goes high when the clk200 signal is generated. The DDR2 memory needs an asynchronous reset when the

200 MHz clock signal is ready, so the *DCM0_LOCKED* signal together with the *rstraw7* signal is used for reseting the memory.

After all modifications are done, the LEON3 processor is connected via GRMON. By running the simple hello program successfully, functionality of the processor is verified. The Atlys board has its own audio chip, however the interface for the audio chip is not designed due to limited time. Instead of its own audio chip, the DAC card in the design for Spartan 3 board is used for playing music. The DAC card is connected to 8-pin Pmod connector of the Atlys board and necessary pin assignments are arranged in the user constraint file (UCF). Finally, the output of the MPG123 application is tested by playing the music with the IDCT core and without the IDCT core. As it can be seen in figures 3.15 and 3.27, the proces time ratio of the dct64 function is decreased to 8.15% from 13.13%. Test results also show that a test music file of 45 seconds is decoded in 23 seconds without the IDCT core and if the IDCT core is used in the system, the decoding time is decreased to 21.75 seconds.

```
Playing MPEG stream 0 of 0: test.mp3 ...
MPEG 1.0 layer III, 128 kbit/s, 44100 Hz mono
Title:   Forever                        Artist: Papa Roach
Album:   The Paramour Sessions
Year:    2006-09-12                     Genre:  Rock

[0:44] Decoding of test.mp3 finished.

Program exited normally.
grlib> prof
function                                samples     ratio(%),
 INT123_synth_1to1                          777       51.49
 INT123_dct36                               349       23.12
 INT123_dct64                               123        8.15
 INT123_do_layer3                            94        6.22
 play_frame                                  36        2.38
 memcpy                                      24        1.59
```

Figure 3.27 Profiling result of the MPG123 application with IDCT core

The MPG123 application is also verified by outputing the decoded raw music and comparing with generated wave music file in Linux environment. As in figure 3.28, necessary changes in the MPG123 source code are done to store raw music in a predefined address location. When the decoded audio output is stored in the predefined address, "*dump"* command of GRMON is used to dump data in the memory to file in s-record format.

```
// changes in the main function to specify output address
unsigned char *outputtest;  //output locating
outputtest=1074790400;      // integer value of 0x40100000 hex address
int indexout;
indexout=0;

            indexout=play_frame(outputtest);
            if(!indexout) break;
            outputtest=outputtest+indexout;


// changes in the play_frame function to get specified address
    // and copy audio output to that address
    unsigned int countd,index;
    unsigned char *i;
    index=0;
    i=audio;
    countd=bytes;

  while(countd>0) {

        countd=countd-1;
        *outputtest=*i;
        outputtest++;
        i++;
        index++;
    };
```

Figure 3.28 Modifications in the mpg123.c C code

4 Conclusion

The goal of this thesis work was to construct an embedded system with hardware and software co-design techniques. According to the objectives, a MP3 player application was implemented on LEON3 processor based hardware.

A time consuming computational kernel IDCT function has been implemented in hardware instead of software to speed up the application. In the same way, user defined IP cores can be added to LEON SoC and different system functions can be realized in hardware.

Using the APB to connect the DAC core resulted in computational delay between frames and sound quality was not good enough. The AHB would be a better choice to connect the DAC core because it can access to memory by using the AHB and get decoded data for each frame while the LEON3 processor does the decoding computations and as a result no computational delay occurs.

It is important to use implementation boards supported by GRLIB. As different boards has different signals or clocks, the modification of the template designs of other boards are time consuming and the outcome of the modification may not be stable. Moreover, the USB-Uart connection which is used to connect with GRMON was too slow to download data and the application to the FPGA and sometimes resulted in DCOM communication errors. Ethernet connection can be used for better download speed and stable connection.

The knowledge gained about LEON3 processors and other IP cores like audio or user-defined cores during the project can help students in their future works and this work can be a reference design for future implementations.

5 Future work

Up to now the thesis tasks mentioned in section 1.2 was completed. An application (MPG123) that has suitable kernels was identified. Required input and output interfaces and suitable interface for hardware acceleration were implemented based on the selected application and the application executed directly on a LEON3 processor. In addition a suitable kernel in hardware that is accessible over AMBA was implemented and interfaced to the software.

After the completion of the thesis tasks, it was seen that the project can be improved and different tasks can be added to the project work. The possible improvements and different tasks for future work are:

1. As it was mentioned during this report the Digilent Spartan3 FPGA board is not suitable for future uses according to the memory limitations but the Atlys board is more flexible and suitable for future uses.
2. The Digilent Atlys Spartan 6 board is ready to use with a LEON3 processor and different applications other than MPG123 can be run on it. For example, video player applications can be run and the HDMI video output of the Atlys board can be used with suitable interface designs.
3. The application was executed directly on a LEON3 processor instead of using an operating system but for future work an operating system can be used such as LINUX 2.6 or LINUX 2.0 [22]. Using an operating system has an advantage for using the MPG123 application in full performance.
4. Using an operating system brings more work to do and suitable driver implementations and necessary memory mappings need to be done so these can be also considered as future work.
5. During the thesis an audio core for the Spartan 3 board was implemented and the DAC card was used for getting the decoded music output but for Atlys board because of the time limitations a suitable AC97 codec interface was not implemented so as future work it can also be done.
6. In the thesis project 64 point IDCT was implemented as hardware but for getting better performance 32 point IDCT could also be implemented as hardware.

References

[1] Aeroflex Gaisler, *http://www.gaisler.com/products/grlib/grlib.pdf. GRLIB IP Library User's Manual.*

[2] ARM. AMBA Open Specifications. *http://www.arm.com/products/system-ip/amba/ amba-open-specifications.php.*

[3] Digital to Analog Converter Specifications and Data Sheet, *http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en024016#1*

[4] Xilinx Spartan 3 FPGA Board Information, *http://www.digilentinc.com/Products/Detail.cfm?Prod=S3BOARD*

[5] Modelsim Advanced Simulation and Debugging, *http://model.com/*

[6] Xilinx ISE Design Suite, *http://www.xilinx.com/products/design-tools/ise-design- suite/index.htm*

[7] Wave File Format and RIFF Bitstream Format, *http://en.wikipedia.org/wiki/WAV*

[8] Microsoft Wave File Format *https://ccrma.stanford.edu/courses/422/projects/WaveFormat/*

[9] Sparc V8 Architecture, *The Sparc Architecture Manuel*

[10] Cygwin Shell for GRLIB usage on Windows, *http://www.cygwin.com/*

[11] RS232 Standard, *http://en.wikipedia.org/wiki/RS-232*

[12] GRMON Debugging Monitor, *Grmon User's Manual, http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=190&Itemid=124*

[13] RTEMS LEON/ERC32 Cross-Compiler System (RCC-1.1.99.15) *http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=161&Itemid=109*

[14] K. Salomonsen et al., "*Design and Implementation of an MPEG/Audio Layer III Bitstream Processor," Master's thesis,* Aalborg University, Denmark, 1997

[15] MP3 Player Source Code, *http://mpg123.orgis.org/*

[16] TSIM ERC32/LEON simulator, *http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=38&Itemid=56*

[17] MPEG Layer III, *http://en.wikipedia.org/wiki/Mp3*

[18] Karlheinz Brandenburg, "*MP3 and AAC explained*", AES 17th International Conference on High Quality Audio Coding

[19] BIN2SREC library source code, *http://www.s-record.com*

[20] S-Record File Format, *http://en.wikipedia.org/wiki/SREC_(file_format)*

[21] Digilent Atlys Spartan6 Board Manual, *http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,836&Prod=ATLYS*

[22] Linux 2.6 for Leon with MMU, *http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=160&Itemid=108*

[23] D.A. Huffman "*A method for the Construction of Minimum-Redundancy Codes*", Proceedings of the I.R.E., September 1952, pp 1098-1102