TABLE OF CONTENTS

Acknowledgements

First and foremost I would like to sincerely thank my supervisor Dr. Neil Burgess, Senior Lecturer, University of Bristol for his guidance and support throughout this project. I would like to express my deep gratitude to Dr. David Wheeler, Technical Director, EnSilica Ltd. for providing all the necessary tools and technical support. I would like to thank EnSilica Ltd. for providing me this wonderful opportunity. I would also like to thank the Phd students Atukem, Mohsin and Daniel for their invaluable suggestions.

Summary

The project aims at implementing the Advanced Audio Coding decoder algorithm on EnSilica's eSi-RISC Embedded Processor. Advanced Audio Coding (AAC) is an audio compression algorithm standardised as a part of the MPEG-4 specifications. AAC includes 48 audio channels in one stream with a bandwidth of 96 kHz. It has a better compression, simpler and efficient filterbank than its predecessor mp3. Hence it is widely used and is the default audio format for most of the portable devices iPhone, iPad, PlayStation, Xbox, etc.

eSi-RISC 3250 is a configurable 32-bit Embedded Processor and can be configured for Harvard or von Neumann memory architecture. It enables users to balance functionality, performance, area and power to produce an optimal processing platform. It has a 5-stage pipelined RISC architecture and allows intermix of 16 and 32-bit instructions. Intermixed 16 and 32-bit instruction gives exceptional code density without compromising performance. A highly dense program code like AAC will benefit from this feature. The high code density of the AAC algorithm demands a lot of program memory and eSi-RISC 3250 has a 32 MB DDR2 SDRAM. eSi-RISC supports upto 96 user-defined instructions. User-defined instructions are small hardware accelerators attached to the main processor and can access the register file up to 2 input arguments and 1 output argument. In software these user-defined instructions have a "C" subroutine call prototype for simple integration into the code. A shared library can implement the functionality of the instruction by connecting to real hardware. Thus user-defined instructions provide hardware acceleration to the program code execution and this is very helpful if the program code has computationally intensive operations. AAC algorithm has compute-intensive operations like IMDCT (Inverse Modified Discrete Cosine Transform), IFFT (Inverse Fast Fourier Transform) complex multiplications, filter window operations, etc. It is interesting to see the impact of the user-defined instructions on these operations and what acceleration they produce for the algorithm execution. Thus the aim of the project is to implement the AAC decoder algorithm on eS-RISC 3250 embedded processor and to optimise the algorithm using the user-defined instructions.

The following are the main achievements in this project.

- I implemented the AAC decoder algorithm for the first time on EnSilica's eSi-RISC 3250 Embedded processor.
- I optimised the algorithm reducing clock cycles by 56% providing real time decoding capability (A 17.55 sec AAC audio clip which initially took 22.33 sec to decode, took only 9.78 sec after optimisation of the decoder algorithm. (Pages 20 - 37)
- I developed a short algorithm in C which performed 32-bit fixed point multiplication. (Page 29)
- I implemented the user defined instructions in hardware using Verilog HDL which can be attached to eSi-RISC 3250 core in future. (Pages 39-42)

Chapter 1

Introduction

Advanced Audio Coding (AAC)

Transmission of large amount of data through a network requires larger bandwidth. This becomes a limitation when large number of users send/receive data on the same network simultaneously. This can be overcome by compressing the data. Audio coding algorithms are used for the same purpose where digital audio signals are compressed reducing the amount of information required for transmission or storage of an audio signal. Audio coding requires two types of algorithms. The first type is the encoding of audio signals where the signals are represented as a coded bit-stream. This representation compresses the signals allowing them to be transmitted conveniently. On the other end the received data has to be decompressed to be played back. This is done by decoding the audio signal which is the next step. The decoder receives the bit-stream and reconverts it into an uncompressed signal.

1.1 Digital Audio

Representation of sound in the form of digital signals gives digital audio. It is often preferred to represent sound in digital form because analog form of audio has certain drawbacks.

- Compressing a digital data is much easier than compressing an analog data.
- It is easier to apply the quality enhancement techniques on the digital signal rather than the analog signal.

The idea behind Digital Audio is to use analog-to-digital conversion technique to represent physical sound in numbers.

1.1.1 Analog-to-digital conversion:

Analog-to-digital conversion involves three steps: Sampling, Quantization and Encoding. This is shown in Figure 1.1.

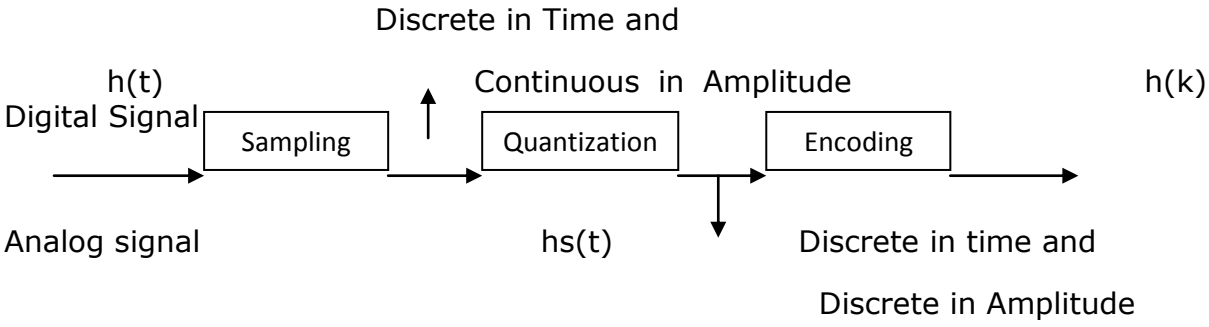


Figure 1.1: Analog-to-Digital conversion

Encoding of sampled and quantized signals are done using error correction codes. These encoded signals are uncompressed data and occupy a large network bandwidth. They are not suitable for digital audio streaming. Hence there is a requirement for digital audio signals to be compressed. There are various compression techniques used:

- Lossless compression
- Lossy compression

Lossless compression: In this form of compression, there is no loss of data. A data compressed using lossless compression technique can be recovered back without any information being lost.

Lossy compression: A lossy compression technique involves some loss of information. But it does give a higher compression rate as compared the lossless compression.

The method of compression to be chosen depends entirely on the type of application. Audio signal compression is usually a lossy type of compression. This is because audio coding technologies are based on Psychoacoustics.

1.2 Psychoacoustics

"The term Psychoacoustics describes the characteristics of the human auditory system on which the modern Audio coding technologies are based" [1]. The human ear cannot perceive certain sound elements in audio signals due to the effect of masking. Masking is a phenomenon where a weaker sound is dominated by the presence of a stronger sound. (Conversation with a friend is not possible in a room with blasting music being played).Having a weaker sound becomes unnecessary as it cannot be heard in presence of a higher sound. Hence the weaker sound can be eliminated resulting in the compression of the data.

The phenomenon of masking happens in the following ways:

- Absolute threshold of hearing
- Frequency Masking
- Temporal Masking

1.3 Audio compression standards

The Moving Pictures Experts Group (MPEG) has established many audio compression standards. Among the various standards, the MPEG-1 layer III (commonly known as MP3) is the most widely used MPEG standard for audio compression. MPEG -1 layer III provided compression at a bit rate of 128kbps. Advanced Audio Coding (AAC) is a successor of MPEG-1 layer III.

1.4 Advanced Audio Coding

Methods of compression that exploit psychoacoustics to eliminate irrelevant audio signal data is referred to as "Perceptual Audio Coding". Advanced Audio Coding (AAC) is a Perceptual Coding method. It is part of MPEG-4 specification. At a low bit rate of 64kbps, AAC achieves high quality audio signal compression with a low bandwidth. This makes it the most preferred scheme for digital audio compression. At higher bit rates, AAC can support a maximum of 48 audio channels and provides full bandwidth of up to 96 kHz. AAC provides algorithmic

tools to efficiently code and compress the audio. A 'profile' is a subset of these algorithmic tools. The AAC standard has three different profiles:

- Main
- Low-complexity
- Sampling-rate-scaleable

Main: The main profile demands more processing power. It has backward prediction.

Low-complexity profile: Simply known as LC-profile, it is the simplest and the most widely used profiles. It has less compression involved to save processing and CPU usage.

Sampling-rate-scaleable(SRS): Also known as Scaleable Sample rate(SSR) profile. This profile has the ability to adapt to different bandwidths.

All the profiles have different characteristics. Based on the complexity of the input data stream to be decoded and the desired performance, a suitable profile can be chosen. This is called Modular Encoding.

As discussed earlier, audio encoding exploits psychoacoustics and is also called 'Perceptual coding'. A typical perceptual coder will have a psychoacoustics model, a filter bank, quantization unit and a spectral processing unit. The structure of a typical perceptual coder is shown in Figure 1.2.

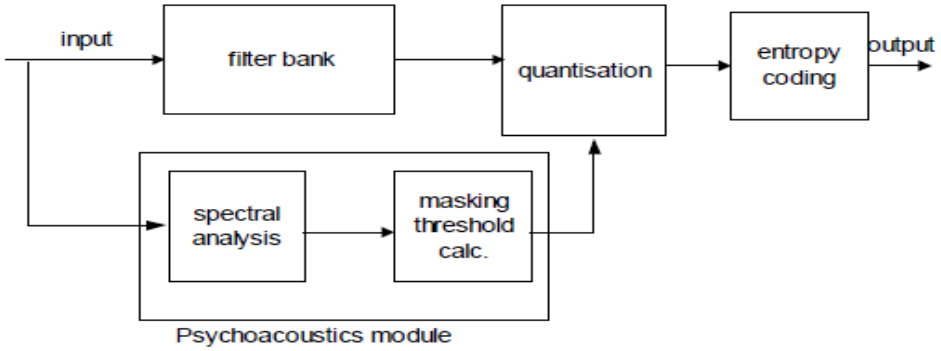


Figure 1.2: Basic structure of perceptual audio coder [2]

Basic perceptual audio coder has a filter bank which uses Modified Discrete Cosine Transform (MDCT). The purpose of this filter bank is to remove all the aliasing errors from the signal and help in faithful reconstruction of the original signal. However, this may not seem to be a good idea as the quantization noise is introduced after the quantization process. But the perceptual coder will make sure that the effect of quantization noise is not heard and the psychoacoustical module helps in achieving this. The psychoacoustic module as discussed earlier (section 2.2) makes use of the masking properties of the human ear. But the real difficulty is in calculating the masking threshold. The calculation of masking threshold is among the computationally intensive tasks of the encoder [2]. AAC encoder has an extra "Window Length Decision" block compared to the basic perceptual coder. This allows the encoder to choose one of the two window sizes based on whether the input signal is transient or stationary.

1.5 AAC Decoder

Figure 1.3 shows the functional blocks of the AAC decoder. The decoding operation starts with the bit stream de-formatter. The M/S (Middle/Side coding), PNS (Perceptual Noise Substitution), Intensity/Coupling, TNS (Temporal Noise Shaping) functional blocks were discussed earlier. Only the functional blocks which are mainly involved in the decoding process are discussed below:

Bit stream De-Formatter: The incoming bit stream is arranged by the de-formatter according to AAC standards. The length of the AAC frames varies from frame to frame. There are 1024 PCM samples in each frame. Also, each frame carries two different headers.

Noiseless Decoding: The noiseless decoder is used to reduce the redundancy in the scale factors and the quantized spectral coefficients [6]. The quantized spectral coefficients are segmented into various sections. Each section can be coded using Huffman codebook. The Huffman codebook contains codewords. When the bit stream is read, the codewords in it are searched for the corresponding codebook. It is therefore important to use efficient Huffman decoder.

Huffman decoders: There are two classes of Huffman decoders: Parallel and Sequential. The parallel decoder has a varying input data rate [9]. However the output data rate is constant. High decoding rate can be achieved using parallel decoder. The Sequential decoder has a constant input data rate. They can be realized using look up tables stores in the memory. This facilitates the decoder to achieve high speed decoding.

Prediction: There exists autocorrelation between the spectral components of the two preceding frames [6]. For each spectral component, up to 16 kHz there exists a predictor. The predictor utilizes the autocorrelation and removes redundancies between the spectral components.

Filter bank: The filter bank converts the spectral coefficients into time-domain values. This is carried out by using Inverse Modified Discrete Cosine Transform filter (IMDCT). This transform provides flexibility in window size matching and is very efficient. The IMDCT employs a technique called Time-Domain Aliasing Cancellation [8]. For each channel, the N/2 time frequency values X_k into N time domain values X_n [8]. The analytical expression for Inverse Modified Discrete Cosine Transform is [7]:

$$X_n = \sum_{k=0}^{N/2-1} Xk \cos\left[\frac{2\pi}{N}(n+no)(k+\frac{1}{2})\right]$$

Where n= 0,1,2.....,N-1

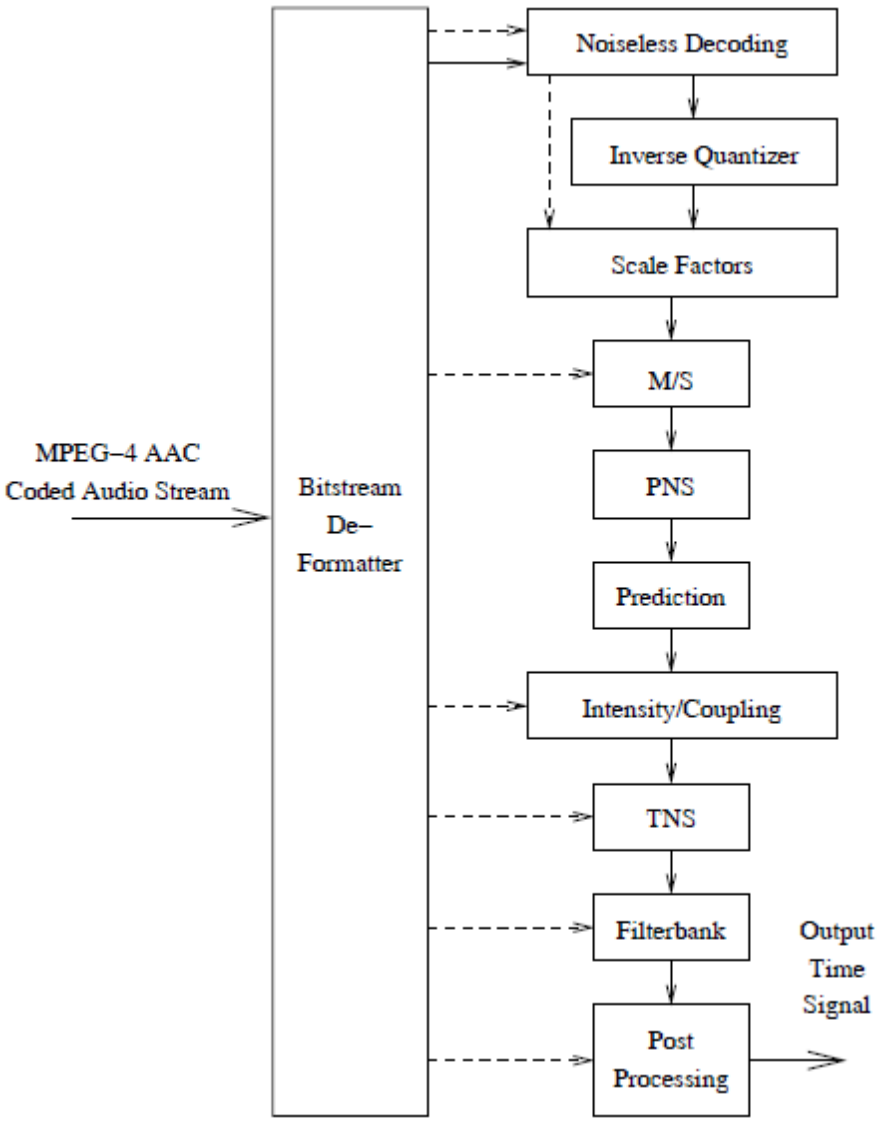


Figure 1.3: MPEG-4 AAC Decoder Block Diagram [6].

When n= sample index, i= window index, k= spectral coefficient index, N= window length based on the window sequence value, $n_o = (N/2 + 1)/2$.

Using the symmetry of the output, the IMDCT can be written as [7]:

$$x_{2n} = \sum_{k=0}^{N/2-1} Xk \cos\left[\frac{2\pi}{4N}(4n+1)(2k+1) + \frac{\pi}{4}(2k+1)\right]$$

Thus the filter bank is a critical component of the AAC decoder as it converts the time-frequency representation of the spectral components into time-domain output signal.

1.6 eSi-RISC – Configurable Embedded Processor

EnSilica's eSi-RISC 3250 is configurable microprocessor architecture for embedded systems. It enables users to balance functionality, performance, area and power to produce an optimal processing platform

1.6.1 Architectural Features

- eSi-RISC 3250 is a 32 bit processor.
- It can be configured for Harvard or von Neumann memory architecture.
- 32 general purpose registers.
- It has a 5-stage pipelined RISC architecture.
- It has 32 general purpose registers.
- 104 basic instructions and 10 addressing modes.
- Supports up to 96 user-defined instructions.
- It is implemented in as little as 8k ASIC gates for minimum 16-bit configuration.
- It allows intermix of 16 and 32-bit instructions.
- It uses a industry standard bus architecture for IP interconnection (AMBA APB/AXI)

The main features of the architecture are as follows:

• A wide range of fundamental architectural parameters are configurable. The most important of these is the intrinsic word width, which can be configured as 16, 32 or 64-bits. These three values provide sufficient scope to cover high-end application like AAC.

• It is a RISC-like, load/store architecture. The RISC architecture allows for simple area efficient implementations as well as aggressive micro-architectural designs implementing pipelining and super-scalar out-of-order execution. RISC architectures are also an excellent target for optimising compilers.

• It uses a mixed 16 and 32-bit instruction set encoding. This allows for exceptional code density, without compromising performance.

• Its instruction set is both configurable and user extendible where the implementation technology allows. For area critical applications, the ability to configure out unnecessary functionality reduces cost and power consumption. Allowing application-specific instructions not only allows for otherwise unobtainable performance levels to be reached, but also helps to future-proof the architecture.

Its RISC pipeline has 5 stages and allows high performance applications to achieve high clock frequencies. The C/C++ compiler provides optimization and schedules the instructions such that the latency problems are eliminated. Static branch prediction is incorporated to minimize the cost of branch instructions. Apart from the basic instructions, the instruction set allows system level instructions to control the external interrupts and software interrupts. The system level instructions also allow the processor to enter the low power state. There are 96 user-defined instructions including many optional instructions and addressing modes to choose from.

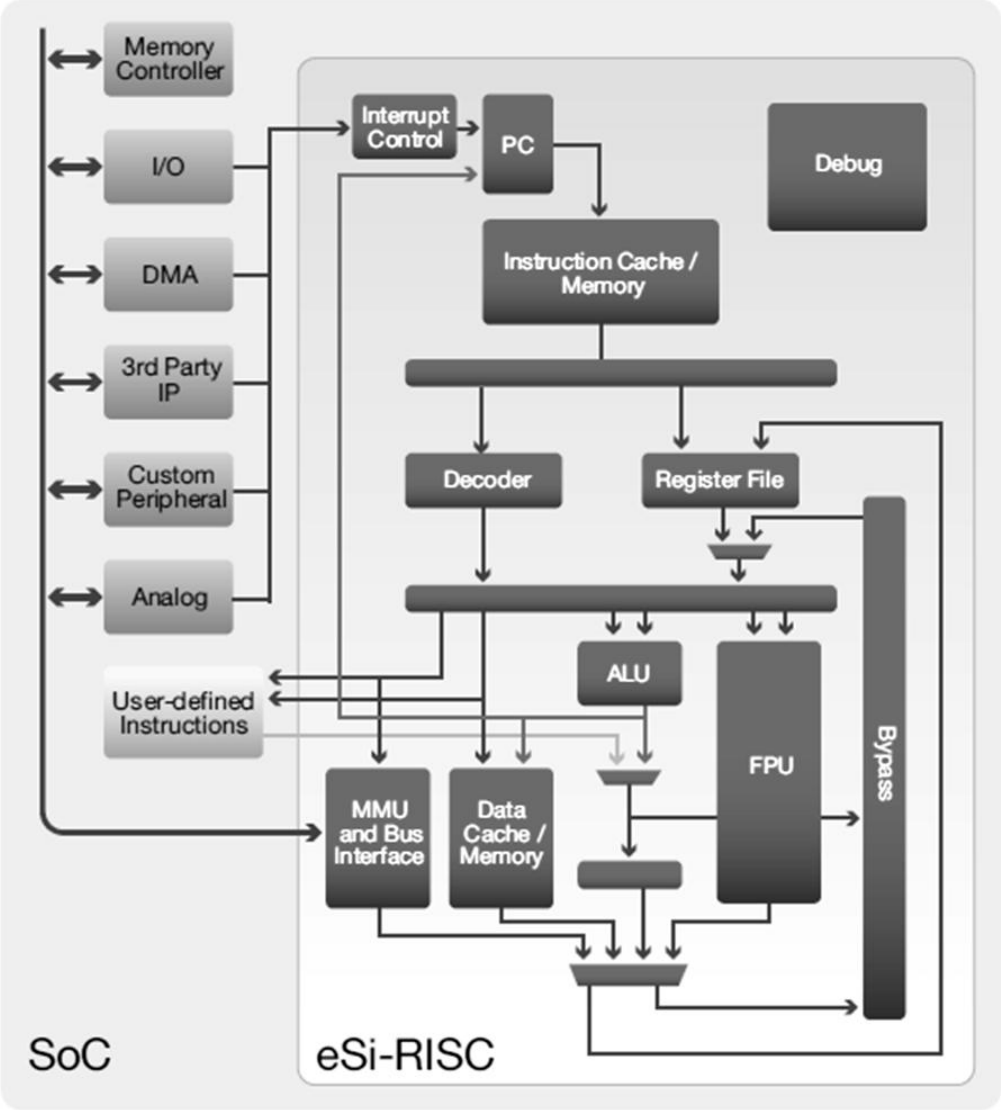


Figure 1.4: eSi-RISC 3250 Configurable Embedded Processor architecture

1.6.2 Configuration options

eSi-RISC is a configurable architecture. This allows implementations to be optimised for a given application or set of applications. The configuration options available are detailed in the following tables. Typically, the configuration will be fixed after implementation, although for reconfigurable technologies, such as FPGAs, limited run-time reconfiguration may be possible.

eSi-3250 CPU Configuration

The instruction set configuration for the eSi-RISC 3250 is shown in the table below. The value TRUE indicates that the instruction set is implemented and FALSE indicates that it is not implemented. Any instruction set can be implemented manually by passing flags to the compiler. Multiply and barrel shifting instructions are implemented in eS-RISC 3250.

Instruction Set

Option	Value
MULTIPLY_ENABLED	TRUE
DIVIDE_ENABLED	FALSE
BARREL_SHIFT_ENABLED	TRUE
ROTATE_ENABLED	FALSE
ABS_ENABLED	FALSE
MIN_MAX_ENABLED	FALSE
CLZ_ENABLED	FALSE
CO_ENABLED	FALSE
FFS_ENABLED	FALSE
USER_ENABLED	TRUE
FP_SINGLE_ENABLED	FALSE
FP_DOUBLE_ENABLED	FALSE
FP_DIVIDE_ENABLED	FALSE
FP_SQRT_ENABLED	FALSE
FP_MULADD_ENABLED	FALSE
SWAP_ENABLED	FALSE
BITFIELD_ENABLED	FALSE
SIGN_EXTEND_ENABLED	FALSE
PARITY_ENABLED	FALSE
LOAD_MULTIPLE_ENABLED	FALSE
LOAD_LOCKED_ENABLED	FALSE
LOOP_ENABLED	FALSE
BRANCH_ZERO_ENABLED	TRUE
MOVCC_ENABLED	TRUE
STOP_ENABLED	TRUE
SCALL_ENABLED	TRUE
VECTOR_MULTIPLY_ENABLED	FALSE
VECTOR_DIVIDE_ENABLED	FALSE
VECTOR_BARREL_SHIFT_ENABLED	FALSE
VECTOR_ROTATE_ENABLED	FALSE
VECTOR_ABS_ENABLED	FALSE
VECTOR_MIN_MAX_ENABLED	FALSE

Addressing Modes

Option	Value
SCALED_INDEX_ADDR_ENABLED	FALSE
UPDATE_ADDR_ENABLED	FALSE

Memory Configuration Options

Option	Value
ENDIAN	LITTLE
MMU_ENABLED	FALSE
MMU_DAT_ENABLED	FALSE
MMU_PTCACHE_ENTRIES	0
ICACHE_ENABLED	TRUE
ICACHE_ASSOCIATIVTY	2
ICACHE_SETS	64
ICACHE_BYTES_PER_LINE	64
DCACHE_ENABLED	TRUE
DCACHE_ASSOCIATIVTY	2
DCACHE_SETS	64
DCACHE_BYTES_PER_LINE	64
IMEM_ENABLED	FALSE
DMEM_ENABLED	FALSE

Interrupts

Option	Value
INTERRUPTS	32
FAST_INTERRUPTS_ENABLED	FALSE
NMI_ENABLED	TRUE

Debug

Option	Value
BREAK_ENABLED	TRUE
DEBUG_ENABLED	TRUE
JTAG_ENABLED	TRUE
BREAKPOINTS	4
WATCHPOINTS	2

Address Map

Address	Peripheral
0x00000000 - 0x00ffffff	16MByte FLASH
0x02000000 - 0x020fffff	1MByte SRAM
0x04000000 - 0x05ffffff	32MByte DDR2 SDRAM
0x80000000 - 0x800000ff	UART
0x80001000 - 0x800010ff	GPIO
0x80002000 - 0x800020ff	Timer
0x80003000 - 0x800030ff	Timer
0x80004000 - 0x800040ff	Watchdog
0x80005000 - 0x800050ff	SPI
0x80006000 - 0x800060ff	I2C
0x80007000 - 0x800070ff	PS/2
0x80008000 - 0x800080ff	EMAC
0x80009000 - 0x800090ff	LCD

Chapter 2

Background

There has been ample work on AAC and its implementations in the last decade. Its modules have been implemented on various platforms, DSP processors like Texas TMS320C31 [12], Motorola DSP56300 [11], Intel Pentium III processor [13] and Embedded processors like ARM [7, 14], PowerPC, etc. Researchers have worked on optimizing the functional blocks in the AAC encoder and decoder blocks [10, 15]. All internet streaming, mobile and broadcasting applications incorporate AAC codec and this increasing demand is forcing its implementation on modern microprocessors.

From published work we can understand the required features of a target processor for AAC implementation. Firstly, it should have a good operational performance and must be able to reach atleast 30 MIPS [17]. This means that the target processor should have a high operating clock frequency. The sound quality of the decoder output is affected by the accuracy of the decoding operation [16]. This is because the fixed point ALU produces quantization noise. In order to minimise this quantization noise, a 32-bit data path is required in the fixed-point ALU architecture. The other requirements that have been suggested are that the target processor should have a multiplier to reduce computational loads and a Barrel shifter to perform the shifting operations during multiplications.

One of the published work talks about the steps to be taken once target processor is chosen. The first step is to modify the code to be suitable for target processor so that the cross compiler generates a better code. Code generated by the C-cross compiler is more efficient than the hand written assembly code. Hand-written assembly is time consuming and incurs more development costs [16].

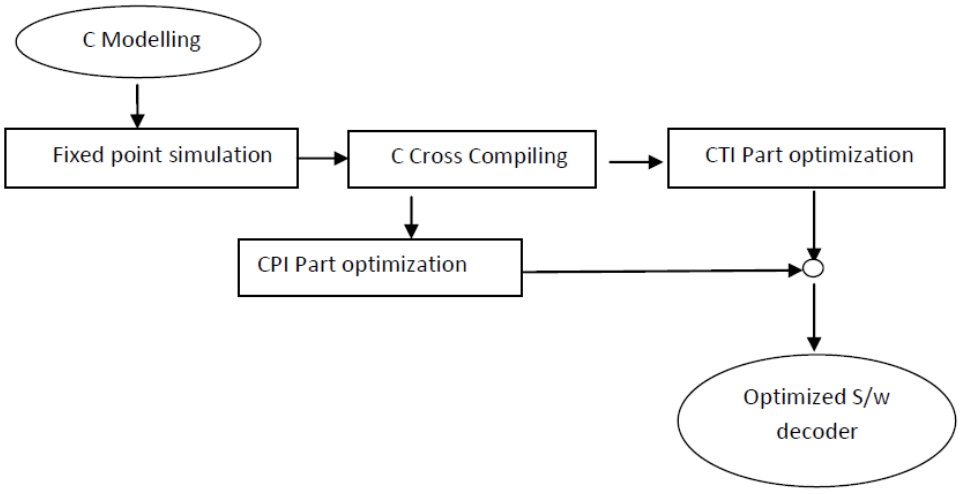


Figure 2.1: Work flow in AAC decoder implementation. CTI-Control Intensive and CPI-Compute Intensive. Modified and adapted from [16].

The AAC Decoder algorithm can be partitioned into 2 parts: control-intensive part and computation-intensive part (Figure 2.2) [16]. The control-intensive part has operations that are related to program flow control. The computation-intensive part has operations on high computational loads and lead to high cycle count. Control-intensive operations have low computational load but are difficult to control. For faster execution of decoder algorithm, computationally intensive operations have to be optimised.

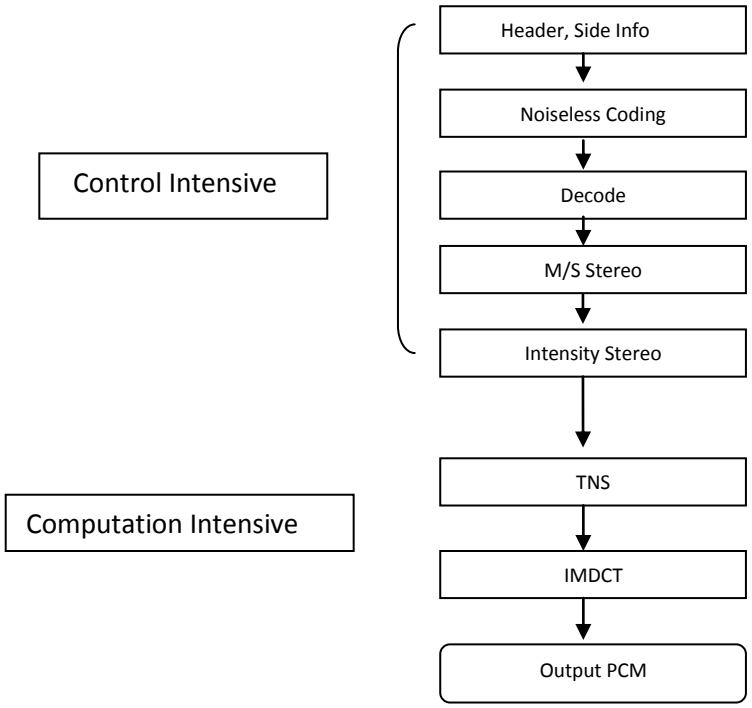


Figure 2.2: Showing the Control-intensive and Computation-intensive parts in the algorithm [16]

In the AAC algorithm the Huffman decoding routine is the control-intensive part and the filterbank routines are the computationally-intensive part which has many complex multiplications [16]. Most of the embedded processors use a 32-bit x 8-bit multiplier and cannot perform complex fixed point multiplication in a fewer cycles. A previously published work has shown that 32-bit x 16-bit multiplication (Figure 2.3) gives a good audio output quality in a MPEG AAC audio decoder [17].

To identify the compute intensive operations, profiling of the algorithm is necessary at run time. Results of runtime profiling, performed on a general PC environment and on ARM platform can be seen in Figure 2.5. Filterbank operation consumes the maximum ratio of cycles on both the processors [18]. Functions like faad_imdct, _MultHigh, ComplexMult, ifilter_bank, passf4pos in the filterbank operation are frequently called and have complex computations (Figure 2.4) [19].

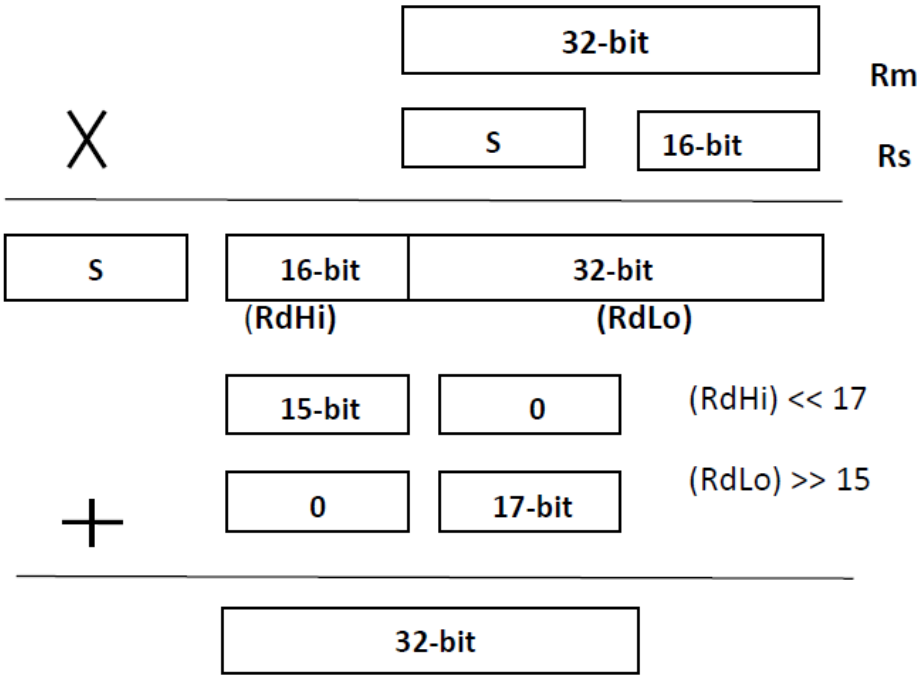


Figure 2.3: 32-bit x 16-bit fixed point multiplication [16].

The approaches taken for optimisation as seen in the previously published work have been either algorithm level optimization (developing better algorithm for the decoder blocks) [10, 15], Software level optimisation (writing inline assembly language in the program code, performing compiler level optimization like loop unroll, etc) [16] and Hardware level optimisation (having a dedicated hardware to run the compute intensive operations and increase its execution speed) [18].

Function name	Calls
Passf4pos	51240
Faad_imdct	13422
Ifilter_bank	11280
ComplexMult	15686400
_Mulhigh	22671360
MUL_F	22671360
Output_to_PCM	5640

Figure 2.4: Showing the number of calls made to the critical functions in the decoder algorithm [19] (adapted and modified).

Block	PC	ARM
	Ratio of Cycles	Ratio of Cycles
Noiseless Decoding	32.5%	18.9%
IQ & rescale	4.0%	10.3%
Stereo processing	6.3%	1.5%
TNS	9.7%	0.8%
Filterbank	47.5%	68.7%
Total	100%	100%

Figure 2.5: Complexity analysis of the AAC decoder [18] (adapted and modified).

Aims and Objectives

This project aims at implementing successfully the Advanced Audio Coding (AAC) algorithm on EnSilica's eSi-RISC embedded processor.

The objectives are:

1. Familiarising with the eSi-RISC processor architecture, Eclipse Integrated Development Environment and GNU tools.
2. Familiarising with the AAC decoder Algorithm.
3. Porting the fixed point software version of AAC decoder to eSi-RISC.
4. Profile the code to investigate the behaviour of the decoder algorithm in order to determine the most compute intensive operations.
5. Write custom instructions and measure the improved performance in terms of clock cycles.
6. Achieve performance improvement by at least 20%.
7. Code the user hardware in Verilog.
8. Demonstrate in real hardware by playing the stored music from a memory chip in the FPGA onto a loudspeaker attached to its port.

Chapter 3

Fixed-point implementation

Embedded systems do not carry a floating point unit. They use fixed point arithmetic and this helps in minimizing the cost and saves power consumption. In order to run an algorithm on a embedded processor it is necessary for the algorithm to have fixed point arithmetic calculations. This was the first step in implementation of AAC Decoder on eSi-RISC 3250. This chapter discusses the fixed point conversion of the decoder algorithm, porting of the algorithm to the eSi-RISC platform and the difficulties encountered in the process.

The FAAD2 is the open source for AAC decoder and was downloaded from http://audiocoding.com. The downloaded zip file contained the source files, header files, plugins and aacDECdrop. aacDECdrop is a Windows application that decodes any AAC file dragged and dropped on it. The front end folder in the FAAD2 zip file had a Visual C project that compiles and runs the AAC decoder floating point version. In order to have only the AAC decoder functionality, the unnecessary source files associated with aacDECdrop were discarded and only the files which were required for the AAC decoder functionality were used. This required understanding of the program flow, the order in which the functions were called, the header in which they were defined and how the algorithm works. The necessary source and header files were organised in three folders namely faad, libfaad and mp4. This task was time consuming and was very important as this was the starting point.

The AAC decoder FAAD2 code is written in C language and supports both floating point and fixed point arithmetic. FAAD2 was developed keeping the Windows operating system and x86 processor in mind. The code had many "ifdef's" which indicated that it was generic and aimed at supporting multiple platforms. The floating point version of FAAD2 supported all the three profiles (chapter 1.4, Page 4) when run on x86 and other platforms. The fixed point version supported only the LC profile. When FAAD2 was run on Windows OS, it always executed the floating point version. As discussed above, the idea was to run the fixed point version of the code on x86 and port the same to the Eclipse IDE which is the Integrated Development Environment for eSi-RISC processor.

Conversion from Floating point to Fixed point:

To change the algorithm from floating point to fixed point version, there were modifications made to the header file common.h. This file was found in the subfolder "Header files" under the faad folder. For the decoder to function in the fixed point version, "FIXED_POINT" and "BIG_IQ_TABLE" was defined in the header file common.h. "FIXED_POINT" when defined, executed the version with fixed point arithmetic. The "BIG_IQ_TABLE" contained the sine, cosine,

```
#ifndef __COMMON_H__
#define __COMMON_H__

/* COMPILE TIME DEFINITIONS */
/* use fixed point reals */

#define FIXED_POINT
#define BIG_IQ_TABLE

#define LC_ONLY_DECODER  // Define LC_ONLY_DECODER if you want a pure AAC
            //LC decoder (independant of SBR_DEC and PS_DEC)
```

Figure 3.1: Showing the part of the header file, common.h, where FIXED_POINT and BIG_IQ_TABLE are visible.

logarithmic tables, etc. It was found that the fixed point arithmetic version supported only the LC (Low complexity) profile. This was tested decoding a main profile AAC audio file and the decoder could not recognise it. This meant that "LC_ONLY_DECODER" had to be defined so that the decoder used LC profile (Figure 3.1).

Once the decoder was functioning for fixed point version on x86 processor and Windows operating system, the next step was to port the same to the Eclipse environment which is the IDE used for eSi-RISC embedded processor. The Eclipse environment was a customised version for the eSi-RISC processor. The eSi-RISC 3250 core used the eSi-RISC Cygwin cross compiler as the toolchain. The target FPGA could be set for the eSi-RISC 3250 in the simulator. The target FPGA board for this project was chosen to be eSi-3250 Cyclone III Demo which features a 32-bit eSi-3250 CPU with peripherals targeted to the Altera NIOS II Embedded Evaluation kit, Cyclone III Edition.

The first step in porting the decoder algorithm into the Eclipse environment was setting the path for the Header files so that they could be recognised by the Cygwin cross compiler. This setting was made in Project properties, C/C++ Build, Settings, eSi-RISC Cygwin C compiler, Directories and the path "${workspace_loc:/DEcoder_AAC_3250/faad/HeaderFiles}" was included (Figure 3.3). The next step was to set the path for environment variables for the Cygwin compiler. For Windows this was done in My Computer properties, Advanced settings, Environment variables, Path and the path was set as C:\cygwin\bin (Figure 3.4). After setting the path for header files and environment variables the program code was compiled, but the compilation failed with errors. Most of these were related to C language syntax and cross compilation and were fixed with ease. The error due to integer data type declaration was discovered after thorough debugging. It was found that the unsigned and signed 8-bit, 16-bit, 32-bit and 64-bit were declared as unsigned __int8, signed __int8, unsigned __int16, signed __int16, unsigned __int32, signed __int32, unsigned __int64 and signed __int64. They were defined in the header file common.h. It was learnt that these integer declarations were defined in the Windows standard library stdlib.h and were understood by the Windows gcc compilers. For the Eclipse eSi-RISC Cygwin cross compiler to understand these integer declarations, they had to be typecast using the traditional signed and unsigned integer data types which were unsigned char, char, unsigned short, short,

unsigned long, long, unsigned long long and long long for 8, 16, 32 and 64-bits (Figure 3.2).

```
/*

#include <stdlib.h>

typedef unsigned __int64 uint64_t;
typedef unsigned __int32 uint32_t;
typedef unsigned __int16 uint16_t;
typedef unsigned __int8 uint8_t;
typedef signed __int64 int64_t;
typedef signed __int32 int32_t;
typedef signed __int16 int16_t;
typedef signed __int8  int8_t;
typedef float float32_t;
*/
#ifndef __TCS__
typedef unsigned long long uint64_t;
typedef signed long long int64_t;
#else
typedef unsigned long uint64_t;
typedef signed long int64_t;
#endif
typedef unsigned long uint32_t;
typedef unsigned short uint16_t;
typedef unsigned char uint8_t;
typedef long int32_t;
typedef short int16_t;
typedef char int8_t;
```

Figure 3.2: Showing integer data type declarations. The original declarations are shown in comments and modified declarations are shown below them.

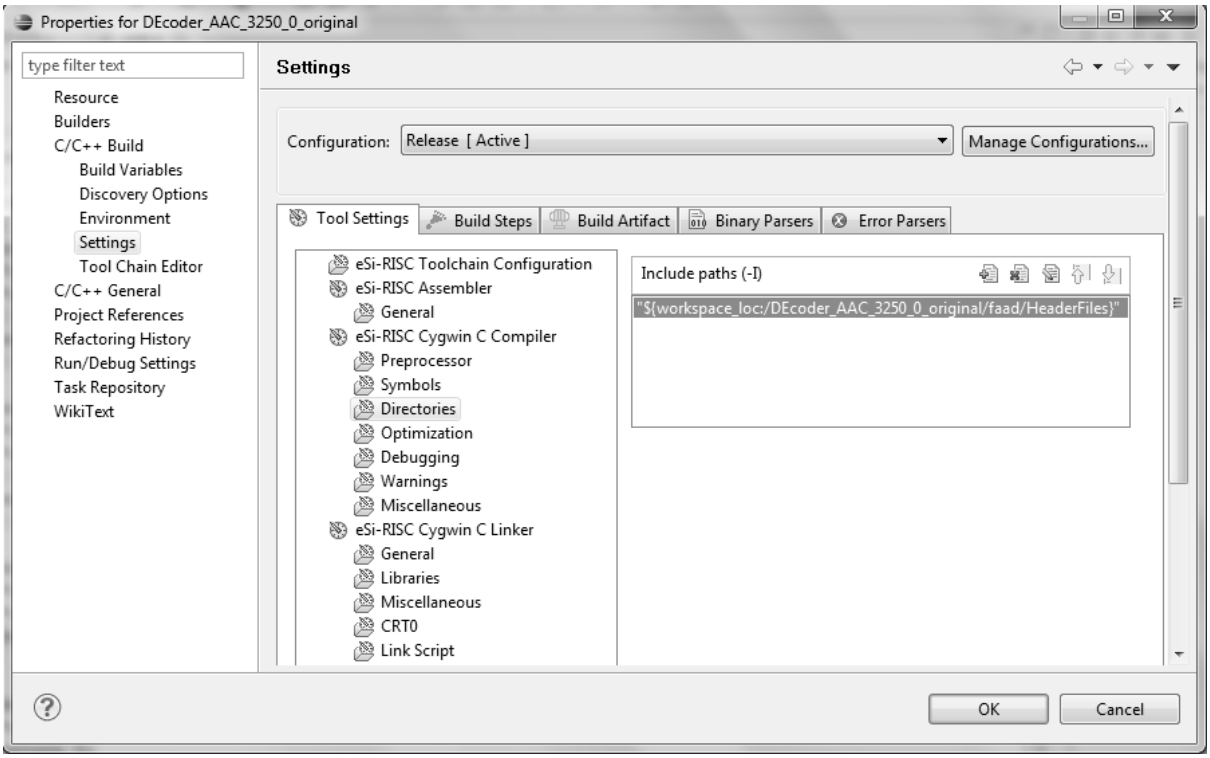


Figure 3.3: Showing the screenshot for setting the path for header files in Eclipse IDE.

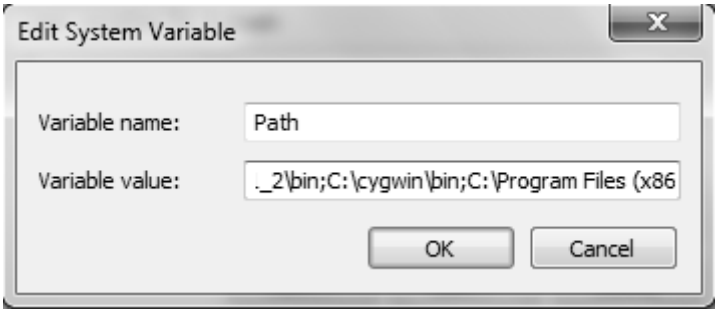


Figure 3.4: Screen shot showing path setting for the environment variables for Cygwin compiler

Once the integer data type declaration was modified for the eSi-RISC Cygwin cross compiler, the program code compilation was successful and a binary executable was generated by the compiler which could be used to run the decoder. Thus the necessary source files were taken and organised, the program code was converted to run the fixed point version, the algorithm was ported to Eclipse IDE with eSi-RISC 3250 as the target processor on the Cyclone III FPGA, necessary tool settings were changed and finally the compilation errors were fixed. The next step was to test the working of the decoder by providing an aac audio file input and perform run time profiling of the code which is discussed in the following chapter.

Chapter 4

Profiling

Investigating the behaviour of the program code with the information gathered during its execution is called Profiling. This chapter discusses profiling of the code for the AAC decoder algorithm. Profiling helps to learn the behaviour of different functions in the program code in terms of the number of instructions executed by each of them, the number of times they were called during the execution, the number of CPU clock cycles consumed, etc. It helps to identify the intensive operations in the program code. Intensive operations are those which either have more number of instructions or consume a lot of CPU cycles. Identifying these computationally intensive operations gives an idea about the dependency of the program code on these operations.

As discussed in the previous chapter, the AAC decoder FAAD2 was compiled successfully for eSi-RISC 3250 Release build with –O2 optimization which gives a good balance between code size and execution speed. A binary executable was generated by the eSi-RISC compiler after successful compilation of the program code. After getting the decoder to function for the eSi-RISC 3250 processor platform, it was necessary to run it on the eSi-RISC simulator by providing an AAC audio file as input. The details of the file input are as follows:

File name : a5.aac

File info : AAC, 17.554sec, 128Kbps, 44100Hz, LC profile.

This file was provided as input to the decoder at runtime by passing it as an argument. Program argument was set for the run configuration by setting the path in the "Arguments" tab. The decoder was run and it took 22.29 sec in real-time (routine in the program uses the timer for "real-time" calculation) to decode a5.aac. In order to view the results of the simulation, the "Output statistics at the end of simulation" option was enabled in the run configuration settings. The statistics obtained from the simulator on running the AAC decoder, FAAD2, is as shown below:

Cycles : 1,114,355,938

Instructions : 953,247,430 (IPC: 0.85)

Prefixed : 505200711 (52%)

Exceptions : 22,287

Decoding a5.aac took 22.29 sec. real-time

eSi-RISC profiler

The eSi-RISC profiler in the simulator instructs the simulator to write a function-level description of the number of instructions and clock cycles of the program it executes. In order to inform the simulator to perform profiling of the code it executes, it is necessary to inform the simulator to run the eSi-RISC profiler. This was done by enabling the output profile option in the run configuration settings. The eSi-RISC profiler wrote the profile of the code in a file which was named as profile.txt. The simulator wrote into profile.txt while it executed the code. A part of the data from profile.txt is shown in Figure 4.1 and the rest of them are shown in Appendix B.

Function	Instructions	Cycles
__udivsi3	174634	274442
__do_global_ctors_aux	12	17
_fseek_r	100	137
__lo0bits	32	38
aac_frame_decode	159461	201044
cfftu	26	40
filter_bank_init	42	50
memcpy	1808580	2322060
tns_decode_frame	66440	72480
__muldf3	2594	2910
__sread	4155	5540
sprintf	3131	3535
NeAACDecDecode	5	7
malloc	36	56
__sfp_lock_acquire	751	2253
__sseek	16	20
_sys_timer_interrupt_handler	522456	674839
__udivmodsi4	3276	5012
_open_r	38	46
__divdf3	236	282
_getargs	24	35
drc_end	7	11
__sclose	7	11
_malloc_r	1539	2011
strlen	78	116
__sfvwrite_r	918	1054
__register_exitproc	31	41
__udivmodsi4	1143227	1759101
faad_free	84	140
fill_buffer	48207	69251
passf4pos	128611714	141178235
_vfprintf_r	3672	5080
_sbrk	60	72
__floatunsidf	54	62
faad_initbits	40771	44550
passf2pos	11762141	13304321
__smakebuf_r	58	70
faad_fprintf	38	48
__malloc_unlock	8	24

Figure 4.1: Showing the contents of profile.txt generated by eSi-RISC profiler.

The results from the eSi-RISC profiler showed that the filterbank operation consumed the maximum ratio of the CPU cycles (Figure 4.2). This was 68% the total cycles which was calculated by adding the cycles taken by functions faad_imdct() and ifilter_bank(). This statistic obtained agreed with the results from published work [18] where the filterbank operation had taken 68.7% ratio of the CPU cycles on the ARM processor (Figure 2.5). Figure 4.2 shows the distribution of the CPU clock cycles among the various functions in the decoder algorithm as obtained from the eSi-RISC profiler.

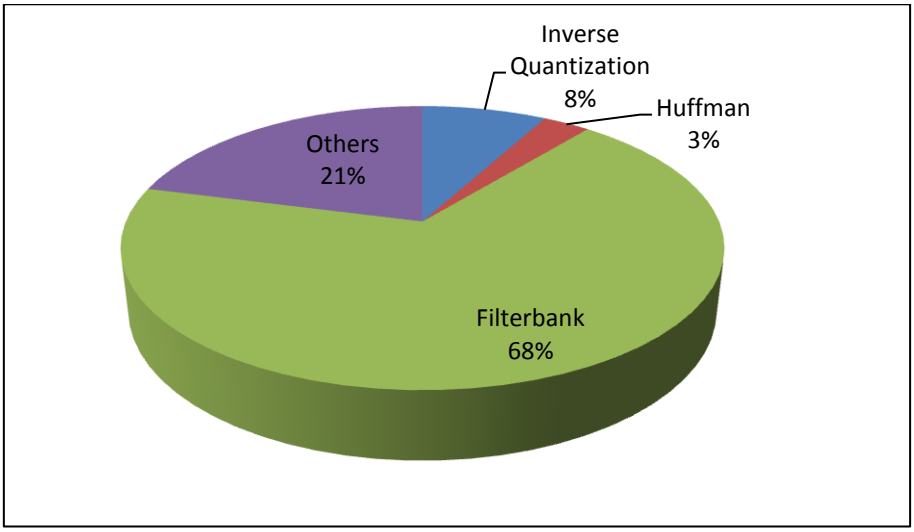


Figure 4.2: Clock cycle distribution as obtained from the eSi-RISC profiler for different functions.

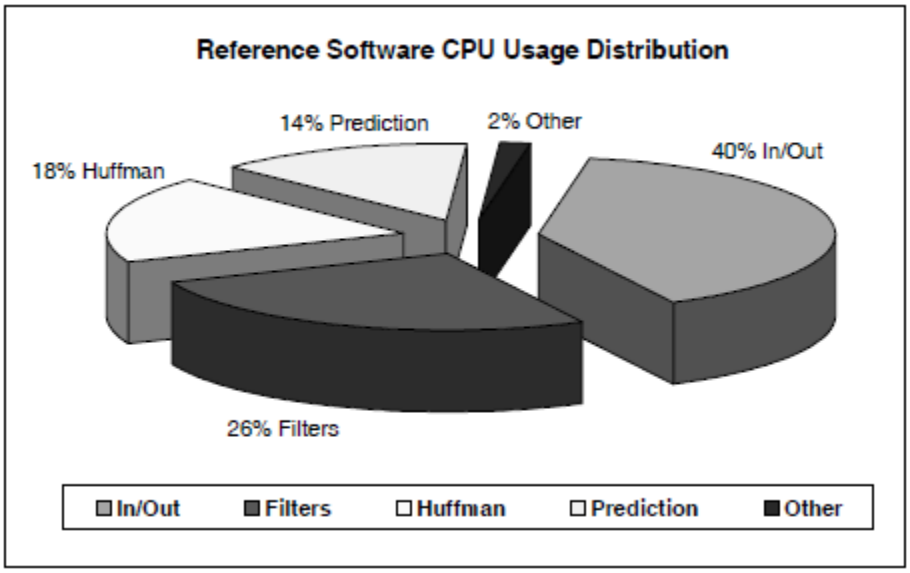


Figure 4.3: CPU usage distribution between MPEG-4 AAC decoding blocks [6].

Results from previously published work considering memory read and writes have given the CPU usage distribution between the decoder blocks as shown in the above pie chart (Figure 4.3). To get an insight into the memory read and write in AAC decoder algorithm extensive profiling was performed using powerful profiling tools in the Linux environment.

The tools used for profiling instruction and data references in the program code were Cachegrind, KCachegrind and Callgrind. These are powerful profiling tools in the Linux environment which perform accurate profiling of the code.

Cachegrind was used first. The commands used for Cachegrind in Linux were,

CFLAGS= "-DFIXED_POINT –g" ./configure

faad –h ../a5.aac

valgrind --tool = cachegrind faad /a5.aac –o /dev/null

The results obtained from Cachegrind were:

```
I   refs:      296,473,347
I1  misses:        113,826
L2i misses:          1,769
I1  miss rate:        0.03%
L2i miss rate:        0.00%
D   refs:      116,582,444  (77,112,513 rd   + 39,469,931 wr)
D1  misses:      1,287,345  (   460,576 rd   +    826,769 wr)
L2d misses:          4,575  (     2,554 rd   +      2,021 wr)
D1  miss rate:         1.1% (       0.5%     +        2.0%  )
L2d miss rate:         0.0% (       0.0%     +        0.0%  )
L2 refs:         1,401,171  (   574,402 rd   +    826,769 wr)
L2 misses:           6,344  (     4,323 rd   +      2,021 wr)
L2 miss rate:          0.0% (       0.0%     +        0.0%  )
```

The sum of the instruction and data references as obtained from Cachegrind is

```
I refs + D refs = 296,473,347 + 116,582,444 = 413,055,791
```

This number accounts to 37% of the total clock cycles 1,114,355,938 and justifies the statistic shown in Figure 4.3 where In/Out is said to have used 40% of the CPU.

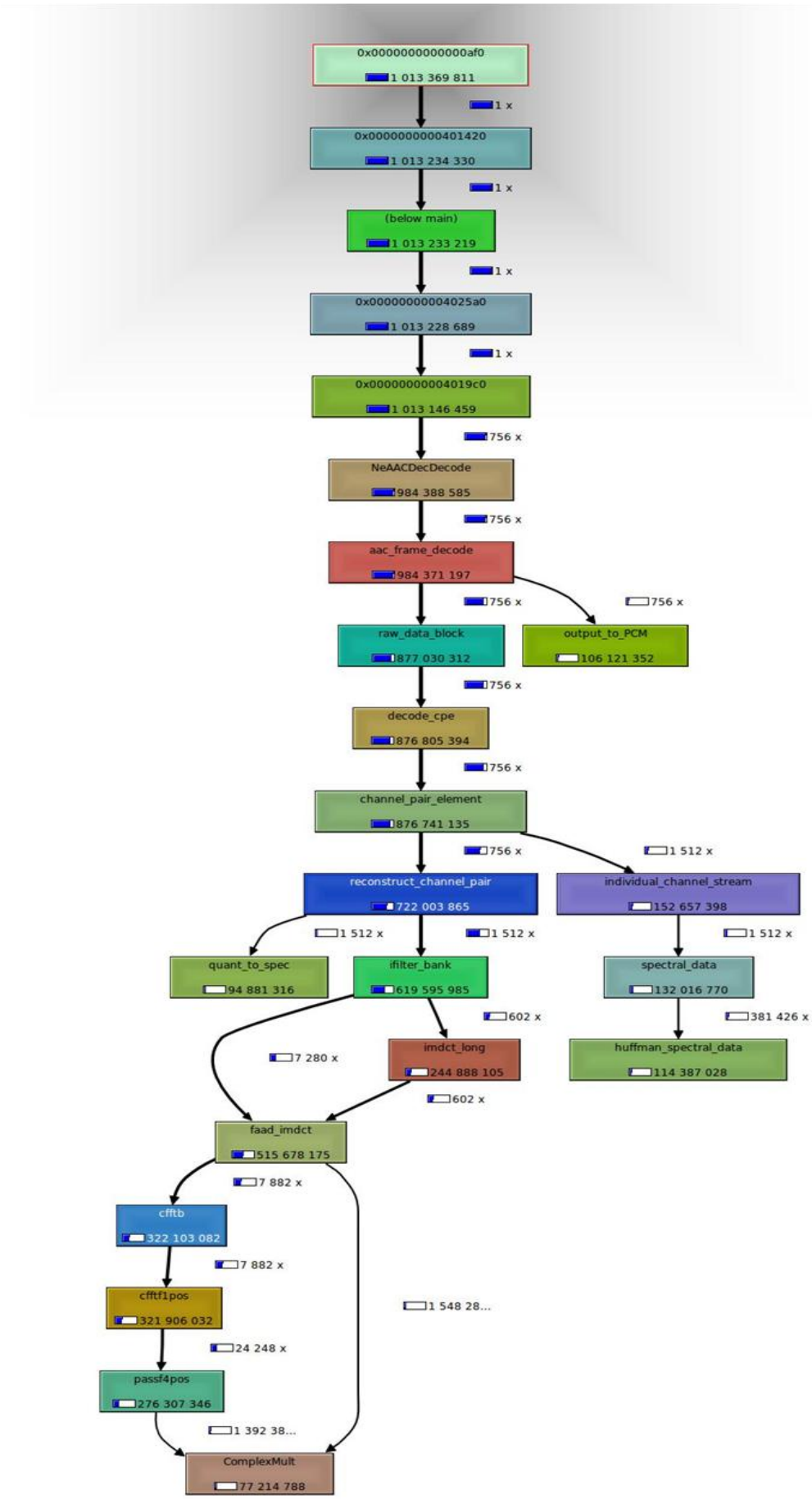
Further profiling was done using the profiling tool Callgrind. Callgrind uses runtime instrumentation via the Valgrind framework for its cache simulation and call-graph generation. The data files generated by Callgrind can be loaded into KCachegrind for browsing the performance results. The commands used for Callgrind in Linux were,

CFLAGS= "-DFIXED_POINT –g" ./configure

faad –h ../a5.aac

valgrind --tool = callgrind faad /a5.aac –o /dev/null

The data file generated by Callgrind was callgrind.out.1702 and this data file was read using the visualisation tool KCachegrind. KCachegrind produces a call graph using this data showing all the function calls during the program execution and the number of clock cycles for each function (Figure 4.4, Page 24). KCachegrind also generates a table listing all the functions and the number of times they were called (Figure 4.5).


0x0000000000000af0
1 013 369 811
1 x
0x0000000000401420
1 013 234 330
1 x
(below main)
1 013 233 219
1 x
0x00000000004025a0
1 013 228 689
1 x
0x00000000004019c0
1 013 146 459
756 x
NeAACDecDecode
984 388 585
756 x
aac_frame_decode
984 371 197
756 x
756 x
raw_data_block
877 030 312
output_to_PCM
106 121 352
756 x
decode_cpe
876 805 394
756 x
channel_pair_element
876 741 135
756 x
1 512 x
reconstruct_channel_pair
722 003 865
individual_channel_stream
152 657 398
1 512 x
1 512 x
1 512 x
quant_to_spec
94 881 316
ifilter_bank
619 595 985
spectral_data
132 016 770
602 x
381 426 x
7 280 x
imdct_long
244 888 105
huffman_spectral_data
114 387 028
602 x
faad_imdct
515 678 175
7 882 x
cfftb
322 103 082
7 882 x
cfftf1pos
321 906 032
1 548 28...
24 248 x
passf4pos
276 307 346
1 392 38...
ComplexMult
77 214 788

Called	Function	Location
756	NeAACDecDecode	libfaad.so.2.0.0: decoder.c
756	aac_frame_decode	libfaad.so.2.0.0: decoder.c
756	raw_data_block	libfaad.so.2.0.0: syntax.c
756	decode_cpe	libfaad.so.2.0.0: syntax.c
756	channel_pair_element	libfaad.so.2.0.0: syntax.c
756	reconstruct_channel_pair	libfaad.so.2.0.0: specrec.c
1 512	ifilter_bank	libfaad.so.2.0.0: filtbank.c
7 882	faad_imdct	libfaad.so.2.0.0: mdct.c
7 882	cfftb	libfaad.so.2.0.0: cfft.c
7 882	cfftf1pos	libfaad.so.2.0.0: cfft.c
24 248	passf4pos	libfaad.so.2.0.0: cfft.c
602	imdct_long	libfaad.so.2.0.0: filtbank.c
3 094 784	ComplexMult	libfaad.so.2.0.0: fixed.h
1 512	individual_channel_stream	libfaad.so.2.0.0: syntax.c
1 512	spectral_data	libfaad.so.2.0.0: syntax.c
381 426	huffman_spectral_data	libfaad.so.2.0.0: huffman.c
756	output_to_PCM	libfaad.so.2.0.0: output.c
1 512	quant_to_spec	libfaad.so.2.0.0: specrec.c
1 548 288	get_sample	libfaad.so.2.0.0: output.c
1 548 288	iquant	libfaad.so.2.0.0: specrec.c
153 704	huffman_2step_pair	libfaad.so.2.0.0: huffman.c
602	passf2pos	libfaad.so.2.0.0: cfft.c
131 332	huffman_2step_quad	libfaad.so.2.0.0: huffman.c
697 794	faad_get1bit	libfaad.so.2.0.0: bits.h
70 132	huffman_2step_pair_sign	libfaad.so.2.0.0: huffman.c
755	0x0000000000404060	faad
1 512	side_info	libfaad.so.2.0.0: syntax.c
62 704	huffman_binary_pair	libfaad.so.2.0.0: huffman.c
150 225	huffman_sign_bits	libfaad.so.2.0.0: huffman.c
449 793	faad_flushbits	libfaad.so.2.0.0: bits.h
450 652	faad_showbits	libfaad.so.2.0.0: bits.h
32 579	huffman_2step_quad_sign	libfaad.so.2.0.0: huffman.c
33 686	huffman_binary_quad_sign	libfaad.so.2.0.0: huffman.c
1 512	scale_factor_data	libfaad.so.2.0.0: syntax.c

Figure 4.5: Table generated by KCachegrind using data from Callgrind. Column 1: Number of times the function was called. Column 2: Function. Column 3: The source of the function. Continued in Appendix A.

ComplexMult() was found to be a filterbank function and this again proved that the filterbank had the intensive operations and consumed more clock cycles as learnt earlier. To summarise decoder algorithm profiling was done to analyse the CPU clock cycles, program flow and number of function calls. The next step after profiling was to optimise the algorithm using the results from profiling.

Chapter 5

Optimisation

The results from the eSi-RISC profiler indicated where the compute intensive operations were present in the code. The next step was to optimise those operations so as to increase the execution speed by decreasing the overall clock cycles. This chapter discusses how optimisation was carried out and the methods used. The eSi-RISC profiler showed the functions with the number of instructions executed and clock cycles taken. The function with the most number of cycles taken for execution was optimised first followed by the others. The chapter also discusses how user-defined instructions were used for optimisation and their impact on the algorithm execution speed. The performance of the decoder is tested after every stage of optimisation and the results are compared. All the optimisation results are later put together and are discussed.

As discussed earlier, the AAC decoder algorithm which is a C language implementation was compiled for a Release build with –O2 optimization which gives a good balance between code size and execution speed. The eSi-RISC profiler showed that the function faad_imdct() consumed massive amount of the CPU cycles (Figure 5.1). Infact the function __muldi3 consumed more cycles, but, it did not belong to the source code. It was called from the faad_imdct() function and was defined in the libgcc library. It was executed by the gcc compiler for multiplication and shifting operations and was computationally intensive.

faad_malloc	12	20
__muldi3	403967998	466055158
memmove	17876286	22998816
.boot	1	3
__nesf2	19	21
_close_r	85	105
_write	143	165
faad_imdct	103490788	119362446
__malloc_lock	8	24
__fixunssfsi	68	84

Figure 5.1: Results obtained from the eSi-RISC profiler. Second column lists the number of instructions and the third column lists the number of clock cycles.

The function faad_imdct() was defined in the source file mdct.c. It was learnt that the function carried out complex multiplications (Figure 5.2). These included pre-IFFT complex multiplication, complex IFFT and post-IFFT complex multiplication. All of these complex multiplications had sub-function calls to a common function called ComplexMult() which was defined in the header file fixed.h. Figure 5.3 shows the definition of ComplexMult() function.

```
void faad_imdct(mdct_info *mdct, real_t *X_in, real_t *X_out)
{
   uint16_t k;
   complex_t x;
   ALIGN complex_t Z1[512];
   complex_t *sincos = mdct->sincos;

   uint16_t N  = mdct->N;
   uint16_t N2 = N >> 1;
   uint16_t N4 = N >> 2;
   uint16_t N8 = N >> 3;

  /* pre-IFFT complex multiplication */
   for (k = 0; k < N4; k++)
   {
       ComplexMult(&IM(Z1[k]), &RE(Z1[k]),
           X_in[2*k], X_in[N2 - 1 - 2*k], RE(sincos[k]), IM(sincos[k]));
   }

   /* complex IFFT, any non-scaling FFT can be used here */
   cfftb(mdct->cfft, Z1);

   /* post-IFFT complex multiplication */
   for (k = 0; k < N4; k++)
   {
       RE(x) = RE(Z1[k]);
       IM(x) = IM(Z1[k]);
       ComplexMult(&IM(Z1[k]), &RE(Z1[k]),

           IM(x), RE(x), RE(sincos[k]), IM(sincos[k]));
```

Figure 5.2: Definition of the function faad_imdct().

```
/* Complex multiplication */
static INLINE void ComplexMult(real_t *y1, real_t *y2,
  real_t x1, real_t x2, real_t c1, real_t c2)
{
  *y1 = (_MulHigh(x1, c1) + _MulHigh(x2, c2))<<(FRAC_SIZE-FRAC_BITS);
  *y2 = (_MulHigh(x2, c1) - _MulHigh(x1, c2))<<(FRAC_SIZE-FRAC_BITS);
}
```

Figure 5.3: Definition of the ComplexMult() function as defined in fixed.h.

The definition showed that the function takes 32-bit signed integers as arguments and performed operations on them. _MulHigh(A,B) was a macro defined as

```
#define _MulHigh(A,B) (real_t)(((int64_t)(A)*(int64_t)(B)+(1 << (FRAC_SIZE-1))) >> FRAC_SIZE)
```

The inputs A and B were 32-bit signed integers. The macro performed 32-bit multiplication on fixed point integers A and B. A and B were type cast as 64 bit signed integers so as to include the sign bits for negative integers. 32-bit fixed point multiplication has been performed on A and B to obtain a 64-bit result. The constant `FRAC_SIZE` had been defined to a value 32. Thus the operation `(1 << (FRAC_SIZE-1))` performed rounding off on the product.

The 64-bit product had lower 32-bit fractional part and upper 32-bit integral part. `(1 << (FRAC_SIZE-1))` was equivalent to adding 0.5 (0x80000000 or 10000000000000000000000000000000) to the fractional part. If the fractional part was greater than 0.5, adding 0.5 to it rounded off the integral part to a value close to the nearest integer. And finally `>> FRAC_SIZE` right shifted out the lower 32-bit fractional part and left was the 32-bit result which was a fixed point integer.

The generated assembly code for the above described operation by the eSi-RISC 3250 compiler was as follows

```
4002ad0:   7e91        sw    (sp+[17]), r13
4002ad2:   e017 da6a   call  4031fa6 <__muldi3>
4002ad6:   e808 3c18   add   r8, r8, r24
4002ada:   e02f 3cf3   addc  r15, r9, r3
4002ade:   f40e 368f   sr    r14, r29, 31
4002ae2:   6494        lw    r9, (sp+[20])
4002ae4:   3dee        mv    r11, r14
4002ae6:   e840 3d6d   mv    r10, r29
4002aea:   e840 3c6b   mv    r8, r27
4002aee:   7f13        sw    (sp+[19]), r14
4002af0:   7f92        sw    (sp+[18]), r15
4002af2:   e017 da5a   call  4031fa6 <__muldi3>
4002af6:   e808 3c18   add   r8, r8, r24
4002afa:   6792        lw    r15, (sp+[18])
4002afc:   e029 3cf3   addc  r9, r9, r3
4002b00:   3f99        add   r15, r15, r9
4002b02:   37a1        sl    r15, r15, 1
4002b04:   bf84        sw    (r4+[0]), r15
4002b06:   e840 3d6d   mv    r10, r29
4002b0a:   6713        lw    r14, (sp+[19])
4002b0c:   6691        lw    r13, (sp+[17])
4002b0e:   3dee        mv    r11, r14
4002b10:   e840 3c6c   mv    r8, r28
4002b14:   3ced        mv    r9, r13
4002b16:   e017 da48   call  4031fa6 <__muldi3>
4002b1a:   e808 3c18   add   r8, r8, r24
4002b1e:   6610        lw    r12, (sp+[16])
4002b20:   e03c 3cf3   addc  r28, r9, r3
4002b24:   6494        lw    r9, (sp+[20])
4002b26:   e840 3c6b   mv    r8, r27
4002b2a:   e840 3d6a   mv    r10, r26
4002b2e:   3dec        mv    r11, r12
4002b30:   e017 da3b   call  4031fa6 <__muldi3>
4002b34:   e808 3c18   add   r8, r8, r24
```

The assembly listing for eSi-3250 for the function ComplexMult() showed four calls to the __muldi3 library (one for each _MulHigh() function as seen in Figure 5.3). __muldi3 was computationally intensive and this operation directly resulted in a high cycle count for the execution of faad_imdct() function.

The next step was to write an efficient algorithm in C language that implemented the _MulHigh() function and prevented the eSi-RISC compiler from executing the library function __muldi3(). This written algorithm is shown in Figure 5.5.

```
/* Complex multiplication */
static INLINE void ComplexMult(real_t *y1, real_t *y2,
    real_t x1, real_t x2, real_t c1, real_t c2)
{

    *y1 = (long_multiply(x1, c1) + long_multiply(x2, c2))<<(FRAC_SIZE-
  FRAC_BITS);

    *y2 = (long_multiply(x2, c1) - long_multiply(x1, c2))<<(FRAC_SIZE-
  FRAC_BITS);
}
```

Figure 5.4: Showing how _MulHigh() is replaced by long_multiply in ComplexMult.

```
real_t long_multiply (real_t v1, real_t v2)
{
  register long a, b, c, d;
  register long x, y;
  register long LO, HI;

  char sign = -1;      //variable to keep track negative integer result

  if(v1 < 0)           //perform positive multiplication on negative int.
   {
       v1 = (~(v1))+1;
       sign = ~(sign);
   }
  if(v2 < 0)          //perform positive multiplication on negative int.
   {
       v2 = (~(v2))+1;
       sign = ~(sign);
   }

  a = (v1 >> 16) & 0xffff; //Get the higher 16 bits of first operand
  b = v1 & 0xffff;         //Get the lower 16 bits of first operand
  c = (v2 >> 16) & 0xffff; //Get the higher 16 bits of second operand
  d = v2 & 0xffff;         //Get the lower 16 bits of second operand
 LO = b * d;
  if (LO > 0xFFFFFFFF)     //check for overflow
   {
       LO = 0xFFFFFFFF;    //saturate result of lower word multiplication
   }
  x = a * d + c * b;
  y = ((LO >> 16) & 0xffff) + x;

  LO = ((LO & 0xffff) | ((y & 0xffff) << 16))*(-sign);
  HI = (y >> 16) & 0xffff;
  HI += a * c;

  if (HI > 0xFFFFFFFF)
    {
       HI = 0xFFFFFFFF;   //saturate result of higher word multiplication
    }
  if ((LO > 0x80000000))
    {
       HI = HI + 1; //if lower fractional part is greater than 0.5
    }              //round-off by adding 1 to the higher 32bits of result
  HI = ((HI >> 32)*(-sign)); //preserve only the higher 32bits of
  return HI;                 //the 64-bit result
}
```

Figure 5.5: Algorithm in C to implement the _MulHigh() function.

The function long_multiply() prevented the eSi-RISC compiler from executing the inefficient __muldi3() function from the libgcc library. Execution of the function long_multiply() showed a very minor improvement in the overall number of CPU clock cycles. This improvement was not significant and a better optimisation technique was necessary.

Optimisation using user-defined instructions

eSi-RISC 3250 allows to write user-defined instructions in the program code. It was interesting to see if the function _MulHigh() could be accelerated by a user-defined instruction. On eSi-RISC 3250 user defined instructions are implemented as small hardware accelerators attached to the main processor and able to access the register file up to 2 input arguments and 1 output argument.

The user-defined instructions have a "C" subroutine call prototype for simple integration into the code. The actual assembly language instruction is inlined after the arguments are resolved to avoid the overhead of an actual function call. The following figure (Figure 5.6) shows how a user-defined instruction can be called instead of the _MulHigh() function inside the ComplexMult subroutine.

```
/* Complex multiplication */
static INLINE void ComplexMult(real_t *y1, real_t *y2,
    real_t x1, real_t x2, real_t c1, real_t c2)
{

    *y1 = (esi_user0(x1, c1) + esi_user0(x2, c2))<<(FRAC_SIZE-
  FRAC_BITS);

    *y2 = (esi_user0(x2, c1) - esi_user0(x1, c2))<<(FRAC_SIZE-
  FRAC_BITS);
}
```

Figure 5.6: Subroutine ComplexMult accelerated by user-defined instruction esi_user0.

The generated assembly code is given below and the user-defined instruction is clearly seen.

```
400295a:   f014 3a24   sub    r20, r20, r4
400295e:   f000 a31e   lw     r22, (r14+[1])
4002962:   a21d        lw     r4, (r13+[1])
4002964:   f000 a29f   lw     r21, (r15+[1])
4002968:   f000 a38f   lw     r23, (r15+[0])
400296c:   f016 3b24   sub    r22, r22, r4
4002970:   f8e4 3a07   user0  r4, r20, r23
4002974:   f8e5 3b05   user0  r5, r22, r21
4002978:   e004 3a94   add    r4, r5, r4
400297c:   3221        sl     r4, r4, 1
400297e:   e800 ba01   sw     (r17+[0]), r4
4002982:   f8f6 3b07   user0  r22, r22, r23
4002986:   f8f4 3a05   user0  r20, r20, r21
400298a:   f814 3b24   sub    r20, r22, r20
400298e:   f014 3221   sl     r20, r20, 1
4002992:   f000 c101   add    r18, 1
4002996:   f800 ba13   sw     (r19+[1]), r20
400299a:   f3ff c17f   and    r18, 0xffff
400299e:   e840 3c72   cmp    r8, r18
```

The operations defined by the user-defined instructions will correspond to efficient hardware taking only a fewer number of gates. This gives acceleration to the algorithm execution and hence the application. The eSi-RISC profiler was then used again to see the impact of the user-defined instruction on the function faad_imdct() and __muldi3(). The figure below shows the results as displayed by the eSi-RISC profiler.

fill_buffer	32	40
__muldi3	92642859	107023789
__floatsidf	368	416
cfftu	26	40
__sflags	19	28
_fstat_r	54	66
tns_decode_frame	66441	72485
__muldf3	2579	2893
faad_imdct	58163899	69375174
sprintf	3131	3535

Figure 5.7: Results displayed by the eSi-RISC profiler after the use of the user-defined instruction esi_user0. First column lists the number of instructions and the second column lists the number of clock cycles.

There was a difference in the number of clock cycles taken to execute the function faad_imdct() after using esi_user0 (Figure 5.1 and Figure 5.7). There was also a significant reduction in the cycles for the function __muldi3() as expected.

Thus the function faad_imdct() was optimised using the user-defined instruction esi_user0. The eSi-RISC profiler also showed that the function ifilter_bank() is the next most computationally intensive function after faad_imdct() (Figure 5.8).

_times	17	21
ifilter_bank	55437763	64908448
__lesf2	128	186
__sfp_lock_acquire	2	6
_read_r	95	115
_fread_r	147	201
NeAACDecClose	1119	1766
__sfp_lock_release	2	6
__srefill_r	12967	17025
_calloc_r	96	116

Figure 5.8: Results obtained from the eSi-RISC profiler. Second column lists the number of instructions and the third column lists the number of clock cycles.

The definition of this function ifilter_bank() was seen in the source file filterbank.c. This function took in window sequence, window shape, frame length, etc as arguments and performed filterbank operations on them. Based on the window sequence, the second half output of the previous frame was added to the windowed output of the current frame. The second half of the current frame was then windowed and saved as an overlap for the next frame. This window operation was again a macro named MUL_F.

The ifilter_bank() function is shown in the figure below.

```
  case LONG_START_SEQUENCE:
       /* perform iMDCT */
       imdct_long(fb, freq_in, transf_buf, 2*nlong);

        /* add second half output of previous frame to windowed output of
                              current frame */
       for (i = 0; i < nlong; i+=4)
       {

time_out[i] = overlap[i]   + MUL_F(transf_buf[i],window_long_prev[i]);
time_out[i+1]=overlap[i+1] + MUL_F(transf_buf[i+1],window_long_prev[i+1]);
time_out[i+2]=overlap[i+2] + MUL_F(transf_buf[i+2],window_long_prev[i+2]);
time_out[i+3]=overlap[i+3] + MUL_F(transf_buf[i+3],window_long_prev[i+3]);
       }

       /* window the second half and save as overlap for next frame */
       /* construct second half window using padding with 1's and 0's */
       for (i = 0; i < nflat_ls; i++)
           overlap[i] = transf_buf[nlong+i];
       for (i = 0; i < nshort; i++)

overlap[nflat_ls+i]=MUL_F(transf_buf[nlong+nflat_ls+i],window_short[nshort-
                                    i-1]);
       for (i = 0; i < nflat_ls; i++)
           overlap[nflat_ls+nshort+i] = 0;
       break;
```

Figure 5.9: Showing a part of the function ifilter_bank() inside a case statement and the MUL_F function.

The window operation was defined by the macro MUL_F and its definition was

```
#define MUL_F(A,B) (real_t)(((int64_t)(A)*(int64_t)(B)+(1 << (FRAC_BITS-1))) >> FRAC_BITS)
```

FRAC_BITS was a constant which held a value of 31.

The inputs A and B were 32-bit signed integers. The macro performed 32-bit multiplication on fixed point integers A and B. A and B were type cast as 64 bit signed integers so as to include the sign bits for negative integers. 32-bit fixed point multiplication has been performed on A and B to obtain a 64-bit result. The constant `FRAC_BITS` held a value 31 and the operation `(1 << (FRAC_SIZE-1)))` performed rounding off on the product. The 64-bit product had lower 31-bit fractional part and upper 33-bit integral part. `(1 << (FRAC_SIZE-1))` was equivalent to adding 0.5 (0x80000000 or 10000000000000000000000000000000) to the fractional part. If the fractional part was greater than 0.5, adding 0.5 to it rounded off the integral part to a value close to the nearest integer. And finally (`>> FRAC_SIZE`) right shifted out the lower 31-bit fractional part and left was the 32-bit result which was a fixed point integer.

The assembly listing of the function ifilter_bank showed calls made to the libgcc function __muldi3. This is shown as follows:

```
400a170:   3d62        mv    r10, r2
400a172:   e000 dbd8   call  400b922 <faad_imdct>
400a176:   22e2        bz    r5, 400a13a <ifilter_bank+0xcc>
400a178:   1380        l     r7, 0
400a17a:   1200        l     r4, 0
400a17c:   fa80 cc00   lhi   r24, 16384
400a180:   1180        l     r3, 0
400a182:   e019 3222   sl    r25, r4, 2
400a186:   e009 33a2   sl    r9, r7, 2
400a18a:   e009 3919   add   r9, r2, r9
400a18e:   f808 3d19   add   r8, r26, r25
400a192:   a509        lw    r10, (r9+[0])
400a194:   a408        lw    r8, (r8+[0])
400a196:   e40b 350f   sr    r11, r10, 31
400a19a:   e409 340f   sr    r9, r8, 31
400a19e:   e013 df04   call  4031fa6 <__muldi3>
400a1a2:   f80a 3d99   add   r10, r27, r25
400a1a6:   e808 3c18   add   r8, r8, r24
400a1aa:   e029 3cf3   addc  r9, r9, r3
```

The function MUL_F could be accelerated by a user-defined instruction. The following figure shows how a user-defined instruction was called instead of the MUL_F function inside the ifilter_bank() subroutine.

```
    case LONG_START_SEQUENCE:
        /* perform iMDCT */
        imdct_long(fb, freq_in, transf_buf, 2*nlong);

     /* add second half output of previous frame to windowed output of
                              current frame */
        for (i = 0; i < nlong; i+=4)
        {
time_out[i] =   overlap[i]+esi_user1(transf_buf[i],window_long_prev[i]);

time_out[i+1]=overlap[i+1]+esi_user1(transf_buf[i+1],window_long_prev[i+1])
;
time_out[i+2]=overlap[i+2]+esi_user1(transf_buf[i+2],window_long_prev[i+2])
;
time_out[i+3]=overlap[i+3]+esi_user1(transf_buf[i+3],window_long_prev[i+3])
;
        }

        /* window the second half and save as overlap for next frame */
        /* construct second half window using padding with 1's and 0's */
        for (i = 0; i < nflat_ls; i++)
            overlap[i] = transf_buf[nlong+i];
        for (i = 0; i < nshort; i++)
overlap[nflat_ls+i]=esi+user1(transf_buf[nlong+nflat_ls+i],window_short[nsh
                                  ort-i-1]);
        for (i = 0; i < nflat_ls; i++)
            overlap[nflat_ls+nshort+i] = 0;
        break;
```

Figure 5.10: Function ifilter_bank() accelerated using user-defined instruction esi_user1.

The assembly code that was generated is shown as follows where the user-defined instruction was clearly visible.

```
4005a78:   e840 3d69   mv     r10, r25
4005a7c:   e000 da0e   call   4006e98 <faad_imdct>
4005a80:   2261        bz     r4, 4005a42 <ifilter_bank+0xc8>
4005a82:   e010 642e   lw     r8, (sp+[2094])
4005a86:   1500        l      r10, 0
4005a88:   3ce8        mv     r9, r8
4005a8a:   a609        lw     r12, (r9+[0])
4005a8c:   e800 a589   lw     r11, (r25+[0])
4005a90:   e800 a688   lw     r13, (r24+[0])
4005a94:   e0eb 3d9d   user1  r11, r11, r13
4005a98:   3d9c        add    r11, r11, r12
4005a9a:   bd86        sw     (r6+[0]), r11
4005a9c:   a619        lw     r12, (r9+[1])
4005a9e:   e800 a599   lw     r11, (r25+[1])
4005aa2:   e800 a698   lw     r13, (r24+[1])
4005aa6:   e0eb 3d9d   user1  r11, r11, r13
4005aaa:   3d9c        add    r11, r11, r12
4005aac:   bd96        sw     (r6+[1]), r11
4005aae:   a629        lw     r12, (r9+[2])
4005ab0:   e800 a5a9   lw     r11, (r25+[2])
4005ab4:   e800 a6a8   lw     r13, (r24+[2])
4005ab8:   e0eb 3d9d   user1  r11, r11, r13
4005abc:   3d9c        add    r11, r11, r12
4005abe:   bda6        sw     (r6+[2]), r11
```

The eSi-RISC profiler was used again to see the impact of the user-defined instruction on the function ifilter_bank() and __muldi3(). The figure below shows the results as displayed by the eSi-RISC profiler.

```
fread                          11162        14146
ifilter_bank                   41561681     50631622
faad_mdct_end                  26           38
_write                         143          165
_free_r                        30776        40196
__muldi3                       41457568     47937952
aac_frame_decode               81           95
_dtoa_r                        405          536
```

Figure 5.11: Results displayed by the eSi-RISC profiler after the use of the user-defined instruction esi_user1. First column lists the number of instructions and the second column lists the number of clock cycles.

These results when compared with the results in Figure 5.8 showed a difference in the number of clock cycles taken to execute the function ifilter_bank(). There was also a significant reduction in the cycles for the function __muldi3() which means that it was not called from the libgcc library by the ifilter_bank() function.

Thus the function ifilter_bank() was optimised using the user-defined instruction esi_user1. The eSi-RISC profiler then showed that the function quant_to_spec() is as computationally intensive as the function ifilter_bank(). This is shown in the figure that follows.

_fflush_r	36466	50897
quant_to_spec	56342385	67828496
_write	182	210
faad_fprintf	190	240
_sbrk_r	102	126
malloc	36	56
_localeconv_r	6	10
_fread_r	147	201
_sbrk_r	102	126
memcpy	1812499	2326591

Figure 5.12: Results obtained from the eSi-RISC profiler. Second column lists the number of instructions and the third column lists the number of clock cycles.

The function quant_to spec() was found in the source file specrec.c. This performed Spectral reconstruction i.e grouping or sectioning, inverse quantization and applying scale factors. This function in particular performed dequantisation and scaling and in case of short block it also performed the deinterleaving. The function quant_to_spec() in specrec.c is shown in Appendix C. Again the macro MUL_C which was used to perform signed multiplication was found out to be an intensive operation. The definition of the function MUL_C is shown below.

```
#define MUL_C(A,B) (real_t)(((int64_t)(A)*(int64_t)(B)+(1 << (COEF_BITS-1))) >> COEF_BITS)
```

COEF_BITS was a constant which held a value of 28. The assembly listing of the function quant_to_spec() showed calls made to the libgcc function __muldi3 and this is shown in Appendix C. The function MUL_C was accelerated by a user-defined instruction. The figure in Appendix C shows how a user-defined instruction was called instead of the MUL_C function inside the quan_to_spec() subroutine. The generated assembly code after the inclusion of the user-defined instruction showed how the user-defined instruction was clearly visible. The eSi-RISC profiler was used again to see the impact of the user-defined instruction on the function quant_to_spec() and __muldi3(). The figure below shows the results as displayed by the eSi-RISC profiler.

faad_malloc	6	10
quant_to_spec	45282394	56955617
__call_exitprocs	85	113
fclose	45	70
_malloc_r	70503	94468
write_audio_file	13964069	17065475
memcmp	46811	69465
fwrite	11356	14388

Figure 5.13: Results displayed by the eSi-RISC profiler after the use of the user-defined instruction esi_user2.

After using esi_user2, it was found out that there were no more calls to the function __muldi3 which means that the function was not at all called from the libgcc library and it was previously called only by the functions faad_imdct(), ifilter_bank() and quant_to_spec().

Implementing user-defined instructions in the eSi-RISC Simulator

Most of the user-defined instructions have corresponding inline functions provided in the header file esirisc/user.h. This allows the instructions to be used as though they were functions, but with zero overhead when optimization is enabled.

In order to be able to use user-defined instructions when executing an application on the instruction set simulator, a shared library was loaded in to the simulator which implemented the instructions. This library can implement any functionality for the instruction it chooses, even by connecting to real hardware. It was first necessary to build the shared library that implemented the instructions. This library was built using the host's toolchain (i.e. Cygwin GCC). The following template shows how a userX() call was mapped to the resolving function "user_insn()" in the shared library.

```
#include <stdint.h>


/* Execute a user-defined instruction. Which operands are valid will depend
upon the opcode. */
unsigned user_insn(void *config, unsigned opcode, signed operand_a, signed
            operand_b, unsigned *unsupported, unsigned *cycles)
{

switch (opcode)
{
case 0:
*cycles = 1;
*unsupported = 0;

return (long)(((long long)(operand_a)*(long long)(operand_b)+(1 << (31)))
                           >> 32);  //esi_user0

case 1:
*cycles = 1;
*unsupported = 0;

return (long)(((long long)(operand_a)*(long long)(operand_b)+(1 << (30)))
                            >> 31); //esi_user1

case 2:
*cycles = 1;
*unsupported = 0;

return (long)(((long long)(operand_a)*(long long)(operand_b)+(1 << (27)))
                            >> 28); //esi_user2

default:
*unsupported = 1;
return 0;
}
}
```

Figure 5.14: Shared library implementation of the user-defined instructions esi_user1, esi_user2 and esi_user3.

The definition of the user-defined instruction was chosen by the opcode. The opcode was the switching parameter in the switch case statement. case 0 was executed if the user-defined instruction was user0(esi_user0), case 1 was executed if the user-defined instruction was user1(esi_user1) and case 2 was executed if the user-defined instruction was user2(esi_user2). The cycles could be defined for each user-defined instruction. The optimisation achieved so far was carried out for single cycle execution of the user-defined instruction and their hardware implementation will have sufficient logic for the same. Hence "*cycles" was assigned a value 1 (single cycle execution) which is seen in Figure 5.14. "unsupported" indicated if the user-defined instruction in the program was undefined. This explains the implementation of the user-defined instructions in the eSi-RISC simulator.

The statistics output at the end of the simulation after optimising the AAC Decoder algorithm using all the three user-defined instructions were as follows:

Cycles : 489426206

Instructions : 401745577 (IPC: 0.82)

Prefixed : 182545072 (45%)

Exceptions : 9788

Decoding a5.aac took 9.78 sec. real-time

The results of optimisation were plotted in a graph against the clock cycles (Figure 5.15). The overall clock cycles consumption decreased every time a user-defined instruction was added to the program code. The last column in the graph does not have the function __muldi3 which explains the elimination of the function by the user-defined instructions as explained in Page 35.

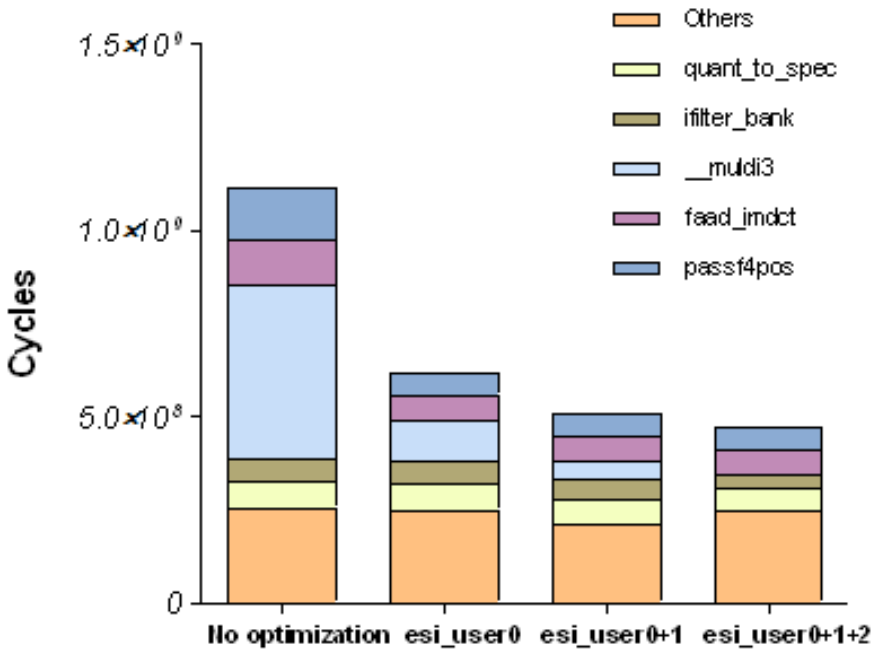


Figure 5.15: Function level distribution of clock cycles during optimisation

To summarise, the 32-bit fixed point multiplication was found to be the most compute-intensive operation in the functions faad_imdct(), ifilter_bank() and quant_to_spec(). A short algorithm developed in C language to implement this multiplication was not as effective in improving the cycle count and the user-defined instructions esi_user0, esi_user1 and esi_user2 were used to achieve optimisation and the overall cycle count was decreased by 56%.

Chapter 6

Hardware implementation of the user-defined instructions

After observing the significant impact of the user-defined instructions in the simulator, the next step was to implement them in hardware using a Hardware Description Language and connect it to the eSi-RISC processor "User Interface". This chapter discusses how the user-defined instructions were modelled in hardware.

The HDL chosen was Verilog. Based on the definition of the user instructions, it was clear that the functions required to be performed are multiplication and shifting. The eSi-RISC architecture has got a 32 bit multiplier to perform both signed and unsigned multiplications and also a Barrel shifter to do the shifting operations. Hence there was no necessity to write the function in Verilog as a structural description of the hardware for the multiplier and shifter, which is also called as Structural modeling. Thus Behavioural model was used and the function in Verilog was written as a behavioural description.

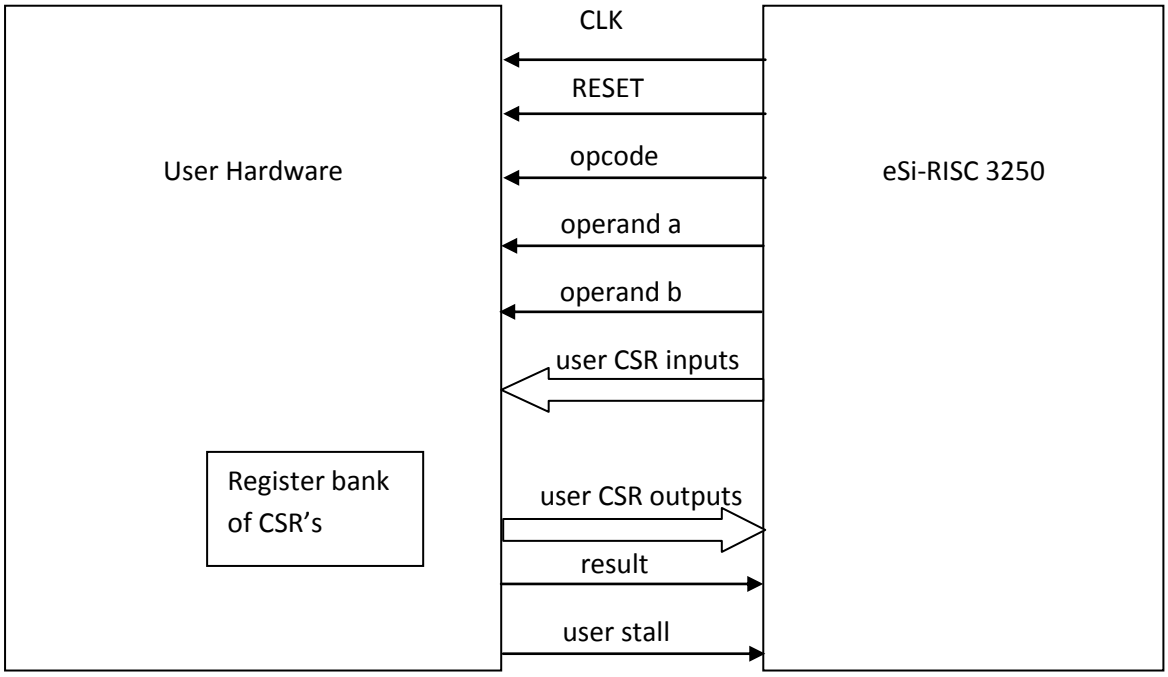


Figure 6.1: Showing the eSi-RISC 3250 and user hardware interface and signals.

On eSi-RISC 3250 user defined instructions are implemented in small hardware accelerators attached to the main processor and able to access the register file up to 2 input arguments and 1 output argument. The user-defined instruction hardware module had input signals like clock, reset, user opcode, user operand_a and user operand_b. It also had a number register bank of control and status registers. The input signals to the module were user_csr_write, user_csr_read, user_csr_bank, user_csr_csr, and user_csr_write_data. The output signals to the user-defined instruction hardware module were user_condition_met,user_stall,user_opcode_unsupported,user_csr_unsupported, user_csr_read_data and user_result (Figure 6.1). The input ports user_csr_write and user_csr_read indicate whether a read or write operation has to be performed on the user control and status register. The port user_csr_bank selectes a register bank from the set of control and status registers and user_csr_csr selectes a particular user control and status register from that bank. The port user_condition_met indicates the CPU about the user-defined branch instruction, user_opcode_unsupported indicates that the user-defined instruction does not exist and the user_csr_unsupported indicates the CPU that the specified user control and status register does not exist. user_csr_write contains data that has to be written into the user CSR and user_csr_read contains data that is read from the user CSR. The port user_result will carry the result of the operation performed by the user-defined instruction to the CPU and user_stall informs the CPU to halt the execution of the user-defined instruction. These input and output ports defined in Verilog are shown in Figure 6.2.

```
//////////////////////////////////////////////////////
// Inputs
//////////////////////////////////////////////////////

input clk;                                  // Clock
input reset_n;                              // Reset, active-low
input user_op;                              // Indicates a valid user-defined instruction
input [`CPU_USER_OPCODE_RNG] user_opcode;    // Opcode of the user-defined instruction to execute
input [`CPU_WORD_RNG] user_operand_a;        // Operand A
input [`CPU_WORD_RNG] user_operand_b;        // Operand B
input user_csr_write;                       // Indicates a write to a user-defined CSR
input user_csr_read;                        // Indicates a read from a user-defined CSR
input [`CPU_CSR_RNG] user_csr_bank;          // Indicates which user-defined CSR bank to access
input [`CPU_CSR_RNG] user_csr_csr;           // Indicates which user-defined CSR to access
input [`CPU_WORD_RNG] user_csr_write_data;   // Data to write to user-defined CSR

//////////////////////////////////////////////////////
// Outputs
//////////////////////////////////////////////////////

output user_condition_met;                  // Indicates user-defined condition is met
wire   user_condition_met;
output user_stall;                          // Indicates user-defined instruction requires multiple-cycles
wire   user_stall;
`ifdef CFG_UNSUPPORTED_ENABLED
output user_opcode_unsupported;             // Indicates the specified user-defined instruction is not supported
wire   user_opcode_unsupported;
output user_csr_unsupported;                // Indicates the specified user-defined CSR is not supported
wire   user_csr_unsupported;
`endif
output [`CPU_WORD_RNG] user_csr_read_data;   // Result of read from user-defined CSR
wire   [`CPU_WORD_RNG] user_csr_read_data;
output [`CPU_WORD_RNG] user_result;          // Result of user-defined instruction
wire   [`CPU_WORD_RNG] user_result;
```

Figure 6.2: Input and output ports to the user-hardware module.

To implement the user-defined instruction operation, the internal signals had to be defined and declared as wires or registers. The wire user0_sel indicated that the CPU is selecting the user0 instruction. The wires mul_result, round_off and mul_result_round could hold 64-bit data and also store the results during the execution of the user-defined instruction. As seen from the Figure 6.3, the wire mul_result held the result of the 32bit signed multiplication performed on the operands a and b. The value 0.5 (0x80000000) which has to be added to the multiplication result to perform rounding-off was stored in the wire round_off. The result obtained after performing the rounding-off operation was stored in the wire mul_result_round. The final result obtained after discarding the lower 32 bits was passed on to user0_result which will be loaded on to the user_result port and passed to the CPU. The wire user0_stall indicated the CPU to stop execution of the user0 instruction.

The figure below shows the description of the internal wires and registers.

```
//////////////////////////////////////////////////////
// Internal nets and registers
//////////////////////////////////////////////////////

wire user0_sel;                         // Indicates the CPU is selecting the user0 instruction
wire [`CPU_WORD_RNG] user0_result;      // Result of user0 instruction
wire [63:0] mul_result;                 // Result of the 32-bit signed multiplication
wire [63:0] round_off;                  // Rounding off the result to the closest integer
wire [63:0] mul_result_round;           // Result of the round-off
wire user0_stall;                       // Indicates user0 instruction needs the CPU to be stalled

wire user1_sel;                         // Indicates the CPU is selecting the user1 instruction
wire [`CPU_WORD_RNG] user1_result;      // Result of user1 instruction
wire user1_stall;                       // Indicates user1 instruction needs the CPU to be stalled

wire user2_sel;                         // Indicates the CPU is selecting the user2 instruction
wire [`CPU_WORD_RNG] user2_result;      // Result of user2 instruction
wire user2_stall;                       // Indicates user2 instruction needs the CPU to be stalled

wire csr_0_sel;                         // Indicates the CPU is selecting user-defined CSR 0
reg [`CPU_WORD_RNG] user_csr_0;         // Holds the value of user-defined CSR 0
```

Figure 6.3: Showing the internal wires and registers

After this, each of the user-defined instruction operation was defined as a combinational logic. This behavioural description of the user-defined instructions user0, user1 and user2 is shown in Figure 6.4. Once the user-defined instruction operations were defined, the control signals were set accordingly. The signals indicated the CPU as to which user-defined instruction is being used at that time, read the result from the appropriate user-defined instruction and inform the CPU to stop execution once the operations are completed (Figure 6.5). Thus the user-defined were implemented in hardware and this can be attached to eSi-RISC.

```
////////////////////////////////////////////////////////
// Combinational Logic
////////////////////////////////////////////////////////

// user0:
// (long)(((long long)(operand_a)*(long long)(operand_b)+(1 << (31))) >> 32);
//  executes in a single cycle
assign mul_result = (user_operand_a * user_operand_b);
assign round_off  = 32'h0000_0000_8000_0000;
assign mul_result_round = mul_result + round_off;
assign user0_result = mul_result_round >> 32;
assign user0_stall = `FALSE;

//user1:
//(long)(((long long)(operand_a)*(long long)(operand_b)+(1 << (30))) >> 31);
//  executes in a single cycle
assign mul_result = (user_operand_a * user_operand_b);
assign round_off  = 32'h0000_0000_4000_0000;
assign mul_result_round = mul_result + round_off;
assign user1_result = mul_result_round >> 31;
assign user1_stall = `FALSE;

//user2:
//(long)(((long long)(operand_a)*(long long)(operand_b)+(1 << (27))) >> 28);
//  executes in a single cycle
assign mul_result = (user_operand_a * user_operand_b);
assign round_off  = 32'h0000_0000_0800_0000;
assign mul_result_round = mul_result + round_off;
assign user2_result = mul_result_round >> 28;
assign user2_stall = `FALSE;
```

Figure 6.4: Showing the operations defined in user0, user1 and user2.

```
// Determine which user-defined instruction is being used
assign user0_sel = user_opcode == `CPU_USER_OPCODE_WIDTH'd0;
assign user1_sel = user_opcode == `CPU_USER_OPCODE_WIDTH'd1;
assign user2_sel = user_opcode == `CPU_USER_OPCODE_WIDTH'd2;

// Select result from chosen instruction.
assign user_result =   ({`CPU_WORD_WIDTH{user0_sel}} & user0_result)
                     | ({`CPU_WORD_WIDTH{user1_sel}} & user1_result)
                     | ({`CPU_WORD_WIDTH{user2_sel}} & user2_result);

// Determine whether to stall the CPU
assign user_stall = user_op & (  (user0_sel & user0_stall)
                               | (user1_sel & user1_stall)
                               | (user2_sel & user2_stall)
                              );

`ifdef CFG_UNSUPPORTED_ENABLED
// Indicates whether the opcode is unsupported
assign user_opcode_unsupported = !user0_sel && !user1_sel && !user2_sel;
`endif

// We don't have any user-defined branch instruction, so user_condition_met should always be FALSE
assign user_condition_met = `FALSE;
```

Figure 6.5: Assignments for the control signals.

Chapter 7

Discussion

This chapter summarises the results obtained on implementation of AAC Decoder algorithm on eSi-RISC 3250 embedded processor and evaluates them comparing with published work. The chapter begins with the analysis of the decoder output (wave audio file) obtained at three different stages during the implementation and later the effect of the user-defined instructions if they were to take more than one cycle for their execution. This observation leads to the discussion of the hardware required if the user-defined executed in two or more cycles. The hardware realisation is made in terms of the gate count and is concluded by discussing power consumption and effective cost.

The output from the AAC Decoder was a wave audio file i.e. the file a5.aac input to the decoder resulted with the file a5.wav output. This file a5.wav was analysed at three different stages during the implementation on AAC on eSi-RISC. The wave files obtained from the raw decoder running on x86, after AAC Decoder was ported to eSi-RISC and after optimisation were plotted in MATLAB. The plots are shown in Figure 7.2. The function used to read the wave file is shown below.

```
%Function in MATLAB used to read a wave audio file

function analyze(file)

[s, F] = wavread(file);      % s is sound data, F is sample frequency.
t = (1:length(s))/F;           % time

ind = find(t>0.1 & t<0.12);    % set time duration for waveform plot
figure; subplot(1,2,1)
plot(t(ind),s(ind))
axis ([100000,200000,-1,1])
```

Figure 7.1: MATLAB function to read a wave audio file.

The MATLAB commands used were:

y = wavread('a5.wav')

plot(y)

plot(y(:,1))

axis ([100000,200000,-1,1]

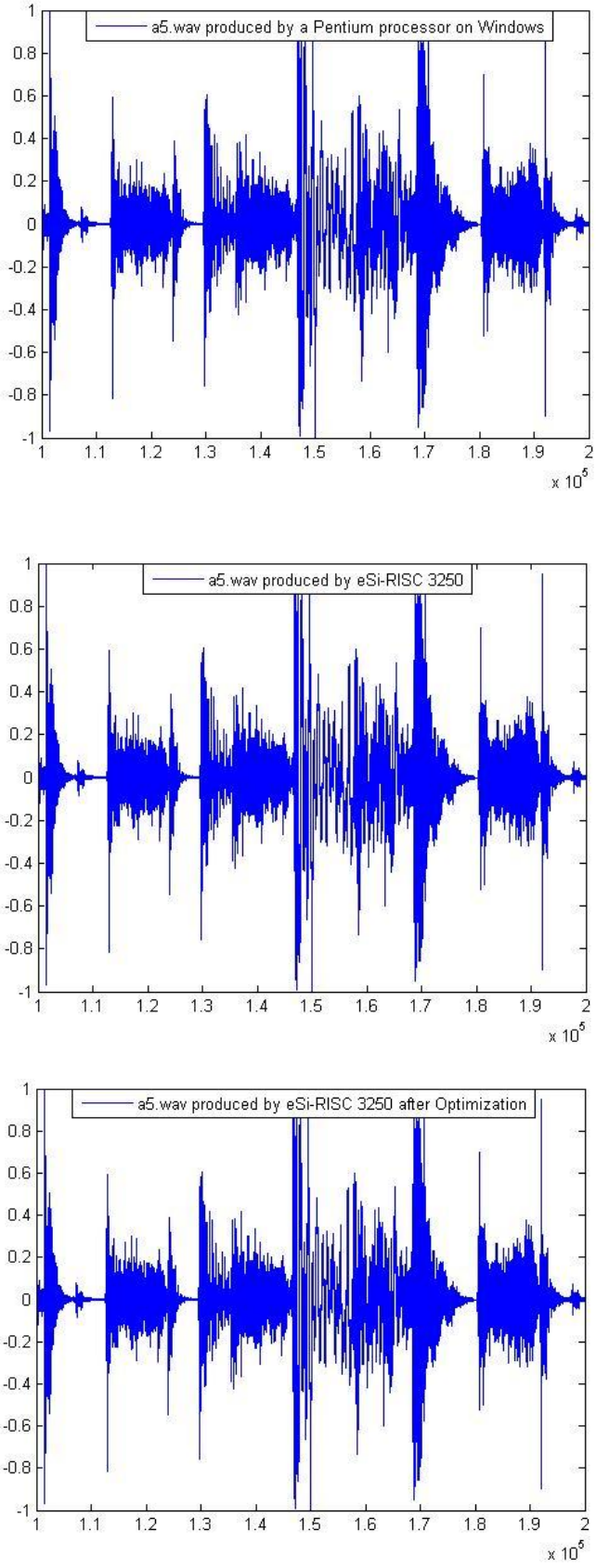


Figure 7.2: Plots of the output file a5.wav at different stages of implementation.

It is seen that all the wave files are exactly similar to each other. This proves that porting the AAC Decoder algorithm to the eSi-RISC processor platform and optimizing the algorithm using user-defined instructions did not affect the efficiency of the algorithm. This also proves that eSi-RISC processor decodes AAC input as effectively as the x86 processor.

The results obtained from optimisation shown in Chapter 6 were for single cycle execution of the user-defined instruction. The optimisation steps for esi_user0, esi_user1 and esi_user2 were repeated all over again by instructing the simulator to take 2, 4, 8 and 16 cycles for their execution. The graph below shows the behaviour of the algorithm in all the five cases.

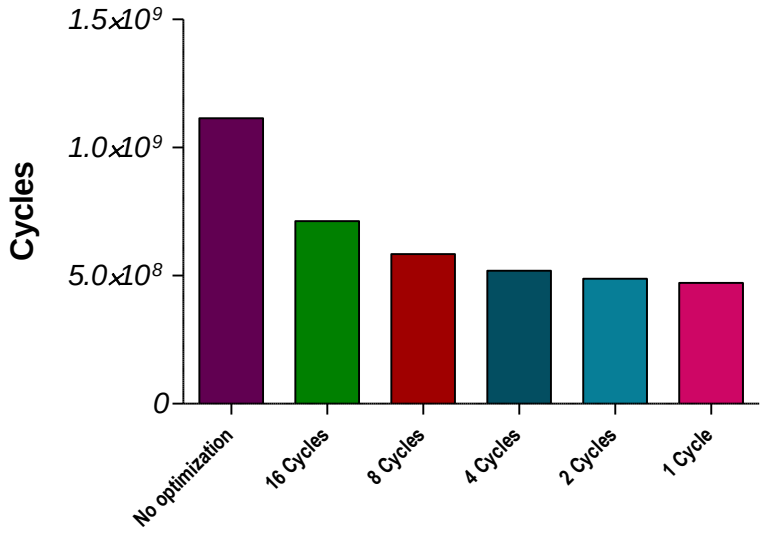


Figure 7.3: Different Hardware implementation of user-defined instructions.

With an increase in number of cycles taken to execute a user-defined instruction, the overall cycle count increases. There is a massive reduction in clock cycles with optimisation as compared to that without. Lower number in cycle execution of user-defined instruction implies more logic in hardware. This will eventually lead to an increase in number of gates during hardware implementation. The single cycle execution is the best acceleration that could be achieved for this algorithm and it will turn out to be the most expensive (in terms of the hardware required). The 16 cycle execution of the user-defined instruction will require less hardware but would lack optimal performance. Hence there is a trade off between cost and performance and the 4 cycle execution gives a balance between the two.

Further, the gate count can be estimated for the hardware implementation of the user-defined instructions. The implementation of eSi-RISC user defined instruction on the simulator had shown that it required a 32-bit multiplier and the Barrel shifter. The 32-bit multiplier will require 32*32 full adders. Each full adder has 6 logic cells. This takes the count to 32*32*6 which is 6k. The eSi-RISC 3250 core has 60k gates (Figure 7.4).

No of Cycles for the User-defined Instruction	No of Gates (Approx.)
16	375+60k
8	750+60k
4	1.5k+60k
2	3k+60k
1	6k+60k

Figure 7.4: Table showing gate count for logic design of User-defined instructions hardware.

The 32-bit full adder has two XOR gates and a MUX. Both XOR gate and MUX have two NOR gates and an AND gate. This makes it 6 logic cells.

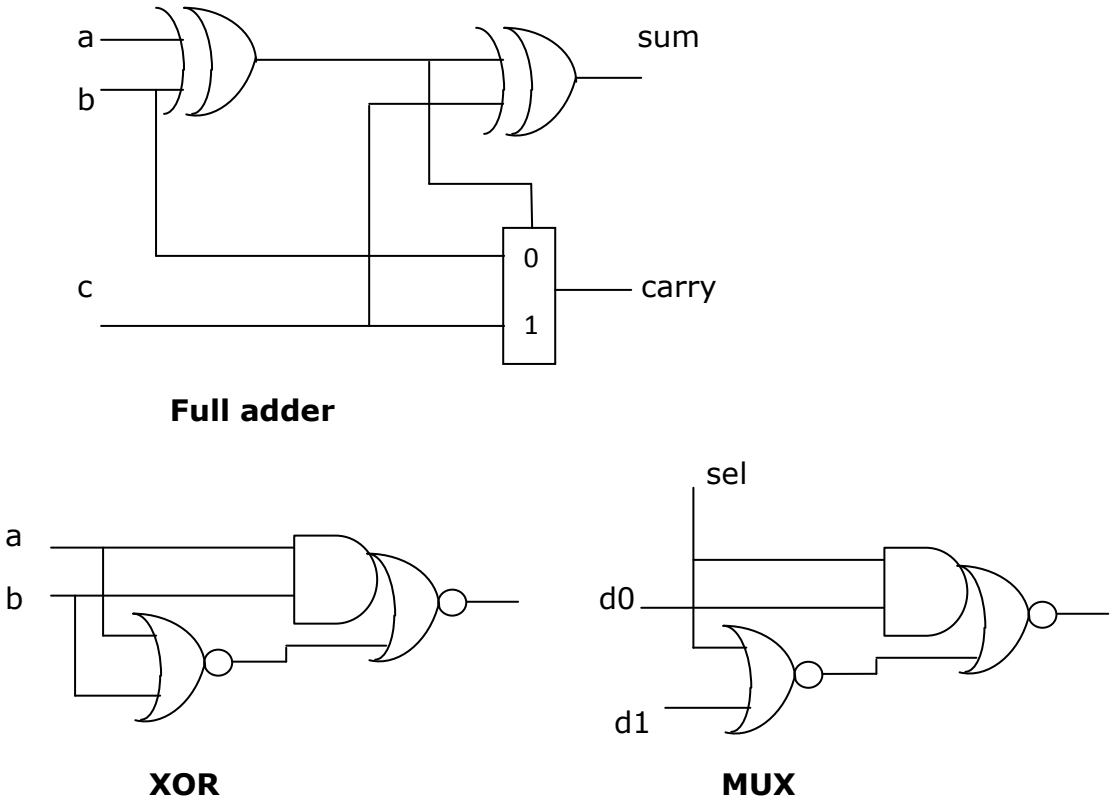


Figure 7.5: Showing the logic cells in a full adder.

The program code size is obtained from the mapfile generated during program compilation. The mapfile can be generated by passing commands to the linker. The linker command which was used to generate the mapfile was:

-Map=mapfile

This was set in project properties, eSi-RISC Cygwin C Linker and other –Xlinker option. The mapfile contains information about where all the object files created by the compiler lie in the memory. It also gives the address of the program code in the memory and its size. The table below shows the decrease in the cycles and program code size every time a user-defined instruction was used as it will replace the function in the program code by a real hardware.

Stage	Cycles	Program codesize (Bytes)
No optimisation	1114355938	276812
esi_user0	616969877	265932
esi_user0+1	506387636	242668
esi_user0+1+2	489426206	239102

Figure 7.6: Table listing the cycle count and program code size at each stage of optimization.

eSi-RISC 1600, eSi-RISC 3200 and eSi-RISC 3250 form EnSilica's family of embedded processors. The initial plan was to implement AAC decoder on eSi-RISC 3200. The object code generated by the compiler at run time overflowed the program memory (eSi-RISC 3200 had a program memory of just 64 kB) because the code size was found to be 276 kB. Hence implementation could not be carried out on eSi-RISC 3200. eSi-RISC 3250 was the next in the family and it has 32 MB DDR2 SDRAM, 1MB SRAM and 16 MB FLASH memory. eSi-RISC 3250 has shown good performance on AAC implementation. Published work has mentioned various criteria for choosing the right processor for AAC implementation (high clock frequency, 32-bit data path for ALU, multiplier and Barrel shifter). Efficient decoding of the AAC algorithm by eSi-RISC 3250 can be attributed to its architecture which has all the above mentioned features.

eSi-RISC profiler gives information about the number of instructions executed and clock cycles taken by all the functions in the program code. However, it does not provide details of memory read and writes from which the CPU usage distribution can be estimated. Extensive profiling was performed using powerful profiling tools like Cachegrind, KCachegrind and Callgrind in Linux environment. This gave details of instruction and data references and number of times every function was called. These results were in accord with the published work (Figure 4.3) and also helped to identify intensive operations in the code which was then compared with the results of the eSi-RISC profiler. These results were useful for the process of optimisation.

There are two approaches to achieve optimisation (Hardware and software) as discussed in Chapter 2. The hardware implementation gives a better processing speed and software implementation gives a flexible design solution. Hence the hardware implementation gives more acceleration but adds complexity to the design. Profiling showed that 32-bit fixed point multiplication was the most computationally intensive operation in the filterbank operation. A short algorithm was developed in C to perform this 32-bit fixed point multiplication and it did not improve the number of cycles for execution significantly. A better performance could have been achieved by writing inline assembly for 32-bit fixed point multiplication. Inline assembly will have a fewer number of instructions than the compiler generated assembly code and will be far most efficient. Using assembly language to implement a function in a fewer number of instructions was not an easily achievable task in the allotted time for this project. Moreover, the project aimed at optimising the AAC algorithm using user-defined instructions (hardware optimisation) and hence not much time was spent to achieve software optimisation like inline assembly, loop unroll, etc.

User-defined instructions were used to implement the 32-bit fixed point multiplication function. The functions faad_imdct(), ifilter_bank() and quant_to_spec() used the fixed point multiplication extensively and the macros defined were _MulHigh, MUL_F and MUL_C. Replacement of the macro functions by the custom instructions esi_user0, esi_user1 and esi_user2 proved to be very useful as it reduced the clock cycles by 56% and helped to achieve a better optimisation than what was initially proposed in the project objectives.

As discussed earlier the eSi-RISC 3200 processor was initially the target processor for the implementation of AAC decoder algorithm. One of the initial objectives of the project was to run the AAC decoder on the FPGA hardware on an Altera Cyclone III development board. eSi-RISC 3200 did not satisfy the program memory requirements and eSi-RISC 3250 embedded processor was used as the target processor. EnSilica did not have the AMBA AXI to Avalon bridge ready for eSi-RISC 3250 to access the 32 MB DDR2 SDRAM through the Avalon DDR controller. Hence the decoder could not be run on the FPGA hardware on an Altera Cyclone III board (Please see the letter attached).

It was mentioned in the initial project plan that AAC encoder algorithm will be implemented on eSi-RISC if time permits and this was a secondary objective. The encoder algorithm could not be implemented because of time constraints and the complexity of the AAC decoder algorithm.

To conclude:

- A fixed point version of the AAC decoder algorithm was successfully ported for the first time to eSi-RISC 3250.
- Compute-intensive operations were identified by profiling the program code on run-time by inputting a5.aac to the decoder.
- Hardware acceleration was achieved for the AAC decoder algorithm by using user-defined instructions and this reduced overall cycle count by 56%.
- The user-defined instruction hardware functionality was coded in Verilog which can be attached to the eSi-RISC 3250 core.

FUTURE WORK

The optimisation of the AAC decoder was achieved using user-defined instructions. However, other software optimisation techniques can be used to optimise the algorithm further.

Secondly, the user-defined instruction functionality coded in Verilog can be attached to the main processor and the decoder can be run on a real FPGA hardware.

ARM9TDMI is a leading embedded processor in the market. It is worth investigating the behaviour of the AAC decoder algorithm on the ARM9TDMI embedded processor and compare the results with eSi-RISC 3250.

Lastly, the AAC encoder algorithm could be implemented on eSi-RISC 3250 and it will be interesting to see how the encoder algorithm performs on eSi-RISC 3250 and how optimization can be achieved.

Appendix A

Results generated by Callgrind showing the number of calls made to each function in the program code

Called	Function	Location
1 512	decode_scale_factors	libfaad.so.2.0.0: syntax.c
33 686	huffman_binary_quad	libfaad.so.2.0.0: huffman.c
33 343	huffman_scale_factor	libfaad.so.2.0.0: huffman.c
1 515	0x0000000000401690	faad
1 510	memmove	libc-2.11.1.so: memmove.c
1 512	section_data	libfaad.so.2.0.0: syntax.c
66 056	faad_getbits	libfaad.so.2.0.0: bits.h
1 344	_wordcopy_fwd_dest_aligned	libc-2.11.1.so: wordcopy.c
13 828	huffman_binary_pair_sign	libfaad.so.2.0.0: huffman.c
1 512	pns_decode	libfaad.so.2.0.0: pns.c
69 796	faad_flushbits_ex	libfaad.so.2.0.0: bits.c
109 536	is_noise	libfaad.so.2.0.0: pns.h
1 512	ics_info	libfaad.so.2.0.0: syntax.c
756	is_decode	libfaad.so.2.0.0: is.c
1 512	window_grouping_info	libfaad.so.2.0.0: specrec.c
71 308	getdword	libfaad.so.2.0.0: bits.h
54 768	is_intensity	libfaad.so.2.0.0: is.h
3 031	memset	libc-2.11.1.so: memset.S
757	adts_frame	libfaad.so.2.0.0: syntax.c
757	fwrite	libc-2.11.1.so: iofwrite.c, libi
882	_IO_file_xsputn@@GLIBC_2....	libc-2.11.1.so: fileops.c
1 765	mempcpy	libc-2.11.1.so: memcpy.S
757	adts_fixed_header	libfaad.so.2.0.0: syntax.c
1 491	fread	libc-2.11.1.so: iofread.c, libio
1 491	_IO_sgetn	libc-2.11.1.so: genops.c
1 491	_IO_file_xsgetn	libc-2.11.1.so: fileops.c
757	adts_variable_header	libfaad.so.2.0.0: syntax.c
14 236	huffman_getescape	libfaad.so.2.0.0: huffman.c
1 491	memcpy	libc-2.11.1.so: memcpy.S
166	_wordcopy_fwd_aligned	libc-2.11.1.so: wordcopy.c
226	vfprintf	libc-2.11.1.so: vfprintf.c, prir
777	malloc	libc-2.11.1.so: malloc.c
1 722	_IO_default_xsputn	libc-2.11.1.so: genops.c
1	_dl_start	ld-2.11.1.so: rtld.c, dl-machi

Appendix B

Results generated by the eSi-RISC profiler

Function	Instructions	Cycles
__udivsi3	174634	274442
__do_global_ctors_aux	12	17
_fseek_r	100	137
__lo0bits	32	38
aac_frame_decode	159461	201044
cfftu	26	40
filter_bank_init	42	50
memcpy	1808580	2322060
tns_decode_frame	66440	72480
__muldf3	2594	2910
__sread	4155	5540
sprintf	3131	3535
NeAACDecDecode	5	7
malloc	36	56
__sfp_lock_acquire	751	2253
__sseek	16	20
_sys_timer_interrupt_handler	522456	674839
__udivmodsi4	3276	5012
_open_r	38	46
__divdf3	236	282
_getargs	24	35
drc_end	7	11
__sclose	7	11
_malloc_r	1539	2011
strlen	78	116
__sfvwrite_r	918	1054
__register_exitproc	31	41
__udivmodsi4	1143227	1759101
faad_free	84	140
fill_buffer	48207	69251
passf4pos	128611714	141178235
_vfprintf_r	3672	5080
_sbrk	60	72
__floatunsidf	54	62
faad_initbits	40771	44550
passf2pos	11762141	13304321
__smakebuf_r	58	70
faad_fprintf	38	48
__malloc_unlock	8	24
fread	15	19
NeAACDecInit	123	153
__malloc_unlock	14	42
__do_global_dtors_aux	30	35
faad_get_processed_bits	6048	7560
main	199	252

__udivmodsi4	362	560
__divsi3	15121	18149
fopen	22	30
_Bfree	9	12
decodeAACfile	36237	50776
_close	65	75
_fstat_r	54	66
_getopt_internal	118	155
_write_r	209	253
__extendsfdf2	52	56
huffman_scale_factor	7078	9036
__udivsi3	147	231
malloc	99	154
is_decode	912169	1158579
_lseek	65	75
__sfp_lock_release	749	2247
open_audio_file	65	88
_Balloc	96	112
_lseek_r	19	23
__sfp_lock_acquire	749	2247
__sread	75	100
__divsf3	764	996
NeAACDecGetCapabilities	2	4
frame_dummy	9	14
_fopen_r	49	61
_fread_r	54159	70184
_cleanup_r	7	11
__env_lock	1	3
_lseek	13	15
_fread_r	54396	70499
print_channel_info	122	161
ms_decode	27	31
__muldf3	1969	2209
reconstruct_channel_pair	140484	169189
__muldi3	29302	33998
__env_unlock	1	3
faad_endbits	756	2268
__sfmoreglue	30	36
decodeAACfile	27332	34930
_vfprintf_r	31992	39808
_getargv	13	15
strchr	96	135
__sfp_lock_release	751	2253
pns_decode	1362	1804
memset	17925408	22390240
faad_malloc	12	20
__muldi3	403967998	466055158
memmove	17876286	22998816
.boot	1	3
__nesf2	19	21
_close_r	85	105
_write	143	165
faad_imdct	103490788	119362446
__malloc_lock	8	24
__fixunssfsi	68	84

_dtoa_r	754	992
__mulsf3	114	136
huffman_2step_quad	10215821	11977941
_read	3510	4050
__mulsf3	184	220
__fixdfsi	161	175
__fixunsdfsi	46	58
ics_info	132	166
_lseek_r	19	23
__divsf3	768	992
get_sample_rate	22	26
strlen	14120	20212
.boot	45	64
strrchr	25	35
__srefill_r	13313	17476
__ssprint_r	28785	34239
__d2b	60	70
faad_flushbits_ex	2441360	2790320
output_to_PCM	37996326	45756797
atexit	10	14
huffman_2step_quad	28687	33655
__lo0bits	59	75
faad_imdct	1	1
_write	182	210
_open	13	15
_sys_timer_interrupt_handler	12288	15872
_malloc_r	895	1173
_localeconv_r	6	10
_fclose_r	233	340
faad_fprintf	2147	2712
_read_r	5130	6210
pns_decode	1734516	2297377
fclose	45	70
_close	13	15
__floatsidf	252	280
vfprintf	130	170
reconstruct_channel_pair	215	286
strcpy	135	173
strlen	253	387
close_audio_file	28	48
__malloc_lock	14	42
__floatsidf	368	416
raw_data_block	90649	122390
__sread	4050	5400
decodeAACfile	152	216
_Balloc	75	99
_localeconv_r	30	50
_sbrk	60	72
is_decode	711	894
__d2b	120	140
ifilter_bank	65	71
_lseek_r	95	115
tns_decode_frame	88	96
memset	27960	34926
memcpy	3872	4480

__addsf3	147	175
__fpclassifyd	30	46
channel_pair_element	96	114
cfftb	9226154	11042010
__subdf3	1085	1218
__malloc_lock	1531	4593
faad_byte_align	17557	23121
__floatsisf	14	16
NeAACDecDecode	6043	9065
__gtdf2	27	39
_open	26	30
_lseek	13	15
__swrite	319	385
fseek	13	17
_write_r	61981	75041
_ftell_r	31	38
__eqdf2	54	60
__fpclassifyd	60	92
_vfprintf_r	1303	1704
ics_info	97845	127175
filter_bank_end	15	23
window_grouping_info	1268	1478
getdword_n	28	32
vfprintf	26	34
__srefill_r	240	315
__sprint_r	192	246
__fixdfsi	168	192
fread	11190	14174
__malloc_unlock	1531	4593
huffman_scale_factor	3189342	4067270
_svfprintf_r	58677	76245
_read	3601	4155
adts_frame	266	332
_fflush_r	80	98
__sfp	79	108
_sys_timer_interrupt_handler	144	186
__sfvwrite_r	567	667
__sfvwrite_r	181238	241042
NeAACDecGetCurrentConfiguration	6	8
getenv	10	14
adts_frame	100416	125335
__sinit	34	42
__extendsfdf2	78	84
_times_r	7	11
__gtdf2	56	78
std	93	114
_fopen_r	98	122
__call_exitprocs	85	113
fopen	11	15
get_sample_rate	8316	9828
faad_getbits	1786175	2319012
__clzsi2	333	449
__umodsi3	672	1056
clock	19	26
cffti	422	512
memcmp	38517	55893

.fini	6	10
_free_r	30774	40186
can_decode_ot	12	14
write_wav_header	300	360
__floatsisf	130	144
faad_endbits	1	3
_fwrite_r	29530	37102
_sbrk_r	102	126
faad_mdct_init	57	73
raw_data_block	95	125
huffman_spectral_data	26765055	35259470
_times	17	21
ifilter_bank	55437763	64908448
__lesf2	128	186
__sfp_lock_acquire	2	6
_read_r	95	115
_fread_r	147	201
NeAACDecClose	1119	1766
__sfp_lock_release	2	6
__srefill_r	12967	17025
_calloc_r	96	116
quant_to_spec	77197	91787
can_decode_ot	12	14
getopt_long	7	11
malloc	6813	10598
_fflush_r	36462	50877
__sprint_r	384	492
exit	13	22
_sbrk_r	102	126
fread	11160	14136
faad_fprintf	190	240
_read	65	75
_fstat	39	45
fclose	9	14
__ltdf2	129	177
__sflags	19	28
__udivsi3	21	33
__swsetup_r	43	56
drc_init	27	33
huffman_spectral_data	12673	16901
__eqdf2	108	120
_fseek_r	233	325
aac_frame_decode	81	95
_open_r	19	23
ms_decode	20386	23410
__clzsi2	418	554
_write	42406	48930
__sfp	151	209
fill_element	180	257
_findenv_r	113	167
__divdf3	237	285
quant_to_spec	56342357	67828357
_fclose_r	49	74
__floatunsisf	123	143
__sflags	34	52
fseek	13	17

__smakebuf_r	141	169
__subdf3	801	908
huffman_2step_pair	12031010	13909702
faad_get1bit	66394	77258
_dtoa_r	405	536
__sprint_r	7104	9102
faad_malloc	66	110
vfprintf	1469	1921
free	6939	10794
_times_r	7	11
fseek	26	34
__ltdf2	84	116
get_sr_index	22	26
ftell	9	14
memcpy	1812500	2326596
_write_r	266	322
_fseek_r	75	100
NeAACDecSetConfiguration	42	51
_sbrk_r	51	63
_getargvlen	13	15
_fstat_r	18	22
esi_timer_auto_init_sys_timer	29	46
esi_device_get	94	158
window_grouping_info	914845	1073080
__swsetup_r	53	70
_fstat	13	15
memmove	18600844	23935485
_exit	4	4
memcmp	11	18
_malloc_r	69128	92677
memset	4248	5304
_times	17	21
esi_timer_init_sys_timer	54	62
write_audio_file	13964054	17065400
fill_buffer	32	40
faad_malloc	6	10
faad_get1bit	104	120
faad_getbits	1666	2162
fwrite	11355	14383
_fwalk	95	133
__swrite	406	490
__umodsi3	70	110
T.47	10292952	13353319
faad_flushbits_ex	1540	1760
strcat	42	57
_free_r	39	51
_read_r	5263	6371
channel_pair_element	80870	97860
clock	19	26
__adddf3	1023	1145
faad_mdct_end	26	38
main	48	67
fill_buffer	48222	69266
NeAACDecOpen	923	1151
_localeconv_r	642	1070

Appendix C

Optimisation details of the function quant_to_spec() using esi_user2

```
      for (win = 0; win < ics->window_group_length[g]; win++)
         {
             for (bin = 0; bin < width; bin += 4)
             {
                 real_t iq0 = iquant(quant_data[k+0], tab, &error);
                 real_t iq1 = iquant(quant_data[k+1], tab, &error);
                 real_t iq2 = iquant(quant_data[k+2], tab, &error);
                 real_t iq3 = iquant(quant_data[k+3], tab, &error);

                 wb = wa + bin;

                 if (exp < 0)
                 {
                     spec_data[wb+0] = iq0 >>= -exp;
                     spec_data[wb+1] = iq1 >>= -exp;
                     spec_data[wb+2] = iq2 >>= -exp;
                     spec_data[wb+3] = iq3 >>= -exp;
                 } else {
                     spec_data[wb+0] = iq0 <<= exp;
                     spec_data[wb+1] = iq1 <<= exp;
                     spec_data[wb+2] = iq2 <<= exp;
                     spec_data[wb+3] = iq3 <<= exp;
                 }
                 if (frac != 0)
                 {
      spec_data[wb+0] = MUL_C(spec_data[wb+0],pow2_table[frac]);
      spec_data[wb+1] = MUL_C(spec_data[wb+1],pow2_table[frac]);
      spec_data[wb+2] = MUL_C(spec_data[wb+2],pow2_table[frac]);
      spec_data[wb+3] = MUL_C(spec_data[wb+3],pow2_table[frac]);
                 }
```

The assembly listing of the function quant_to_spec() showed calls made to the libgcc function __muldi3

```
4019dd4:   f000 2076   bz    r16, 4019ec0 <quant_to_spec+0x30e>
4019dd8:   f800 a401   lw    r24, (r17+[0])
4019ddc:   e00c 3b1c   add   r12, r6, r12
4019de0:   a40c        lw    r8, (r12+[0])
4019de2:   f419 340f   sr    r25, r24, 31
4019de6:   e840 3d68   mv    r10, r24
4019dea:   e840 3de9   mv    r11, r25
4019dee:   e409 340f   sr    r9, r8, 31
4019df2:   7e92        sw    (sp+[18]), r13
4019df4:   7f11        sw    (sp+[17]), r14
4019df6:   7f90        sw    (sp+[16]), r15
4019df8:   f000 7813   sw    (sp+[19]), r16
4019dfc:   f000 788f   sw    (sp+[15]), r17
4019e00:   7e14        sw    (sp+[20]), r12
4019e02:   e003 d002   call  401fe06 <__muldi3>
4019e06:   e808 3c1d   add   r8, r8, r29
4019e0a:   e829 3cfe   addc  r9, r9, r30
4019e0e:   6614        lw    r12, (sp+[20])
4019e10:   34a4        sl    r9, r9, 4
4019e12:   e408 344c   sru   r8, r8, 28
4019e16:   e008 3cc8   or    r8, r9, r8
4019e1a:   bc0c        sw    (r12+[0]), r8
4019e1c:   3e64        mv    r12, r4
4019e1e:   c601        add   r12, 1
4019e20:   3622        sl    r12, r12, 2
4019e22:   e00c 3b1c   add   r12, r6, r12
4019e26:   a40c        lw    r8, (r12+[0])
4019e28:   e840 3d68   mv    r10, r24
4019e2c:   e840 3de9   mv    r11, r25
4019e30:   e409 340f   sr    r9, r8, 31
4019e34:   7e14        sw    (sp+[20]), r12
4019e36:   e002 dfe8   call  401fe06 <__muldi3>
4019e3a:   e808 3c1d   add   r8, r8, r29
4019e3e:   e829 3cfe   addc  r9, r9, r30
4019e42:   6614        lw    r12, (sp+[20])
4019e44:   34a4        sl    r9, r9, 4
4019e46:   e408 344c   sru   r8, r8, 28
4019e4a:   e008 3cc8   or    r8, r9, r8
4019e4e:   bc0c        sw    (r12+[0]), r8
4019e50:   3e64        mv    r12, r4
4019e52:   c602        add   r12, 2
4019e54:   3622        sl    r12, r12, 2
```

Function quant_to_spec() accelerated using user-defined instruction esi_user2 and its assembly listing

```
            for (win = 0; win < ics->window_group_length[g]; win++)
            {
                for (bin = 0; bin < width; bin += 4)
                {
                    real_t iq0 = iquant(quant_data[k+0], tab, &error);
                    real_t iq1 = iquant(quant_data[k+1], tab, &error);
                    real_t iq2 = iquant(quant_data[k+2], tab, &error);
                    real_t iq3 = iquant(quant_data[k+3], tab, &error);

                    wb = wa + bin;

                    if (exp < 0)
                    {
                        spec_data[wb+0] = iq0 >>= -exp;
                        spec_data[wb+1] = iq1 >>= -exp;
                        spec_data[wb+2] = iq2 >>= -exp;
                        spec_data[wb+3] = iq3 >>= -exp;
                    } else {
                        spec_data[wb+0] = iq0 <<= exp;
                        spec_data[wb+1] = iq1 <<= exp;
                        spec_data[wb+2] = iq2 <<= exp;
                        spec_data[wb+3] = iq3 <<= exp;
                    }
                    if (frac != 0)
                    {
            spec_data[wb+0] = esi_user2(spec_data[wb+0],pow2_table[frac]);
            spec_data[wb+1] = esi_user2(spec_data[wb+1],pow2_table[frac]);
            spec_data[wb+2] = esi_user2(spec_data[wb+2],pow2_table[frac]);
            spec_data[wb+3] = esi_user2(spec_data[wb+3],pow2_table[frac]);
                    }


4022d9a:    f7ff 2320    bz     r22, 4022cda <quant_to_spec+0x200>
4022d9e:    e004 3d94    add    r4, r11, r4
4022da2:    e800 a308    lw     r6, (r24+[0])
4022da6:    a384         lw     r7, (r4+[0])
4022da8:    e0e7 3ba6    user2  r7, r7, r6
4022dac:    bb84         sw     (r4+[0]), r7
4022dae:    3a6d         mv     r4, r13
4022db0:    c201         add    r4, 1
4022db2:    3222         sl     r4, r4, 2
4022db4:    e004 3d94    add    r4, r11, r4
4022db8:    a384         lw     r7, (r4+[0])
4022dba:    e0e7 3ba6    user2  r7, r7, r6
4022dbe:    bb84         sw     (r4+[0]), r7
4022dc0:    3a6d         mv     r4, r13
4022dc2:    c202         add    r4, 2
4022dc4:    3222         sl     r4, r4, 2
4022dc6:    e004 3d94    add    r4, r11, r4
4022dca:    a384         lw     r7, (r4+[0])
4022dcc:    e0e7 3ba6    user2  r7, r7, r6
4022dd0:    bb84         sw     (r4+[0]), r7
4022dd2:    c683         add    r13, 3
4022dd4:    36a2         sl     r13, r13, 2
```

References

[1] Madisetti, Vijay. K and Williams, Douglas B , eds.”The Digital Signal Processing Handbook,” Section IX. CRC Press LLC, 1998.

[2] E.Kurniawati, C.T.Lau, B. Premkumar, J.Absar, S. George,” New Implementation Techniques of an Efficient MPEG Advanced Audio Coder,”IEEE, 2004

[3] ISO/IEC 14496-3, “Information Technology – Coding of audio-visual objects, Part 3: Audio,” 1999

[4] D.Y.Huang, X.Gong, D.Zhou, “Implementation of the MPEG-4 Advanced Audio Coding Encoder on ADSP-21060 SHARC,”IEEE 1999.

[5] SPANIAS, Andreas. “Speech Coding. A Tutorial Review,”Proceedings of IEEE, Newyork,Oct. 1994.

[6] A.Servetti, A.Ronotti, J.C.D.Martin, “Fast Implementation of the MPEG-4 AAC Main and Low complexity Decoder,”IEEE, 2004.

[7] Y.Ming, W. Jia,” Design and Implementation of MPEG-4 AAC Decoder on ARM Embedded System for CMMB receiver,”IEEE, 2009.

[8] J.P. Princen, A.W. Johnson, and A.B. Bradley, “Subband/transform coding using filter bank designs based on Time Domain Aliasing Cancellation,” in Proc. IEEE Int.Conf. Acoust., Speech, Signal Processing, 1987.

[9] K.H. Bang, J.S. Kim, N.H. Jeong , Y.C. Park “ Design Optimization of MPEG-2 AAC Decoder,”IEEE , 2001.

[10] R.K. Chivukula, Y.A. Reznik, V. Devarajan, “Efficient algorithms for MPEG-4 AAC-ELD, AAC-LD and AAC-LC filter banks, “ IEEE, 2008.

[11] HILPERT, Johannes et al. Real-Time Implementation of the MPEG-4 Low Delay Advanced Audio Coding Algorithm (AAC-LD) on Motorola DSP56300. In: AES CONVENTION, 108., 2000. Proceedings... Newyork: Audio Engineering Society, 2000.

[12] To operate from within 70MIPS MPEG-2 AAC Encoder Chip Design (An MPEG-2 AAC Encoder chip design operating under 70MIPS). Institute of Electronics Engineers of SD / v.42, no.4 , 2005, pp.61-68

[13] High-quality and Processor-Efficient Implementation of an MPEG-2 AAC Encoder,’ 0-7803-7041-4/01 2001 IEEE Takamizawa, Yuichiro; Nomura, Toshiyuki; Ikekawa , Masao.

[14] K.S Lee, Y.C Park and D.H. Youn, “SOFTWARE OPTIMIZATION OF THE MPEG-AUDIO DECODER USING A 32-BIT MCU RISC PROCESSOR,” IEEE, August 2002.

[15] K.H. Bang, J.S. Kim, N.H. Jeong, Y.C. Park, D.H. Youn, "Design Optimization of MPEG-2 AAC Decoder," IEEE, 2001.

[16] K.S Lee, Y.C Park and D.H. Youn, "Software Optimization of the MPEG-Audio Decoder using a 32-bit MCU RISC Processor," IEEE, 2002.

[17] K.S. Lee H.O. O, Y.C. Park and D.H. Youn, "High Quality MPEG-Audio Layer III Algorithm for a 16-BIT DSP," IEEE, 2001.

[18] C.N. Liu, T.H. Tsai, "SoC Platform Based Design of MPEG-2/4 AAC Audio Decoder," IEEE, 2005.

[19] Y. Ming, H. Guorong, W. Jia, "Design and Implementation of MPEG-4 AAC Decoder on ARM Embedded System for CMMB Receiver," IEEE, 2009.

[20] EnSilica's eSi-RISC 3250 user manual.

From: "David Wheeler" <david.wheeler@ensilica.com>
Subject: Comments to MSc Examiners
Date: Wed, September 15, 2010 9:27 am
To: "Avinash X" <ax9710@bristol.ac.uk>
Cc: "Neil Burgess" <Neil.Burgess@bristol.ac.uk>

Hi Avinash,

Please print this email and pass on to your MSc examiners

Dear Examiners

I'd just like to clarify why one particular initial objective set for Avinash could not be achieved. We had hoped Avinash could try out the AAC decoder in FPGA hardware on an Altera Cyclone III development board. Once we found out the program memory requirements it became clear that this would have needed access to the DDR memory. Unfortunately we didn't have the AMBA AXI to Avalon bridge ready for our processor to access DDR through the Avalon DDR controller. Therefore Avinash couldn't go on to complete this objective.

Kind Regards,

David

Dr David Wheeler - Technical Director

_____

EnSilica Limited
The Barn, Waterloo Road
Wokingham
Berkshire
RG40 3BY

Tel : +44 (0)1183 217 332
Mob : +44 (0)7941 373 246
Fax : +44 (0)1189 798 160
Web : http://www.ensilica.com <http://www.ensilica.com/>

Email: david.wheeler@ensilica.com <mailto:david.wheeler@ensilica.com>