ADVANCED AUDIO CODING ON AN FPGA

By

Ryan Linneman

School of Information Technology and Electrical Engineering,

University of Queensland, Brisbane.

Submitted for the Degree of

Bachelor of Engineering (Honours)

in the division of

Computer Systems Engineering

October 2002

Cromwell College

Walcott St.

St. Lucia QLD 4067

16 October 2002

Professor Simon Kaplan

Head of School of Information Technology and Electrical Engineering

University of Queensland

St. Lucia, QLD 4072

Dear Professor Kaplan,

In accordance with the requirements of the Degree of Bachelor of Engineering (Honours) in the division of Computer Systems Engineering, I present the following thesis entitled "Advanced Audio Coding on an FPGA". This work was performed under the supervision of Dr Peter Sutton.

I declare that the work in this thesis is written with honesty and integrity, and is a true report of the work I have undertaken. The work submitted is my own, except as acknowledged, and has not previously been submitted for a degree at The University of Queensland or any other institution.

Yours sincerely,

Ryan Linneman

Abstract

This thesis presents an investigation and partial implementation of the MPEG-2 AAC decoding algorithm on a Field Programmable Gate Array (FPGA). Advanced Audio Coding (AAC) is a state-of-the-art natural audio coding algorithm that is superior to MP3 and is capable of producing better than CD quality audio. The algorithm incorporates a number of decoding tools, most of which involve heavily repetitive computations.

An official AAC conformance test bitstream was selected for decoding, and five of the ten decoding tools are required to decode the bitstream. An in-depth analysis and partial decoding of the bitstream was undertaken and the results provide the theoretical foundation for the thesis.

The hardware environment for the thesis was a Xess XSV Development board housing a Virtex XCV300 FPGA. The FPGA's capacity to handle parallel processing and to perform well with repetitive tasks suited it well to the AAC decoding algorithm.

An full implementation of the quantisation and scalefactor tools using fixed-point arithmetic was written in VHDL and a partial implementation of the bitstream demultiplexer was also written. The quantisation and scalefactor tools were synthesised and implemented with Xilinx Foundation and when executed on the FPGA, showed ±1% accuracy compared to a floating-point software implementation.

Recommendations for future work and development for the MPEG-2 AAC algorithm on an FPGA are made also.

Acknowledgements

This thesis could not have been completed without the guidance and encouragement of many people. I would like to particularly acknowledge those below.

Dr Peter Sutton for his supervision and guidance throughout the project and his commitment to meeting with me each week to encourage me and to offer me feedback.

Simon Leung, for his help with the VHDL implementation and taking the time to respond to my bothersome emails.

My **Mum** and **Dad** for their continual encouragement and their wonderful parenting over 21 years to put me in a place to write this thesis.

My friends at Cromwell, especially **Mel Stuart**, for sharing the load.

Contents

List of Figures

List of Tables

1 INTRODUCTION

1.1 The Internet Audio Market

Internet Audio is one of the world's fastest growing markets (Figure 1-1). All analysts have predicted dramatic growth over the next few years, continuing the trend of the last few years [1]. Many coding algorithms are available and emerging, competing for their share in the market.

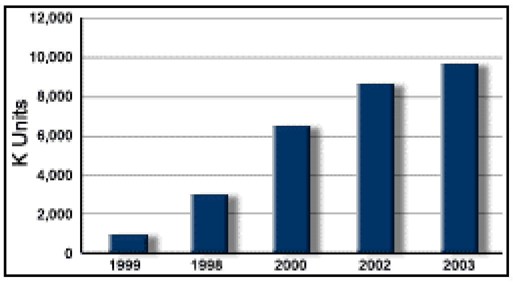


Figure 1-1: Market Growth

Currently, MP3 is the most popular format for Internet Audio distribution, though RealNetworks G2 format is the undisputed leader in the streaming audio market. The MP3 algorithm provides an 11:1 compression rate for "near CD quality" stereo audio, with a sampling rate of 44.1kHz.

The growth of the Internet Audio market has been so rapid that the legal framework for distribution is lagging behind. Copyright infringement and the distribution of Internet Audio are the central issues in an ongoing court battle involving Napster Inc. and their website that allows users to share and swap MP3s [2]. Thus, quality and compression are not the only important considerations for new encoding algorithms; copyright protection is in increasing demand.

Table 1-1 presents a brief survey of the Internet Audio market [3].

Format	**Developer**	**Type**	**Secure**
WMA	Microsoft	Both	Yes
G2	RealNetworks	Streaming	No
Music Codec	QDesign	Streaming	No
LiquidAudio	Liquid Audio	Download	Yes

TwinVQ	Yamaha	Download	No
a2b	AT&T	Download	Yes
AAC	Fraunhofer Institute	Download	No
MP3	Fraunhofer Institute	Download	No

Table 1-1: Survey of Internet Audio Market

AAC (Advanced Audio Coding) is a state-of-the-art natural audio coding algorithm that can handle 48 channels and sample at rates up to 96kHz. AAC is an advance of the successful MP3 format and its superior performance arises from the efficient use of existing coding tools and the introduction of new coding tools [4]. On average, AAC's compression rate is 30% higher than that of MP3 while providing better sound quality [5].

The AAC standard itself is not secure, but "is only being licensed in the context of secure distribution. Encoders and players each have their own Digital Rights Management mechanisms, and interoperability between devices is not possible." (Andrew Fischer – Director of Licensing Business Development at Dolby Laboratories). Thus AAC is also more secure than MP3.

1.2 The AAC Standard ISO/IEC 13818-7

Advanced Audio Coding was developed by AT&T Corporation, Dolby Laboratories, the Fraunhofer Institute for Integrated Circuits and Sony Corporation. AAC was standardised under the joint direction of the International Organisation of Standardisation and the International Electro-Technical Committee as part of the MPEG-2 specification in 1997 [6].

Initially, AAC was defined by 13818-3 [7] as a multi-channel extension to MPEG-1 (the standard defining MP3) and was backwards compatible with MPEG-1. However, AAC was limited by the MPEG-1 standard, and 13818-7 was introduced. This standard defines AAC audio that is incompatible with MPEG-1.

AAC is defined by three different profiles – Main Profile, Low Complexity (LC) Profile and Scalable Sample Rate (SSR) Profile. The Main Profile includes the full set of coding tools and produces the highest quality output. The LC profile limits some coding tools (TNS) and excludes others (Prediction and Pre-programming). The SSR profile is the least complex of the three, using a crippled filter bank [8].

The difference in the performance and the quality of the Main Profile compared to the LC profile is small. However, the encoding and especially the decoding complexity of the Main Profile compared to the LC profile is significant [9].

Table 1-2 is a comparison of MPEG-2 AAC coding and MPEG-1 MP3 coding. AAC can handle more channels than MP3 and higher sampling frequencies. In two channel listening tests, it has been demonstrated that AAC at 96 kb/s can provide slightly better audio quality than MP3 at 128 kb/s, and MP2 at 196 kb/s [10].

	AAC	**MP3**
Channels	48	5
Sampling Rate	96kHz	48kHz
Comparison Ratio	15:1	11:1
Kilobytes / minute	670kB	900kB

Table 1-2: Comparison of AAC and MP3

Besides its strong near-term market in Internet Audio, AAC currently has two other major area of application. The Japanese Association of Radio Industries and Businesses has selected MPEG-2 AAC as the only audio coding scheme for all of Japan's digital broadcast systems, including standard-definition television (SDTV), high-definition television (HDTV), digital radio, and new multimedia services. As well, the Digital Radio Mondiale organisation has selected an extended version of MPEG-2 AAC for the Digital Broadcasting on short-, medium-, and longwave (AM) in the United States [11].

1.3 Objectives and Contribution

The objective of this thesis project is to investigate, and implement, an audio coding algorithm on an FPGA. As the basis for the project, an MPEG-2 AAC decoder for real-time decoding of an AAC LC bitstream has been selected. Such an implementation could be suitable for use as the primary processing core of a portable Internet Audio player, and could also find application in a standalone network device for decoding audio files remotely.

As stated, the processing core of the hardware environment in which the implementation will take place is a Field Programmable Gate Array (FPGA). FPGAs are a relatively recent technology (first introduced in 1989) and differ significantly in architecture from the more widely used architectures that have fixed CPUs [12]. Traditional CPUs are largely sequential in operation, but FPGAs offer substantial parallel processing power.

The implementation of repetitive algorithms using FPGAs aims to take advantage of the parallel processing power of the FPGAs. In this thesis, the tools of the AAC decoding algorithm were investigated and an attempt to implement the tools on an FPGA was made. In doing so, this thesis made original contributions to the audio coding field in the following areas.

- A survey of the developments using the MPEG-2 AAC coding standard that revealed the absence of any implementations of the MPEG-2 AAC decoding algorithm using FPGA technology as either the central processing core or as a co-processor.
- A full fixed-point VHDL implementation of two of the five required decoding tools (quantisation tool and scalefactor tool) and partial VHDL implementation of the bitstream demultiplexer tool.
- An evaluation of the suitability of the use of an FPGA as the processing core, and recommendations for future developments with the AAC decoding algorithm using FPGA technology.

1.4 Overview of remaining chapters

The first step toward an FPGA implementation of an AAC decoder was an investigation of the commercial applications and academic research involving hardware and software implementations of the algorithm. Chapter 2 details the findings of this investigation and explains why the MPEG-2 AAC algorithm was chosen.

The AAC algorithm is complex and involves many decoding tools. Chapter 3 provides an overview of the algorithm and coding tools, as well as the available conformance test bitstreams, and extrapolates a block diagram for the FPGA implementation.

Before proceeding to a hardware description of the AAC decoder, a thorough investigation of the conformance test bitstream gives a deep understanding of the MPEG-2 AAC algorithm. Chapter 4 presents the findings of this investigation and the details and results of a hand-calculated partial decode of the bitstream.

Together with an understanding of the algorithm, a good understanding of the hardware environment of the AAC decoder is essential. Chapter 5 surveys the hardware used for prototyping and the software used for coding the hardware description of the decoder.

With a broad foundation of knowledge of the algorithm and the hardware, the actual hardware description of the decoder is detailed in Chapter 6. VHDL coding techniques and implementation difficulties and solutions are included.

An evaluation of the implementation described in previous chapters is the basis of Chapter 7. The final product is evaluated and the overall performance of myself as an engineer is evaluated.

In light of the product evaluation, future work and developments on the topic are suggested and elaborated upon in Chapter 8.

Chapter 9 draws together the main conclusions of the thesis.

2 REVIEW OF PREVIOUS WORK

This chapter reviews the relevant hardware and software implementations of the most recent audio coding algorithms, especially AAC, with the goal of highlighting the significance of the contribution of this thesis project. Also, some reasoning for selecting the MPEG-2 AAC algorithm is given, and the use of an FPGA as the processing core is investigated.

2.1 MP3 FPGA Solutions

The most dominant algorithm in the rapidly expanding field of Internet Audio is the MPEG-1 Layer III (MP3) standard [13]. Accordingly, the MP3 standard has been widely investigated and hardware implementations of encoders and decoders are well documented. Two of these are outlined below.

Xilinx have entered the portable Internet Audio player market with a combination solution of an FPGA and a microcontroller [1]. The Spartan FPGA provides high performance as well as quick time to market. The VHDL source code is freely available at the Xilinx website, and Xilinx are regularly developing new FPGA solutions for Internet Audio.

Celoxica, in an attempt to demonstrate the power of their Handel-C programming language, implemented a fully functional MP3 encoder on an FPGA [14]. Line by line, software source code (written in C) was translated to Handel-C in less than eight weeks.

In summary, implementations of the MP3 standard are too well common for MP3 to be the base algorithm of this thesis. Such a topic would fail to make an appropriate contribution to the subject area. This information, together with advice from Dr Peter Sutton, led to choosing the more recent and largely unimplemented AAC coding algorithm.

2.2 Licensed AAC decoders

At the time of writing, only a handful of AAC licensees had announced a decoder implementation for consumer electronic applications. These products are reviewed and compared below.

2.2.1 Texas Instruments

The TMS320C54x series of DSPs [15] are the basis of Sanyo's new SSP-PD7 Internet Audio player [16] (Figure 2-1). The player is extremely low power, running for 5 hours on a single AAA battery. The maximum sampling rate is 96kHz and the LC profile is used (both ADIF and ADTS – see section 4.1 for more information). The TMS320C67x series [17] can also be used and includes floating-point capabilities.

Figure 2-1: Sanyo SSP-PD7

2.2.2 Princeton

The PT8402 [18] (released November 2001) is capable of decoding an AAC bitstream (either ADIF or ADTS) at sampling frequencies up to 24kHz. The 100-pin chip requires a 2.5V power source and has an in-built D/A converter. The bitstream is delivered via a serial bit stream interface. The PT8402 also decodes MP3 and its main application is in portable MP3 players.

2.2.3 Cirrus

The CS49400 family [19] of multi-standard audio decoders (released 2002) is designed for use with DVD players and similar entertainment systems. Along with AAC, it decodes Dolby Digital EX^{TM} (DVD), MP3 and a host of other standards. The chip requires no external logic or memory despite its compatibility with all three AAC profiles.

2.2.4 Micronas GmbH

The MAS3509F [20] decodes the AAC LC Profile at a maximum sampling rate of 48kHz. The DSP core runs on a 2.5V power supply and has a D/A converter for a PCM output. The chip is an "all-in-one" solution but requires a small microcontroller for controlling communications.

2.2.5 ARM MOVE^{TM} Technology Audio Components

ARM have implemented a complete codec for an AAC decoder that is optimised for their 32-bit RISC processors. The ARM7TDMI, ARM9TDMI, ARM9E and StrongARM can run MOVE^{TM} Technology [21] and decode with sampling frequencies up to 48kHz. The processor cores are embedded in emerging portable Internet Audio devices.

2.2.6 Comparison and Conclusion

Table 2-1 summaries the commercial market for AAC decoders. The current technologies are implemented with either specific DSP cores or microprocessor cores. No AAC licensees have released any products that use either an FPGA core or an FPGA co-processor.

Producer	Processing Core	Max Sampling Frequency	Profiles
Texas Instruments	Fixed/Float DSP	96kHz	LC
Princeton	Fixed-point DSP	24kHz	LC
Cirrus	Fixed-point DSP	48kHz	LC
Micronas	Fixed-point DSP	48kHz	All
ARM MOVE Tech	RISC Processor	48kHz	All

Table 2-1: Comparison of commercial AAC decoders

2.3 Research publications and implementations

Several papers address implementations of the AAC LC profile with a fixed-point DSP. Lee, Jeong, Bang and Youn [22] designed a real-time system with a sampling rate of

48kHz but required that two external hard-wired logic modules be added for the Huffman decoding and prediction modules. These two compensations were required because of the high computational load of the Huffman decoding and the use of floating point arithmetic by the prediction tool.

Similarly, Chen and Tai [9] implemented an AAC LC encoder on a fixed-point TMS320C62x DSP. At a sampling rate of 48kHz, 9.2MIPS and 1.6M RAM were required for the full implementation – both within the capabilities of the C62x. The encoder was successful, but much time was spent to ensure that the fixed-point calculations were accurate. Modifications to some tools and approximations of algorithms in several tools were necessary to reduce memory and runtime demands while maintaining the integrity of the output bitstream.

In summary, the availability of research publications and implementations of the AAC algorithm in a hardware-based solution is limited. Some solutions built around DSPs provide some preliminary information about the advantages and difficulties of AAC decoding, but no relevant information on an FPGA-based solution, either as the core or as a co-processor, is available.

2.4 Comparison of DSP with FPGA

It is worthwhile to briefly introduce DSP technology and compare DSPs with FPGAs to give insight into the contexts that DSPs and FPGAs are used. The following comparison shows the strengths and weaknesses of an FPGA-based approach to decoding media such as an AAC bitstream, and why, to date, DSPs are more often chosen as the processing core for AAC decoders.

DSPs are specialised microprocessors used for signal processing applications. They are well suited to complex mathematical tasks and programs that require much conditional processing. DSPs are typically programmed using C or assembler.

DSP chips have dedicated arithmetic units that can be used as required such as adders and multipliers. Like regular microprocessors, DSPs run on a system clock, and the clock rate therefore limits the number of instructions and operations that can be carried out in any given time period.

Architecturally, FPGAs are very different from DSPs [23]. FPGAs consist of a "sea of gates", uncommitted in function, that can be configured into specific hardware blocks. Blocks range from simple registers, adders and multipliers to more complicated units that perform FIR filtering and FFTs. Instantiating multiple instances of the same units provides great scope for hardware solutions with large bandwidths and extensive parallel processing capabilities.

FPGAs offer many advantages over DSPs [24]. FPGAs have more internal multipliers than DSPs, and are able to handle higher sampling rates. FPGAs' capacity for parallel processing makes them clearly beneficial for high repetition tasks, such as multiply-accumulates (MACs). Good design can ensure that FPGAs have low power dissipation in comparison to DSPs offering a clear advantage for portable hardware devices.

FPGAs, however, are not entirely superior to DSPs. For programs that involve extensive conditional evaluations, FPGAs may require dedicated hardware resources for each possible configuration and datapath whereas DSPs can re-use processing units such as multipliers regardless of the system flow. DSPs are more efficient at implementing floating point arithmetic. DSPs' architectures make the translation of a software solution to a hardware solution easier. DSPs are optimised for the use of external memory, whereas FPGAs have only small internal memories and require additional external memory modules for large data sets.

Most of today's emerging digital audio processing applications, especially in the area of portable Internet Audio, are DSP based due to the strong software influence in coding the algorithms. FPGAs are more suited to simple, repetitive operations, while DSPs handle complex software problems to which they are more suited. In addition, small RISC microcontrollers are often required to handle communications and system control.

An ideal solution would consist of a hybrid of DSP, FPGA and microcontroller with each component dedicated to its area of strength [24].

2.5 Software Implementations

Though the objective of the project is to implement an algorithm in hardware, a survey of software implementations of AAC encoders, decoders, recorders and players is beneficial. Ensuring that the subject of AAC decoding has been considered in an appropriate scope provides a firm foundation for achieving the project objective. Both commercial and open source implementations are outlined.

In the commercial arena, Mayah Communications was the first company to release an AAC player and recorder [25]. The 'AAC-Recorder' performs WAV to AAC recording and Mayah mention that, as an outstanding product feature, AAC-Recorder is "fully software, no additional hardware necessary."

Apple Computers have recently adopted the MPEG-4 AAC standard into their QuickTime 6 media player [26]. The software is capable of a dull encode and decode of AAC. However, AAC's development in the commercial market is still in its early days.

There is also little development of the AAC algorithm in open source circles. Audiocoding.com [27] provide open source codecs for Advanced Audio Coding (MP4 and AAC). Their FAAC and FAAD2 programs encode and decode all profiles of AAC with "great accuracy". The code is written in C++ for compilation under either Windows or Linux, and is well linked to the ISO/IEC standards that it was based upon. Each module is individually coded and assembled into a LIB file that can be accessed by a command line front end. The source code is free and published for developers and other interested parties to enhance and modify.

In summary, FAAD2 provides a useful secondary reference to the ISO/IEC 13818-7 standard, but does redeem the relatively undeveloped field of software implementations of AAC to great usefulness.

2.6 Summary

This chapter has provided the details of the literature review carried out for the project. A survey of the well-known MP3 standard, and its well-documented successes in FPGA implementations, concluded that AAC, as an advancement over MP3, would be the most suitable algorithm to use as the basis of the project (section 2.1). Accordingly, the commercial (section 2.2) and academic (section 2.3) implementations of hardware-based AAC decoders were investigated, concluding that no details of implementations incorporating an FPGA core or FPGA co-processor were available. DSP technology was introduced and compared to FPGAs (section 2.4) to show why DSP-based solution are prevalent, and for completeness, a survey of software implementations of AAC was carried out (section 2.5).

3 MPEG-2 AAC DECODING ALGORITHM

This chapter overviews the MPEG-2 AAC decoding algorithm and tools as well as the official conformance test bitstreams and details the extrapolation of a block diagram for the decoder implementation. The block diagram provides the conceptual basis for the project.

3.1 Overview of the MPEG-2 AAC Decoding Algorithm

The MPEG-2 decoding algorithm consists of a number of coding tools that decode a 13818-7 AAC bitstream to its corresponding PCM values. Following is a brief description of the function of these tools, and Figure 3-1 shows the block diagram of the decoder.

The bitstream demultiplexer tool reads in a 13818-7 AAC bitstream and separates the data stream into the relevant sections for each of the decoding tools. Both control and data information are sorted, and information concerning the bitstream, such as profile type, sampling frequency index and copyright ID, is obtained.

The noiseless decoding tool reconstructs the quantised spectral data by parsing the information received from the bitstream demultiplexer and Huffman decoding it. The Huffman and DPCM (Differential Pulse Code Modulation) coded scalefactors are also reconstructed.

The inverse quantiser tool processes the quantised values of the spectral data and converts these integer values to the non-scaled, reconstructed spectra.

The scalefactor tool changes the integer representations of the scalefactors to their actual values (by a non-linear transformation) and scales the un-scaled inversely quantised spectra according to their respective scalefactors.

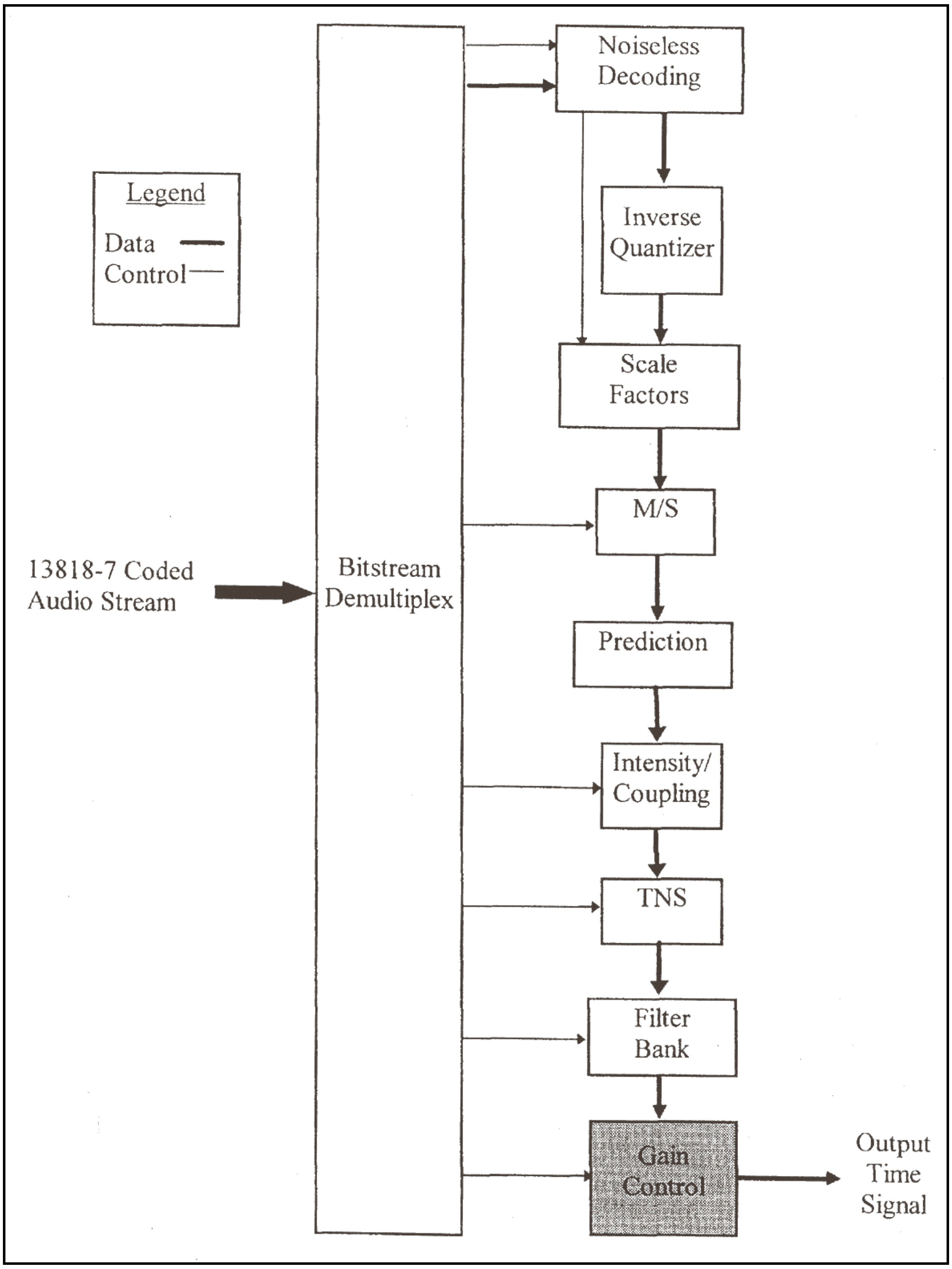


Figure 3-1: MPEG-2 AAC Decoder Block Diagram

The M/S tool decodes paired spectral values from Mid/Side to Left/Right. This is done under the control of the M/S decision information extracted from the bitstream.

The prediction tool reverses the prediction process carried out in the encoder. Prediction is a complicated and demanding encoder tool that, in simple terms, removes redundancy from the spectral data. This redundancy is re-inserted under the control of the predictor state information extracted from the bitstream.

The intensity stereo / coupling tool reinstates information that was coupled during the encoding process. When multiple channels are present and are similar, they are encoded as a single channel. Under the guidance of the coupling control information extracted from the bitstream, intensity stereo decoding is implemented.

The temporal noise-shaping (TNS) tool restores the actual shape of the temporal envelope that was flattened out in the encoding process. This reduces coding noise and is done under the control of the TNS information extracted from the bitstream.

The filter bank tool applies an inverse modified discrete cosine transform (IMDCT) to the spectral data to map the frequency values back to the time domain. This transformation is carried out according to the shape and sequence of the windows as extracted from the bitstream.

The gain control tool can be used in conjunction with the filter bank tool to apply separate time domain gain control to distinct frequency bands. This tool is used only for the scaleable sampling rate (SSR) profile.

The usage of these tools is determined by which profile is used for the encoded bitstream. The standard defines three profiles:

1. The main profile uses the full set of encoding and decoding tools and provides the best data compression possible. It is the most complex of the three profiles and is best suited to applications where memory cost is not significant, and where substantial processing power is available.

2. The low complexity profile (LC) does not allow the use of the prediction and gain control tools, and uses a limited TNS order. The LC profile is suited to applications where RAM usage, processing power and compression requirements are restricted.
3. The scaleable sampling rate profile (SSR) requires the gain control tool but does not permit prediction or coupling. The SSR profile is most appropriate in applications with a reduced audio bandwidth.

Table 3-1 summaries the use of the decoder tools.

Tool Name	**Required/Optional**	**Main**	**LC**	**SSR**
Bitstream Formatter	Required	✓	✓	✓
Noiseless Decoding	Required	✓	✓	✓
Inverse Quantiser	Required	✓	✓	✓
Scalefactors	Required	✓	✓	✓
M/S	Optional	✓	✓/✗	✓/✗
Prediction	Optional	✓	✗	✗
Intensity/Coupling	Optional	✓	✓/✗	✗
TNS	Optional	✓	✓	✓
Filter bank	Required	✓	✓	✓
Gain Control	Optional	✗	✗	✓

Table 3-1: Summary of MPEG-2 AAC Decoder Tools

The profile that is best suited for the project is the low complexity profile (LC). This profile offers high audio quality but makes smaller demands on processing resources than the main profile. As audio bandwidth will not be an issue, the SSR profile is not appropriate.

3.2 *ISO/IEC 13818-7 Audio Test Bitstreams*

AT&T Corporation has written audio conformance bitstreams for AAC developers [28]. These bitstreams are available for all three AAC profiles and come in multiple configurations. The configurations for the LC profile are tabled in Appendix A.

A suitable audio conformance bitstream is chosen for the practical purposes of testing the decoder. The selected bitstream satisfies one primary condition – that it can test the functionality of the decoder's tools in small increments. Accordingly, 'L1_fs' was chosen as the test bitstream. L1_fs is encoded with only the required tools, and an implementation of an AAC decoder capable of decoding L1_fs would provide the best foundation for incorporating the remaining LC profile tools (TNS, intensity, M/S). The sampling frequency of L1_fs can range from 8kHz to 96kHz.

3.3 *Extrapolation of Block Diagram*

From an examination of the decoding tools of the AAC algorithm and the elements within the L1_fs conformance stream, a block diagram of the required decoder is extrapolated. The implementation contains the minimum number of decoding tools but will successfully decode the L1_fs bitstream. Figure 3-2 shows the block diagram.

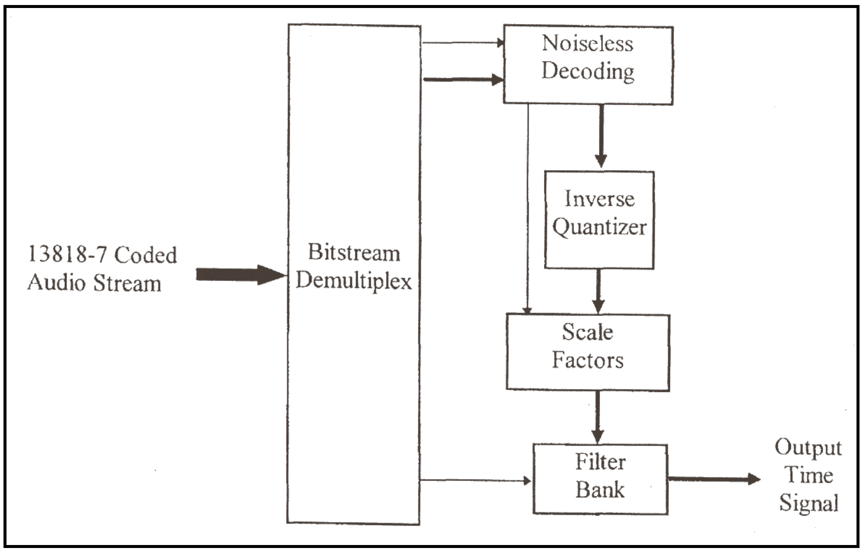


Figure 3-2: Block Diagram for Decoder Implementation

3.4 Summary

This chapter has outlined the MPEG-2 AAC decoding algorithm by surveying each of the decoding tools and each of the three profiles (section 3.1). Also, the official conformance bitstreams provided by AT&T were surveyed (section 3.2). In consideration of the decoding profiles and the conformance bitstreams, the conformance bitstream 'L1_fs' was chosen as the bitstream for decoding, and a block diagram of the decoder was extrapolated (section 3.3).

4 A CASE STUDY: ‘L1_fs_mod’

To provide a greater understanding of the decoding algorithm, the conformance bitstream ‘L1_fs’ was partially decoded by hand. A record of the decoding of the first 411 bytes is included as Appendix B and is a vital reference for checking the results of the output of the various stages of the decoder implementation. This chapter details the processes and results of decoding the AAC bitstream ‘L1_fs_mod’.

4.1 Decoding the header

The header determines the structure of an AAC audio bitstream. There are two basic formats for the bitstream – Audio Data Interchange Format (ADIF) and Audio Data Transport Stream (ADTS) – and the use of a particular format is determined by the application. ADIF is used when decoding only occurs from the start of the bitstream and never from within, such as decoding from a disk file. Thus ADIF contains one header at the start of the file. ADTS, on the other hand, contains multiple headers that change from frame to frame to allow the bitstream to be decoded from any point within it. ADTS is suited to streaming applications. L1_fs_mod is an ADIF bitstream, and the hierarchy of L1_fs_mod as an ADIF bitstream is shown in Figure 4-1.

```
Adif_sequence()
    Adif_header()
        Program_config_element()
    Byte_alignment()
    Raw_data_stream()
        Raw_data_block()
            Data_stream_element()
            Single_channel_element()
                Individual_channel_stream()
                    Ics_info()
                    Section_data()
                Scale_factor_data()
                Spectral_data()
        Fill_element()
        Terminator()
    Byte_alignment()
    Raw_data_block()
        Data_stream_element()
        Single_channel_element()
            Individual_channel_stream()
                Ics_info()
                Section_data()
                Scale_factor_data()
                Spectral_data()
        Terminator()
    Byte_alignment()
```

Figure 4-1: L1_fs_mod Hierarchy

Decoding the header reveals that L1_fs_mod is indeed a 48kHz LC profile bitstream. There is a single front channel element that will map to a centre front speaker for audio playback.

The syntactical elements of the partial bitstream are <DSE> <SCE> <FIL> <TERM> <DSE> <SCE> <TERM>. Table 4-1 summarises these elements.

Element	Abbrev	Description
Single Channel Element	<SCE>	Contains coded data for a single audio channel
Data Stream Element	<DSE>	Contains extra data not related to the audio
Fill Element	<FIL>	Contains fill data to adjust the data rate
Terminator	<TERM>	Indicates the end of a data block

Table 4-1: Summary of syntactical elements

4.2 Decoding <DSE>

‘A data element contains any additional information, e.g. auxiliary information, that is not part of the audio information itself,’ [6, p32] and can contain up to 512 bytes of data. The decoding process for a <DSE> is described clearly by subclauses 6.3 (Table 6.20) and 8.6 of ISO/IEC 13818-7. The two data elements within L1_fs_mod (one <DSE> in each raw data block) each contain two bytes of data [00 AB and 00 9D].

4.3 Decoding <SCE>

‘A single channel element is composed of an element instance tag and an individual channel stream’ [6, p25]. It is the most complex syntactical element in L1_fs_mod and contains all the data necessary to decode one channel. A <SCE> comprises of four main sections – ics_info(), section_data(), scale_factor_data() and spectral_data() – but also makes provision for some optional sections – tns_data(), pulse_data() and gain_control_data(). The decoding process for a <SCE> is described by subclauses 6.3 (Tables 6.9, 6.12) and 8.3 of ISO/IEC 13818-7. The separate processes for decoding the sections within a <SCE> follow.

4.3.1 Decoding ics_info()

Ics_info() carries window information associated with an individual channel stream (ICS). The need for windowing arises from the use of the quantisation encoding tool. Quantisation is done in the frequency domain, but obviously, the sampled signal is represented in the time domain. Under the control of a modified discrete cosine transform (MDCT), the encoder can change its time/frequency resolution by using two different windows – a LONG_WINDOW and a SHORT_WINDOW. A long window comprises of 1024 coefficients and a short window comprises of 128 coefficients. A raw data block always contains data representing 1024 coefficients, and accordingly, one long window is equivalent to eight short windows.

Table 4-2 shows the transform windows for 48kHz [6, p33]. Note the inclusion of LONG_WINDOW_START and LONG_WINDOW_STOP that provide meaningful transitions between the long and short windows.

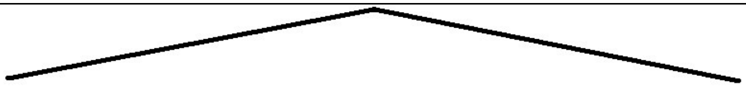
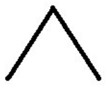
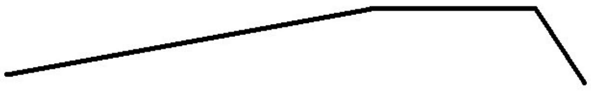
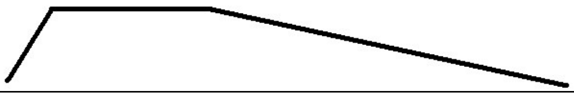
Window	**Looks like…**
LONG_WINDOW	
SHORT_WINDOW	
LONG_START_WINDOW	
LONG_STOP_WINDOW	

Table 4-2: Transform window for 48kHz

Along with providing the window_sequence, ics_info() carries information about the window_shape. The window shape is used by the IMDCT, together with the window sequence, to select the window configuration. When the window shape is 1, the IMDCT employs a Kaiser-Bessel derived (KBD) window, and when window shape is 0, the IMDCT uses a sine window. The decision on window shape is made during encoding, and the decoder does not need to know why a particular window configuration has been chosen.

The first raw data block of L1_fs_mod has a long start window sequence and uses a sine shaped window. The second raw data block has an eight short window sequence and also uses a sine shaped window. Figure 4-2 shows the combination of these two blocks.

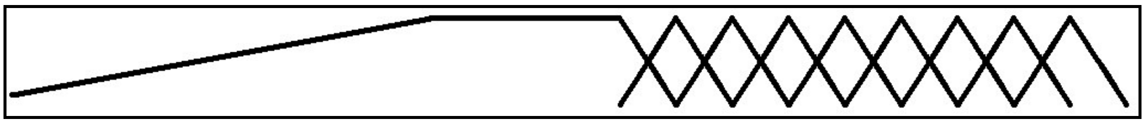


Figure 4-2: Window sequence for L1_fs_mod

Aside from describing the window sequence and shape, ics_info() contains vital information concerning scalefactor bands. Simply, a scalefactor band is a group of consecutive spectral values, with the width of the grouping calculated from the critical bands of the human auditory system. Thus, the number of scalefactor bands in a spectrum depends upon the sampling frequency (in this case, 48kHz) and the transform length (1024 or 128).

Ics_info() specifies the maximum number of scalefactor bands to reduce the transmission of spectral data relating to inactive scalefactor bands. Ics_info() also specifies window grouping for short window sequences. Windows that are grouped together have the same scalefactors applied to them in order to reduce the amount of side information transmitted.

The first raw data block of L1_fs_mod has a maximum of 41 scalefactor bands, and has no window grouping because it is a long window sequence. The second raw data block has a maximum of 11 scalefactor bands, and its eight short windows are grouped together {3, 1, 4}. Subclause 8.3.5 of ISO/IEC 13818-7 clearly describes how the spectral coefficients are ordered according to scalefactor bands and windows groupings, and Figures 4-3 and 4-4 show the ordering for the two raw data blocks.

sfb 0	sfb 1	sfb 2	. . .	sfb 39	sfb 40

Figure 4-3: Spectral co-efficient ordering for the first raw data block

<-------- sfb 0 -------->			<-------- sfb 1 -------->			.	.
win 0	win 1	win 2	win 0	win 1	win 2		

<-------- sfb 10 -------->			sfb 0	sfb 1	.	.	sfb 10
win 0	win 1	win 2	win 3	win 3			win 3

<--------------- sfb 0 --------------->				<--------------- sfb 1 --------------->			
win 4	win 5	win 6	win 7	win 4	win 5	win 6	win 7

.	.	<-------------- sfb 10 -------------->			
		win 4	win 5	win 6	win 4

Figure 4-4: Spectral co-efficient ordering for the second raw data block

The final set of information retrieved from ics_info() is derived from the sampling frequency, the window sequence, the scalefactor grouping and the maximum number of scalefactor bands. These derived 'help' variables and arrays are required to further describe the scalefactor band arrangement for all tools using scalefactor information. Subclause 8.3.4 of ISO/IEC 13818-7 clearly describes how the additional variables are derived.

In summary, ics_info() provides information regarding the window shape and sequence as well as information on scalefactor bands and window grouping. The decoding process is according to subclauses 6.3 (Table 6.11) and 8.3 of ISO/IEC 13818-7.

4.3.2 Decoding section_data()

Section_data() describes the Huffman codes that apply to the scalefactor bands in the individual channel stream. The sectioning data describes firstly the codebook that is used, and then the length of the section that is coded with that codebook, starting from the first scalefactor band and continuing until all scalefactor bands are completed. Subclauses 6.3 (Table 6.13) and 8.3.2 of ISO/IEC 13818-7 describe this decoding.

Huffman coding is a form of lossless entropy encoding based on the probabilities of certain data inputs occurring. It takes a block of input data of fixed length and produces a block of output data of variable length. Short codewords are assigned to data that occurs frequently, and longer codewords are assigned to data that occurs less frequently. Thus, a Huffman 'code tree' is constructed and this is the basis for a Huffman codebook.

MPEG-2 AAC uses eleven spectral Huffman codebooks for encoding the spectral data. Codebooks can be signed or unsigned, and be of two or four dimensions. The encoder makes the decisions about which codebook to use for which spectral values, and the section_data() simply carries the information regarding these decisions so that the Huffman codewords can be decoded.

The first raw data block of L1_fs_mod is encoded entirely with Huffman codebook zero. Therefore, no scalefactor data or spectral data is transmitted. The second raw data block is encoded in many different sections and with many different codebooks. Figures 4-5 and 4-6 show the codebook sections for the two raw data blocks.

g=0	<--	codebook 0	-->			
	sfb 0	sfb 1	sfb 2	. .	sfb 39	sfb 40

Figure 4-5: Codebook sections for the first raw data block

g=0	<--				codebook 0		-->				
	sfb 0	sfb 1	sfb 2	sfb 3	sfb 4	sfb 5	sfb 6	sfb 7	sfb 8	sfb 9	sfb 10
g=1	cb 10	<--------------------------------------				codebook 4		------------------------------------->			
	sfb 0	sfb 1	sfb 2	sfb 3	sfb 4	sfb 5	sfb 6	sfb 7	sfb 8	sfb 9	sfb 10
g=2	codebook 11			codebook 6				cb 8	codebook 6		
	sfb 0	sfb 1	sfb 2	sfb 3	sfb 4	sfb 5	sfb 6	sfb 7	sfb 8	sfb 9	sfb 10

Figure 4-6: Codebook sections for the second raw data block

4.3.3 Decoding scale_factor_data()

For each scalefactor band that is not coded with the zero codebook, a scalefactor is transmitted. (Direct quote). Scalefactors are used to further shape the noise injected by quantisation. They change the amplitude of all the spectral coefficients within a scalefactor band. Like the spectral data, the scalefactors are Huffman coded to increase the data compression, and for this purpose there is a special scalefactor Huffman codebook.

As well as being Huffman coded, the scalefactors are also differentially coded. The global gain (a variable retrieved at the start of the individual channel stream) is the first active scalefactor; the second scalefactor encoded is encoded relative to the first, the third to the second and so on.

The process for decoding scale_factor_data() is described in subclauses 6.3 (Table 6.14), 8.3.2 and 11.3.2 of ISO/IEC 13818-7. However, the description is not entirely explicit, and an examination of the source code of the FAAD2 decoder provides a more intuitive explanation of how the scalefactor data is retrieved. The following code segment (Figure 4-7) shows how FAAD2 [27] identifies and decodes the Huffman codewords.

```
int i, j;
unsigned long cw;
codebook *h;

h = book_table[cb];
i = h->len;
cw = faad_getbits(ld, i DEBUGVAR(0,0,""));

while (cw != h->cw)
{
    h++;
    j = h->len-i;
    i = h->len;
    if (j!=0) {
        while (j--)
            cw = (cw<<1) | faad_get1bit(ld DEBUGVAR(0,0,""));
    }
}
return h->scl;
```

Figure 4-7: FAAD2 code for retrieving scalefactor data

Briefly, the code segment steps through the consecutive entries of the scalefactor codebook and tries to match the codeword to the bitstream. When a matching codeword is found, the differentially coded scalefactor is retrieved and the scalefactor is calculated relative to the previous (not shown). The bitstream is examined until all the scalefactor codewords have been read.

Only groups one and two of the second raw data block of L1_fs_mod have scalefactors transmitted because group zero, and the entire first block, are encoded with the zero codebook. For details on how the scalefactors are applied to the spectral data, see section 4.8.

4.3.4 Decoding spectral_data()

Spectral_data() consists of all the non-zeroed coefficients remaining in the spectrum and is ordered as described in ics_info(). (Direct quote). Like scale_factor_data(), the first task is to retrieve the Huffman codewords. To do this, the sectioning information from section_data() regarding the codebook number and section length is used. Depending on the codebook, two or four spectral coefficients are decoded.

The second decoding task is setting the sign magnitude and is dependent upon the spectral values and the codebook used. For unsigned codebooks, sign bits are decoded following the codeword and applied to the spectral values. No sign bits are transmitted when the spectral values are zero, or a signed codebook is used.

The third decoding task is related only to the use of codebook eleven. Codebooks one through ten allow the encoding of spectral values with an absolute value no greater than 12. By using codebook eleven (the ESCAPE codebook), the largest absolute value is increased to 8191 through the addition of an escape sequence.

Though this decoding process is described in subclauses 6.3 (Table 6.16), 8.3.2 and 9.3 of ISO/IEC 13818-7, it is beneficial to supplement the description with the relevant source code from the FAAD2 decoder. The details of the decoded, signed spectral values are included in the Appendix B.

4.4 Decoding <FIL>

Fill elements have to be added to the bitstream if the total bits for all audio together with all additional data is lower than the minimum allowed number of bits in the frame necessary to reach the target bitrate. (Direct quote) More simply, a fill element is included to adjust the instantaneous bitrate for a constant rate channel. Practically, in L1_fs_mod, a <FIL> element in included when an entire <SCE> is encoded with the zero Huffman codebook (ZERO_HCB). Elements encoded with ZERO_HCB are not transmitted with any spectral data or scalefactor data, and hence are transmitted with a <FIL> element to compensate for their relatively small size.

The original 13818-7 specification of the <FIL> element was relatively simple, but was revised by 13818-7 Technical Corrigendum 1 to include Dynamic Range Control (DRC). The DRC model increases the complexity of decoding a <FIL> element, and this decoding process is now described by subclauses 6.3 (Tables 6.22, 6.24-26) and 8.7 of ISO/IEC 13818-7/Cor.1.

The single <FIL> element in L1_fs_mod is 1300 bits long – 868 bits less than the maximum size for a <FIL> element. Its configuration does not use DRC, but is rather the default <FIL> element format. By comparison, the 1300 bits of <FIL> in the first raw data block that is included to compensate for the ZERO_HCB coded <SCE> is equivocated by the 1250 bits (approx) of spectral data and scalefactor data in the second raw data block. Thus, the <FIL> element's role in adjusting the instantaneous bitrate is demonstrated.

4.5 Decoding <TERM>

The terminator element is transmitted to signal the end of a raw data block. It is decoded simply according to subclause 6.3 (Table 6.8) of ISO/IEC 13818-7. There is exactly one <TERM> element per raw data block.

4.6 Noiseless Coding

Noiseless Coding is the term given to the process of constructing of the spectral coefficients from the spectral data components in the bitstream. Noiseless coding requires the Huffman coded spectral values, the sign bits and any escape sequences to produce two or four (depending on the codebook number) spectral coefficients. This decoding procedure is described by subclause 9.3 of ISO/IEC 13818-7.

Noiseless coding is done in tandem with decoding spectral_data(), and this is essential. The two processes are distinctly defined, but in practice, are done dependently upon one another. Table 4-3 attempts to show the manner in which they interact. Note particularly that the magnitude of the coefficients must be known before the bitstream decoder can make a decision about how many sign bits to look for.

Spectral_data() decoding	**Noiseless Coding**
Read Huffman codeword	
	Decode Huffman codeword and get the magnitude information of the coefficients
For each non-zero value, get a sign bit (if using an unsigned codebook) If using cb11, get escape sequence information	
	Apply sign information to coefficients Apply escape sequence information

Table 4-3: The interaction of spectral_data() decoding and noiseless coding

Inspection of the FAAD2 decoder source code sheds further light on the interaction of the spectral_data() decoding and the noiseless coding. One notable difference in the FAAD2 implementation is the rearrangement of the Huffman codebooks. ISO/IEC 13818-7 specifies the 11 codebooks using four variables: *dim* (the dimension of the codebook – either two or four), *lav* (the largest absolute value that the codebook can encode), *unsigned* (whether or not the codebook is signed) and *idx* (the codeword index). An algorithm is then required to translate this code to the spectral coefficients. In contrast, the FAAD2 decoder stores the Huffman codebooks with index, codeword, codeword length and the actual spectral coefficients. The FAAD2 implementation requires more memory resources but saves on computation time. However, this does demonstrate the interdependence of the two decoding processes.

4.7 Quantisation

During encoding, the spectral coefficients are encoded with a non-uniform quantiser. Therefore it follows that this quantisation must be reversed in decoding, and this is done after the Huffman decoded spectral values are obtained. Inverse quantisation is done according to the following formula:

$$x\_invquant = Sign(x\_quant) \cdot |x\_quant|^{\frac{4}{3}} \quad \forall \; k$$

The value of x_quant is determined by systematically stepping through the interleaved array of spectral data – each coefficient in each scalefactor in each window band in each group. Clause 10 of ISO/IEC 13818-7 describes this decoding process.

4.8 Scalefactors

Scalefactors are transmitted as Huffman coded, differentially coded integers. Once decoded to their integer values (see section 4.3.3), the scalefactors undergo a non-linear transformation before being applied to the inversely quantised spectral data. As mentioned in section 4.3.1, scalefactors are applied to scalefactor bands, and the formula for this non-linear transformation reflects this and is as follows:

$$gain = 2^{0.25 \cdot (sf[g][sfb] - SF\_OFFSET)}$$

The constant SF_OFFSET is always set to 100. Using the same systematic stepping through the interleaved array as for the quantisation decoding, the spectral values are multiplied by the gain to give a rescaled set of data. Subclause 11.3.3 of ISO/IEC 13818-7 describes the application of the scalefactors.

4.9 Filter Bank

The final step in decoding is the transformation of the time-frequency signal back to the time domain, and this is done under the control of an inverse modified discrete cosine transform (IMDCT) in the Filter Bank tool. There are a number of considerations that must be taken into account for this transformation, and these will be highlighted. The decoding process is described in subclauses 9.3 and 15.3 of ISO/IEC 13818-7.

Before the IMDCT can begin, the spectral coefficients, having been inversely quantised and scaled, must be arranged in a non-interleaved fashion. Previous decoder tools have used a four dimensional interleaved array (see Figure 4-3, 4-4 in section 4.3.1) based on group, scalefactor band, window and coefficient. The IMDCT requires that the spectral coefficients be arranged according the their window number and frequency within the window. The C-code section [6, p42] in Figure 4-8 shows the separation of the interleaved coefficients x_quant[][][][] to a non-interleaved format spec[][].

```
void quant_to_spec (void) {

   unsigned int g,sfb,width,win,bin,j,k = 0;

   k = 0;
   for (g = 0; g < num_window_groups; g++) {
      j = 0;
      for (sfb = 0; sfb < max_sfb; sfb++) {
         width = (swb_offset[g][sfb + 1] - swb_offset[g][sfb]);
         for (win = 0; win < window_group_len[g]; win++) {
            for (bin = 0; bin < width; bin++) {
               spec[win+k][bin+j] = x_quant[g][sfb][win][bin];
            }
         }
         j += width;
      }
      k += window_group_len[g];
   }
}
```

Figure 4-8: C-code for de-interleaving the spectral coefficients

The non-interleaved array spec[][] is the basis for the IMDCT. The analytical expression of the IMDCT is:

$$x_{i,n} = \frac{2}{N}\sum_{k=0}^{\frac{N}{2}-1} spec[i][k]\cos\left(\frac{2\pi}{N}(n+n_0)\left(k+\frac{1}{2}\right)\right) \quad \text{for} \quad 0 \le n < N$$

where n = sample index

N = transform block length

i = block index

k = spectral coefficient index

n_0 = (N/2 + 1) / 2.

Once the complete set of values $x_{i,n}$ are obtained, a window function is applied – either KBD (Kaiser-Bessel derived) or SINE – according to the window sequence and window shape retrieved from ics_info(). These window functions are complicated and will not be detailed. Full details are found in subclause 15.3.2 of ISO/IEC 13818-7.

Finally, an overlap/add function is used to produce the PCM output values. These values can be passed directly to a 16-bit digital-to-analog (D/A) converter. Full details are found in subclauses 15.3.3 and 8.3.6 of ISO/IEC 13818-7.

4.10 Summary

This chapter has briefly outlined the decoding procedure and results of decoding L1_fs_mod – the first 411 bytes of conformance bitstream L1_fs (sections 4.1-4.9). These results are documented in Appendix B and are a useful reference for understanding the algorithms and tools as well as for testing the integrity of the output values of an implementation.

5 HARDWARE ENVIRONMENT

This chapter provides a survey of the hardware environment of the project. The hardware allocated for the project was a Xess XSV300 board [29]. The main features of this board that are used for the project are the Xilinx XCV300 FPGA, the AK4520A stereo A/D & D/A converter, and the XC95108 CPLD and parallel port. The board was interfaced to the PC and configured using the XSTools package and the hardware description was written using VHDL. Simulation, synthesis and implementation of the VHDL source code were done using Active-HDL and the Xilinx Foundation F3.1i software package.

5.1 Xilinx XCV300 FPGA

The XCV300 FPGA [30] is a member of Xilinx's Virtex series of FPGAs. It is the main repository of reprogrammable logic for the implementation of the decoder, and has 323K system gates. These system gates are arranged in logic modules (Figure 5-1) in the form of combinational and sequential circuits that implement logic functions [31]. Logic modules are routed to I/O modules using pre-fabricated wire segments and programmable switches. The Xilinx Foundation software package (see section 5.5) is used to configure the FPGA's programmable logic.

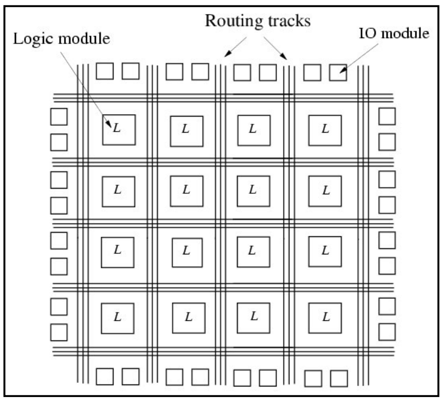


Figure 5-1: FPGA Architecture

5.2 AK4520A stereo codec

The AK4520A stereo codec [32] is used to convert a serial digital audio stream to an analog signal and vice versa. The AK4520A is controlled by the FPGA, requiring three clocks to function – MCLK, SCLK and LRCK. The datasheet specifies that MCLK and

SCLK are a function of the sampling frequency, in this case, 256fs and 64fs respectively.

The oscillator on board the XSV board is 100MHz. Using a simple divisor of 2048 (or 2^{11}), the sampling frequency will be 48.828kHz. This determines MCLK to be 12.5MHz and SCLK to be 3.125MHz.

The AK4520A was not used in the implementation due to the progress on the decoder. However, James Brennan's audio interface [33] would have been adapted to control the chip for D/A conversion of the output signal.

5.3 XC95108 CPLD

The XC95108 CPLD (Complex Programmable Logic Device) [34] is used to manage the configuration of the XCV300 FPGA. Programming is done via the parallel port. It is possible to alter the configuration of the CPLD to load a bitstream from the onboard Flash RAM and program the FPGA with the loaded bitstream [35]. This overcomes the problem of the FPGA losing its configuration each time that it is turned off, but such an alteration is unnecessary for the project at this stage. (See section 8.1.4 for more details.)

5.4 VHDL and Active-HDL

VHDL (Very High Speed Integrated Circuit Hardware Description Language) is a language for describing the structural and behavioural characteristics of a digital system [36]. A structural description is defined using component instantiation and mapping the I/O ports of components together. A behavioural description is described by writing processes on both a sequential and concurrent level. VHDL, in combination with Programmable Logic Arrays (PLAs), allows for rapid prototyping and reconfiguration of hardware devices, reducing time and effort.

There are two main reasons why VHDL was chosen as the programming language for the project. Firstly, previous subjects have provided a strong introduction to VHDL. Secondly, all the resources required to program in VHDL (Active-HDL, Foundation etc.) are readily accessible in the school laboratories. Though other HDLs (e.g. Verilog) and alternate languages (Handel-C) do exist, VHDL is used to write code for the FPGA for the project.

Active-HDL was the VHDL development environment used. Active-HDL provides control tools for design management, design entry tools for VHDL modules and test benches, debugging tools for monitoring the source code execution and powerful simulation tools for producing waveforms and verifying the code's functionality.

5.5 Xilinx Foundation

The Xilinx Foundation F3.1i software package "converts" a hardware description into a bitstream that is used to configure the system gates on an FPGA or CPLD. The two important functions of Foundation concerning the project are the synthesis and implementation functions. These functions are very complicated and will only be overviewed briefly.

Synthesis (Figure 5-2) is the process of translating the design (VHDL description) into gates and optimising it for the specific target architecture (in this case, a Virtex XCV300) [37]. A structural netlist is created and is used as the basis of implementation. Additional constraints can be specified such as mapping constraints – how the blocks are mapped to the logic modules, block placement constraints – where the blocks can be placed, and timing constraints – set timing requirements for particular data paths.

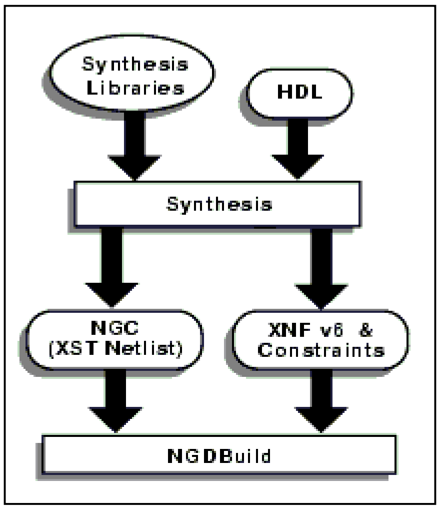


Figure 5-2: Synthesis Flowchart

Implementation (Figure 5-4) has three major functions – fitting the design into the specified device, routing the physical design and generating a bitstream [38]. NGDBuild accomplishes the first function by reading the netlist from synthesis and creating a logical design in terms of logic elements such as AND gates, OR gates, decoders and flip-flops etc. The NGD output file is then mapped on to the FPGA. The mapping output is a Native Circuit Description (NCD) file.

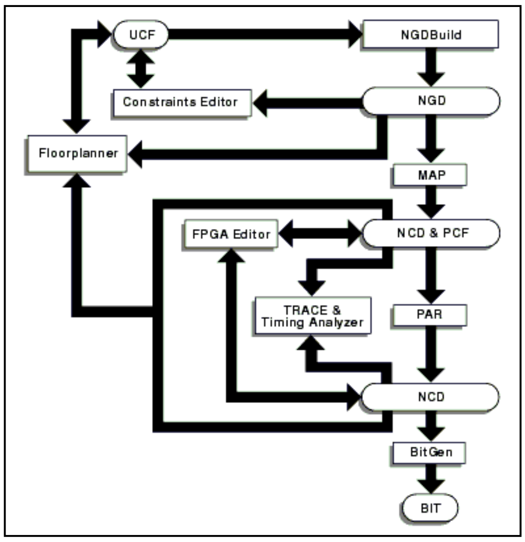


Figure 5-3: Implementation Flowchart

The second function of implementation is placement and routing. It adds information to and modifies the NCD file by placing the logical elements and routing the logical design.

The third function of implementation is to generate a bitstream for the configuration of the FPGA. BitGen takes a fully routed NCD file and creates a .bit file containing all the configuration information defining the internal logic and interconnections of the FPGA. The bitstream file is then ready for download to the XCV300 FPGA using the XSTools' GSXLOAD.

5.6 XSTools

The XSTools package [39] is used to provide access to the XSV board's programmable components (FPGA, CPLD, Oscillator, RAM etc.) using the PC parallel port. There are four basic tools that each performs a useful function:

- GXSLOAD: Loads configuration bitstreams (for more details, see section 5.5) into the FPGA or the CPLD through the parallel port. Without a JTAG interface, this program replaces the 'program' function in Xilinx Foundation. Also provides access to either upload or download the RAM and Flash RAM on the board. (See Figure 5-5)

- GXSSETCLK: Sets the frequency of the oscillator on the board.
- GXSPORT: Used to send a byte of data through the PC parallel port to the XSV Board.
- GXSTEST: Runs a diagnostic test on the XCV300 FPGA and XC95108 CPLD. Also tests the parallel connection.

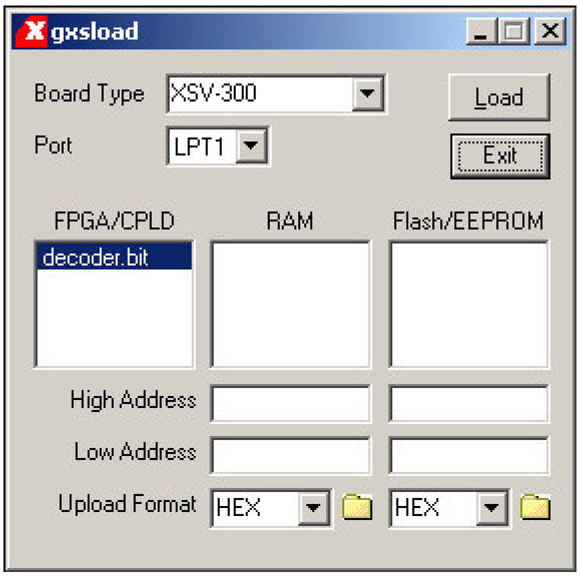

Board Type: XSV-300
Port: LPT1
FPGA/CPLD: decoder.bit
Upload Format: HEX, HEX

Figure 5-4: GXSLOAD User Interface

5.7 Summary

This chapter has provided an overview and explanation of the hardware components (sections 5.1-5.3) of the AAC decoder and the software packages used to design and implement the hardware description of the AAC decoder's functionality (sections 5.4-5.6). The XSV-300 board is well suited as a prototyping environment for the decoder, with the XCV300 FPGA, AK4520A stereo codec and parallel programming capacity of the XC95108 CPLD all ideal for the VHDL implementation.

6 ALGORITHM IMPLEMENTATION

The tools required in the decoder's datapath were outlined in chapter three – the bitstream demultiplexer tool, the noiseless coding tool, the quantisation tool, the scalefactor tool and the filter bank tool. The tools' application to a specific conformance bitstream – 'L1_fs_mod' – was detailed in chapter four. The hardware environment for the implementation of the algorithm was outlined in chapter five.

This chapter details the work done toward an implementation of the AAC decoding algorithm with VHDL. In consideration of the hardware environment, the techniques for coding the required tools are discussed. When difficulties are encountered, design decisions are justified and when appropriate, solutions are presented.

6.1 Bitstream Demultiplexer

The bitstream demultiplexer was designed to decode the L1_fs_mod bitstream as detailed in chapter four. The bitstream consists of a header followed by the syntactical bitstream elements <DSE> <SCE> <FIL> <TERM> <DSE> <SCE> <TERM>. A further constraint on the demultiplexer is that the bitstream should be decoded in real-time, and at 48.8kHz, a data bit arrives every 20.48μs.

The complexity of the 13818-7 bitstream is in its dynamic coding. On many occasions, information describing the format of the next bits is gathered from the values of the bits currently being processed. The vast number of decisions and computations made in between receiving bits complicates a state machine implementation of the demultiplexer.

The approach implemented to handle the dynamic bitstream was coupling the state machine with a programmable counter. The counter has a countdown range of 2048 (11-bit) given that the largest single block of data of L1_fs_mod to be received at once is the 1300 bit payload of the <FIL> element. The maximum payload for a <FIL>

element is 2168 bits, and a full implementation of the bitstream demultiplexer would require a 12-bit counter. However, for the chosen context, 11 bits is sufficient.

Figure 6-1 shows the general flowchart for the state machine's interaction with the counter.

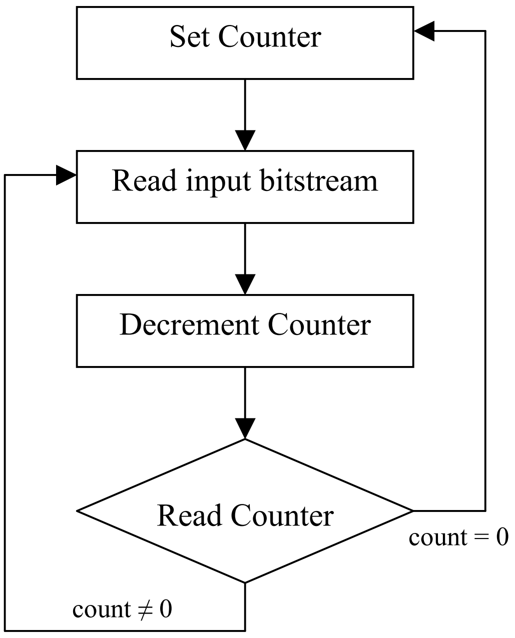


Figure 6-1: State-machine and counter flowchart

The third component of the bitstream demultiplexer is a simple SIPO (serial in, parallel out) shift register. The shift register is 32 bits long. The length of the register was determined by the maximum size data variable arriving in the bitstream that was required to be stored or used in a computation. The escape sequence data variables associated with Huffman codebook eleven have a maximum length of 31 bits. A 32-bit shift register will accommodate a variable of such length. While other data variables are longer than 32 bits (e.g., the <FIL> element payload is 1300 bits), they are not required for any computation and therefore are not necessary to retain.

Following is a description of the approaches incorporated while coding the individual syntactic elements of the bitstream demultiplexer in VHDL. The VHDL source code files (bitstream_top.vhd, bitstream.vhd, counter.vhd and shift_register.vhd) are included as Appendices C.1, C.2, C.3 and C.4.

6.1.1 The header

A minimalist approach was taken for coding the header decoder. For the purposes of implementing the basic AAC decoder with L1_fs_mod as the input bitstream, none of the header information needs to be retained as it is already known from the results of the work outlined in chapter four and the syntactical description of the file outlined in chapter three. The state machine counter simply counts through the header (608 bits) and the state machine then begins decoding raw data blocks.

6.1.2 <DSE>

The data stream element was coded in three states. There is one decision that must be made while decoding <DSE>. After an examination of the 8-bit variable **count**, a decision must be made to determine whether the following bits are either **esc_count** or the first of the **data_stream_byte**s. Figure 6-2 shows the <DSE> state diagram.

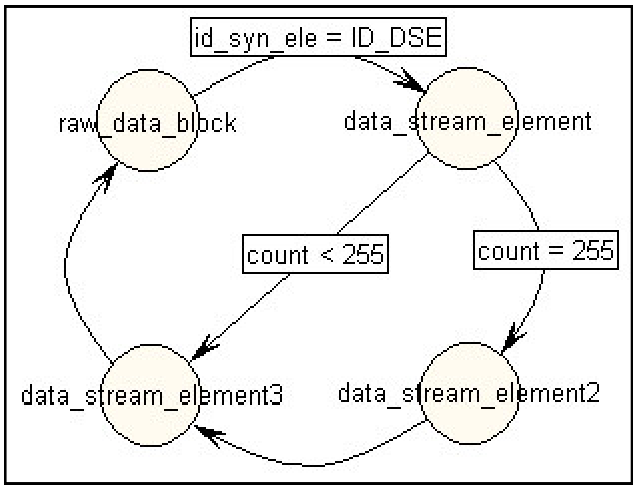


Figure 6-2: <DSE> state diagram

Adding to the complexity of decoding a <DSE> element is the presence of a byte_alignment() function. Simply, the function checks whether the next bit will be the start of a byte, and if not, packs in redundant data until it is. To incorporate the capability to perform byte alignments, the 32-bit shift register was modified to include a counter of its own. Together with the parallel output of data, the register has a countdown counter indicating how many bits remain until the start of a new byte. Practically, starting from seven, the counter counts down to zero and then starts counting again from seven. In the case that a byte_alignment() is required, the state machine simply adds to the programmable counter the value of the alignment counter produced by the shift register.

6.1.3 <SCE>

The single channel element is by far the most complicated element to decode in the L1_fs_mod bitstream. As outlined in chapter four, the <SCE> has four main sections – isc_info(), section_data(), scale_factor_data() and spectral_data(). Of these four sections, only the decoding of ics_info was described in VHDL. An explanation of the difficulties encountered in coding the other three sections also follows.

Isc_info() was coded in three states because it requires one decision to be made. **Global_gain**, **ics_reserved_bit**, **window_sequence** and **window_shape** are all decoded at one time, and based on the value of **window_sequence** (either a long window or a short window), the size of **max_sfb** is determined as is the presence of **scale_factor_grouping** information. Figure 6-3 shows the state diagram for decoding ics_info().

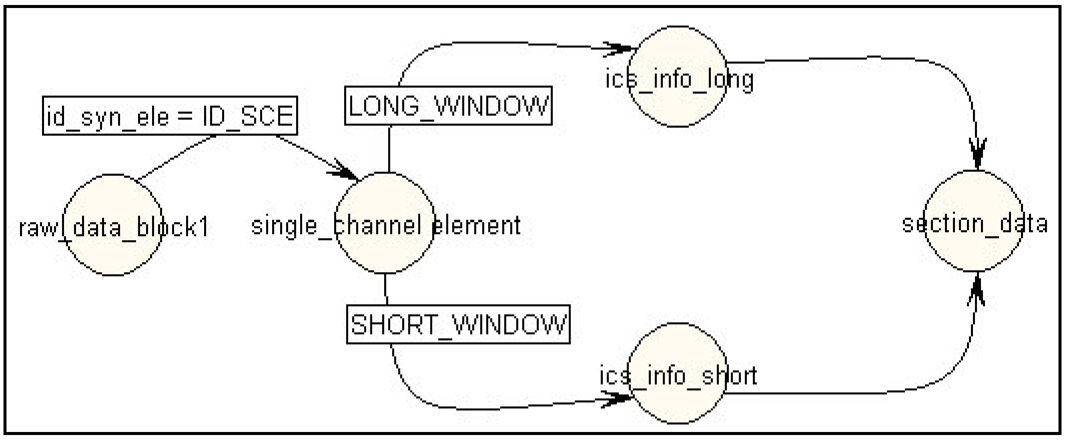


Figure 6-3: Ics_info() state diagram

To complete the decoding of ics_info(), the ‘help’ variables that detail the scalefactor bands and the scalefactor grouping (discussed in section 4.3.1) must be calculated, and the process for calculating these variables is run concurrently with the continuing decoding of the bitstream. At this point however, difficulties in describing the process in VHDL arise due to the presence of a ‘for loop’. Figure 6-4 shows the code for one of the ‘for loops’ [6, p28].

```
for( i = 0; i < max_sfb + 1; i++) {
    sect_sfb_offset[0][i] = swb_offset_long_window[i];
    swb_offset[i] = swb_offset_long_window[i];
}
```

Figure 6-4: Example 'for loop'

The variable swb_offset_long_window[] is implemented in VHDL as a lookup table in the file scalefactor_table.vhd (Appendix C.5). Thus whenever the variable ‘i’ is presented to the table’s input, the value appears on the output. This implementation presents no problem – it is the controller implementation that causes difficulty, and the attempted controller description is included as forloop.vhd (Appendix C.6).

VHDL has two solutions for writing ‘for loops’ – the concurrent statement GENERATE and the sequential statement FOR LOOP. The GENERATE statement is used mainly for component instantiation and is not suitable for executing the ‘for loop’ in case. The FOR LOOP statement is sequential, meaning that it must be run from within a PROCESS statement. The syntactical expression for writing a FOR LOOP [40] is shown in Figure 6-5.

```
[loop_label:]
FOR variable_name IN range LOOP
   statement ;
  {statement ;}
END LOOP [loop_label] ;
```

Figure 6-5: FOR LOOP syntax

The *variable_name* is implicit to the FOR LOOP – it is the value of the range that is important. In the example 'for loop' (Figure 6-4), the range is from 0 to max_sfb + 1. Therefore, the range in the VHDL code would be '0 TO max_sfb + 1'. Assuming that max_sfb is defined as a VARIABLE and not as STD_LOGIC, the syntax is correct and the 'for loop' will loop for the correct number of iterations.

The *statements* within the 'for loop' that store the value from the scalefactor_table in the local arrays sect_sfb_offset[0][] and swb_offset is the source of the problems with the implementation. This only becomes explicitly obvious with an understanding of how VHDL executes a PROCESS statement.

A PROCESS statement runs a series of sequential statements, and the values assigned to variables and signals within a process only take effect when the process is completed. Simply, this implies that the *variable* used in the FOR LOOP statement cannot be used to 'load' a value from the scalefactor table. Only when the process is completed will this value be assigned, and since the FOR LOOP requires a new value for each iteration, the 'for loop' does not evaluate in the desired manner.

At this point it is obvious that a solution for evaluating 'for loops' is required. Apart from the case in question, section_data(), scale_factor_data() and spectral_data() all have complicated looping requirements (including 'while loops' and nested 'for loops'). As well, scale_factor_data() and spectral_data() require more rigorous access to tables (similar to the scalefactor table needed in this case).

With these considerations, work on the decoding of a <SCE> was halted. The state machine required to describe the decoding process of a <SCE> would be very complicated without a well-designed solution to handle loops. The time available did not yield a solution and a VHDL description was not attempted. Also, work on coding the Huffman tables was halted because without a looping solution, the time required to implement them using VHDL was better used elsewhere.

Section 8.1.1 makes some recommendations about a workable controller solution.

6.1.4 <FIL> and <TERM>

The <FIL> and <TERM> elements of the L1_fs_mod bitstream were not described in VHDL primarily due to the incomplete description for a <SCE>. The presence of 'for loops' in the decoding procedure of a <FIL> element was a secondary deterrent. The decoding procedure for <TERM> is simply to look for the next syntactic element.

6.2 Noiseless Coding

In section 4.6 (particularly Table 4-3), the relationship between noiseless coding and spectral_data() decoding was explained. These two processes are entirely dependent upon one another; the noiseless coding tool cannot be coded separately to the spectral_data() decoder. Thus the decision to abandon the VHDL description of the spectral_data() decoder (section 6.1.3) forces the abandonment of the VHDL description of the noiseless coding tool.

6.3 Quantisation and Scalefactors

There are two reasons that the quantisation tool and the scalefactor tool were considered together. The first is that they both apply non-linear transformations to all the Huffman decoded spectral values (see section 4.7 and 4.8). The second is that both tools can be applied within the same control loop.

6.3.1 Implementation of non-linear functions

The quantisation and scalefactor tools are very easy to implement in software because the inputs and the outputs for the calculations can be specified as floating point numbers. However, VHDL does not incorporate a standard floating-point arithmetic library, leaving two options for the implementation of non-linear functions. Firstly, a FPU (floating-point unit) core could be imported and used as the arithmetic core, or alternatively, a fixed-point approximation of the functions could be used.

6.3.1.1 FPU Core

An implementation based on an FPU core would require a floating-point adder and a floating-point multiplier. Both the quantisation and scalefactor tools are based on functions that involve an integer raised to the power of a fraction (e.g. $x^{4/3}$). For a simple FPU core to execute these non-linear functions, polynomial regression must be used to change the exponential functions into polynomial functions that can be evaluated with standard multiplication and addition.

Figure 6-6 shows a graph of the output values (*x_invquant*) of the quantisation tool plotted against a range of input values (*x_quant* between 0 and 32). This function can be approximated using second order polynomial regression, and the approximating equation is:

$$x\_inquant = 0.04 \cdot x\_quant^2 + 1.98 \cdot x\_quant - 1.77$$

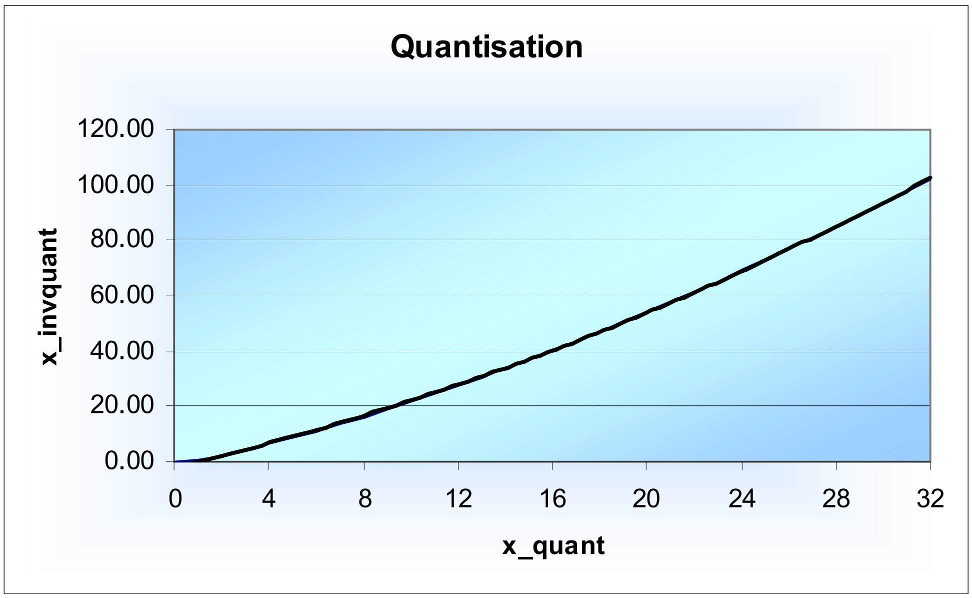


Figure 6-6: Approximation of the quantisation function using polynomial regression

Figure 6-7 shows a graph of the output values (*gain*) of the scalefactor tool plotted against a range of input values (*sf* between 0 and 252). Also shown is an approximation of the function using fifth order polynomial regression. It stands to reason that while the quantisation tool can be approximated with polynomial regression, the scalefactor tool cannot. Therefore, it would have been impractical to implement the scalefactor tool using an FPU core.

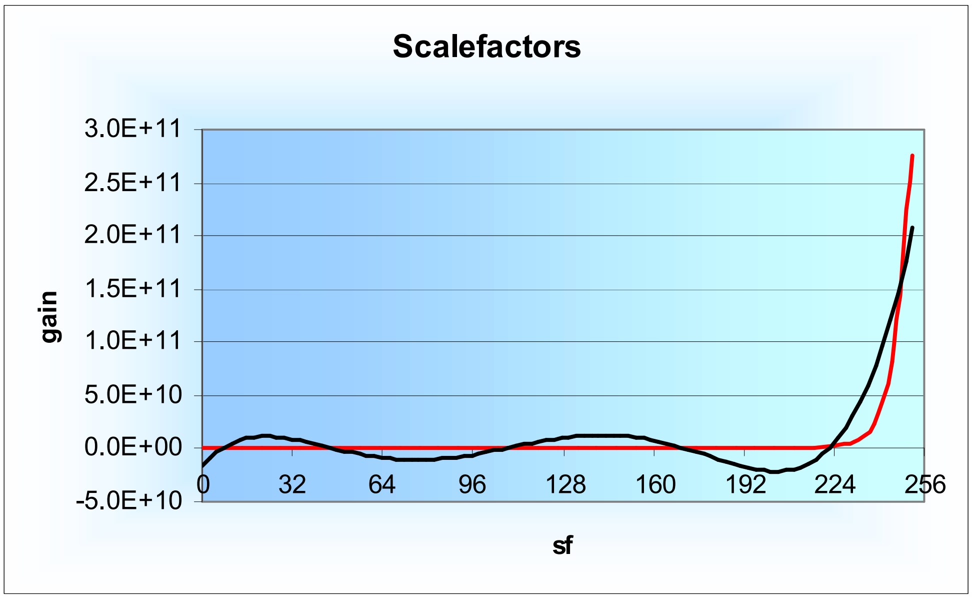


Figure 6-7: Approximation of the gain function using polynomial regression

As well as accuracy, it was also important to consider the resources required for importing an FPU core and a paper on FPU optimisation by Irvin Ortiz Flores gave relevant insight into an FPU core's resource requirements. Table 6-1 summarises the FPGA resources (slices and lookup tables) used by his optimised FP adder and multiplier. These units conform to the IEEE standard for single precision floating-point numbers.

Operation	Exponent Size	Mantissa Size	Slices Used	LUTs Used
FP Add	8 bits	23 bits	424	2402
FP Multiply	8 bits	23 bits	398	1062

Table 6-1: Resource requirements of floating-point units

The XCV300 has 3072 slices and 6144 LUTs. Using the two FP units would require that 26.8% of the FPGA's slices and 56.4% of the FPGA's LUTs be dedicated to the FPU arithmetic core. Though these results offer only an approximation, they are sufficient to conclude that it is inappropriate to dedicate this much resource to FP arithmetic in light of the complexity (and resource required) of the AAC decoder itself.

6.3.1.2 Fixed-point Arithmetic

The alternative to an FPU core is a fixed-point approximation, and this was the option chosen for the quantisation and scalefactor tools. A fixed-point approximation uses only integer values and can therefore be implemented using either the standard VHDL arithmetic operators contained in the STD_LOGIC_ARITH package or by storing the integer values in lookup tables.

The non-linear functions used in the two tools are approximated using lookup tables. The output values are stored and indexed according to their corresponding input values. To overcome the inaccuracy introduced by fixed-point arithmetic, the stored values were all multiplied by eight before rounding occurred, and the scalefactor tool was nested within the quantisation tool (section 6.3.2). Were the scalefactor tool not nested within the quantisation tool, it would be unnecessary to multiply the values in the quantisation lookup table by eight.

Table 6-2 shows the steps involved in calculating the values stored in the lookup tables.

Quantisation		
Step	**Action**	**Result**
1.	Apply Quantisation function (e.g. value = 9)	18.72
2.	Multiply result by 8	149.76
3.	Round to nearest integer and store in lookup table	150
Scalefactors		
Step	**Action**	**Result**
1.	Apply Scalefactor gain function (e.g. value = 118)	22.63
2.	Multiply result by 8	181.04
3.	Round to nearest integer and store in lookup table	181

Table 6-2: Application of adjustment factor for lookup table values

6.3.2 Implementation of control loop

The control loops for the quantisation and scalefactor tools are complicated, consisting of four nested ‘for loops’. The combination of these two loops results in the pseudo-code shown in Figure 6-8. An examination of the structure of the code indicated that considering the two tools together may yield a more optimised solution, and this was the case.

```
for (g = 0; g < num_window_groups; g++) {
   for (sfb = 0; sfb < max_sfb; sfb++) {
      width = (swb_offset[g][sfb + 1] - swb_offset[g][sfb]);
      for (win = 0; win < window_group_len[g]; win++) {
         gain = 2^(sf[g][sfb] * 0.25f - 25);
         for (bin = 0; bin < width; bin++) {
            x_invquant[g][win][sfb][bin] =
                   sign(x_quant[g][win][sfb][bin]) *
                   abs(x_quant[g][win][sfb][bin])^(4/3);
            x_rescal[g][win][sfb][bin] =
                   x_invquant[g][sfb][win][bin] * gain;
         }
      }
   }
}
```

Figure 6-8: Quantisation and scalefactor control loop

The effectiveness and accuracy of the implementation of the integer lookup tables and the hybrid control loop is discussed in section 7.something; an explanation of the VHDL coding of the implementation – both datapath and controller – is included here.

6.3.3 Datapath

The datapath design for the quantisation and scalefactor tools is shown in Figure 6-9. The datapath does not require an external clock as concurrent assignment statements are used for all the events in the VHDL description. Similarly, no reset function is required as all the logic is combinational. The VHDL source files – quantisationTop.vhd, invquant.vhd, scaleapp.vhd and quantadjust.vhd – are listed as Appendices C.7, C.8, C.9 and C.10.

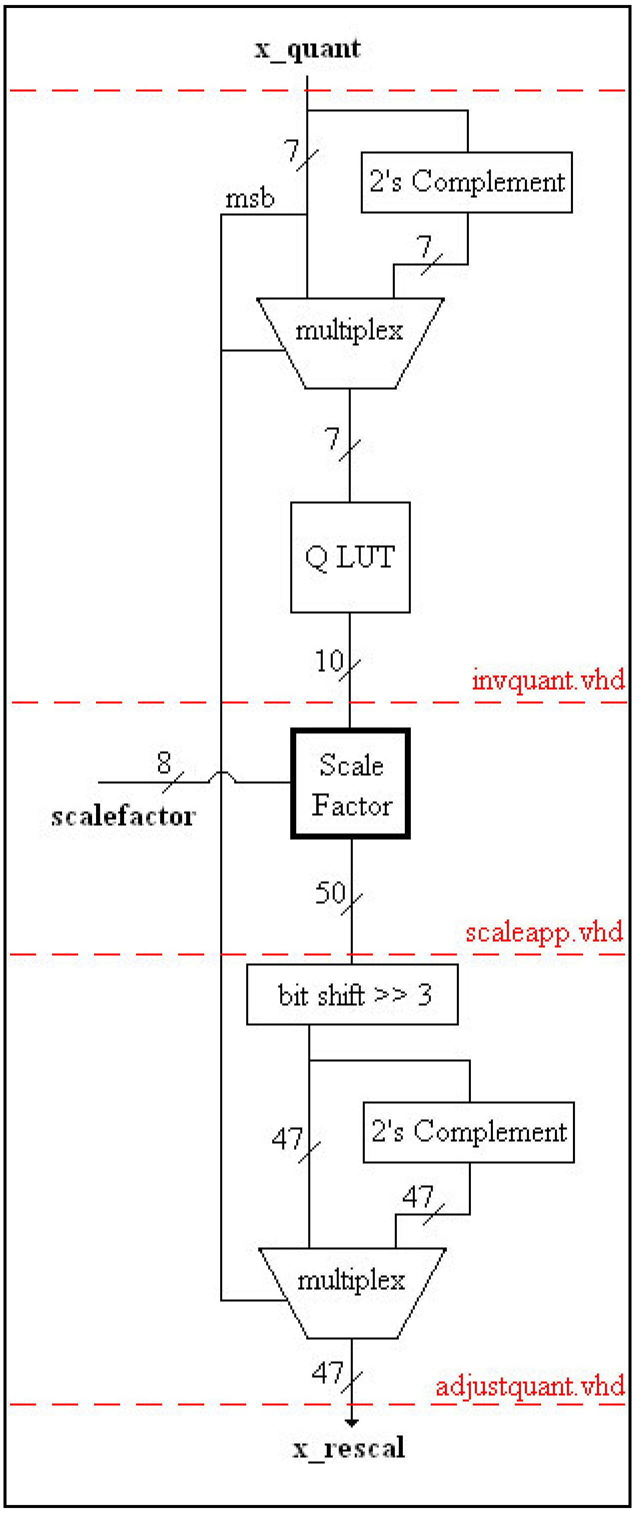


Figure 6-9: Quantisation and scalefactor datapath

The data input for the datapath is the spectral value *x_quant*. The maximum absolute value for *x_quant* is 8191, but for decoding L1_fs_mod, the maximum absolute value is

32. The *x_quant* input must therefore be of a bit magnitude high enough to represent a two's complement range including +32. Seven bits are required since six bits only provide a range from –32 to +31.

The file invquant.vhd describes the use of the quantisation lookup table. To halve to size of the Q LUT, only the positive values (0 to 32) were used as indexes. Thus the first operation in the datapath prepares the two's complement of *x_quant* and uses the MSB of *x_quant* to determine whether the original value or the two's complement value is the absolute value of *x_quant*. The absolute value is then multiplexed into the Q LUT, which contains an *x_invquant* value for each *x_quant* 0 to 32. *[Note: The value stored is eight times the floating-point value for* x_invquant *and rounded to the nearest integer.]*

The scalefactor tool is nested inside the quantisation tool, and the tool's datapath (described by scaleapp.vhd) is shown in Figure 6-10. A gain factor is retrieved from the SF LUT and multiplies with *x_invquant* to give *x_rescal*. As the tool is nested, this multiplication is done before the sign is reinstated to *x_rescal* and before the value of *x_rescal* is adjusted by the divisor of eight.

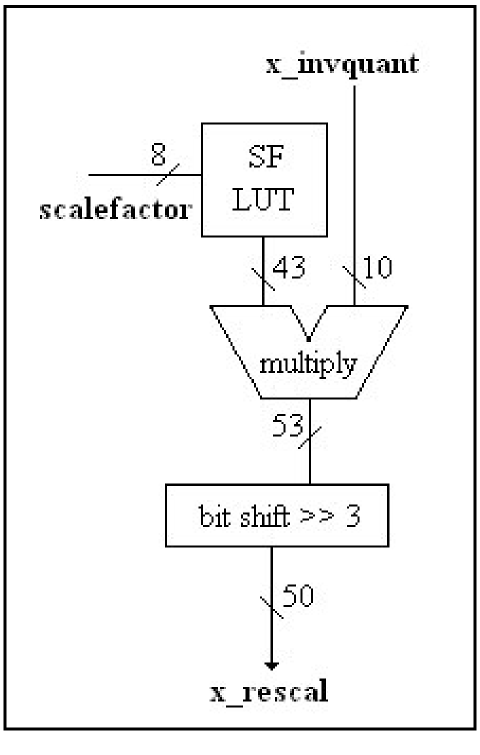


Figure 6-10: Scalefactor datapath

A design assumption was made in regard to the width of the scalefactor input. For L1_fs_mod, the largest scalefactor value is 125, requiring 7 bits. However, the first active scalefactor is the 8-bit value **global_gain**, and so an 8-bit input was used. It is assumed that no scalefactor in any encoded bitstream will exceed 255.

The final function in the datapath is adjusting the 53-bit value for *x_rescal* by dividing it twice by eight (once in scaleapp.vhd and once in adjustquant.vhd) and reinstating the sign. The result is a 47-bit two's complement value *x_rescal*. An array of these values is ready for transformation under the IMDCT in the Filter Bank tool.

6.3.4 Controller

As introduced in section 6.3.2, the control of the quantisation and scalefactor tools is done by a hybrid block of four nested 'for loops' (see also sections 4.7 and 4.8). Work done on the implementation of the bitstream demultiplexer to decode a <SCE> (section 6.1.3) was halted due to an inadequate method for encoding 'for loops'. As the hybrid controller for the quantisation and scalefactor tools has four nested 'for loops', implementation in VHDL was not begun. However, section 8.something gives consideration to how the controller for these tools could be implemented.

6.4 Filter Bank

Some investigation of the Filter Bank tool was carried out but the tool was not implemented in VHDL. Instead, considerations for an implementation of the tool are presented in section 8.1.3.

6.5 Summary

This chapter has addressed the work done on the coding of the AAC decoder in VHDL for implementation on an FPGA. The bitstream demultiplexer was partially decoded, but a full VHDL implementation was not attempted due to the lack of a well-designed solution for handling loops (section 6.1). The noiseless coding tool was not described because the bitstream demultiplexer was incomplete (section 6.2). A fixed-point implementation of the quantisation and scalefactor datapath was realised, but the controller was not implemented (section 6.3). Finally, no work was done on coding the filter bank in VHDL.

7 PROJECT EVALUATION

This chapter evaluates the performance and results of the work undertaken toward a VHDL implementation of an AAC decoder as well as the personal performance of the engineer – the processes and methodologies used. The evaluation compares the final results of the thesis project with the project objectives.

7.1 Evaluation of Product Performance

The objective of the project was to investigate, and implement, an MPEG-2 AAC decoder for real-time decoding of an AAC LC bitstream. The conformance bitstream 'L1_fs' was selected; the tools required for a full decoding the bitstream are the bitstream demultiplexer tool, the noiseless coding tool, the quantisation tool, the scalefactor tool and the filter bank tool.

Implementations of the noiseless coding tool and the filter bank tool were not attempted. Therefore, a full decoding of the L1_fs bitstream was impossible. However, significant work was carried out with the bitstream demultiplexer tool (section 6.1) and the quantisation and scalefactor tools (section 6.3). An evaluation of the performance of the three tools implemented follows, and conclusions are drawn.

7.1.1 Bitstream Demultiplexer

The bitstream demultiplexer was evaluated in two stages. Firstly, the correctness of the VHDL implementation was tested using Active-HDL's waveform editor and a testbench. Secondly, the design was synthesised and implemented with Xilinx Foundation.

7.1.1.1 Active-HDL Waveform and Testbench

As detailed in section 6.1, the bitstream demultiplexer was only partially implemented in VHDL. The implementation decodes the header followed by the <DSE> and up to the end of ics_info() in the first <SCE>. The integrity of this implementation was observed using a VHDL testbench and a memory block (written in VHDL) dedicated to streaming the L1_fs_mod bitstream into the decoder. The VHDL descriptions testbench.vhd and bitstream_source.vhd are included as Appendices C.11 and C.12.

The waveform in Figure 7-1 shows that the header is the first element decoded by the VHDL model. The value of the counter *count_data* is set to 25Fh to count down from 607 to zero. The input *data_available(0)* is the most recent bit that was shifted into the decoder, and the header data is not stored, but is simply allowed to shift in and out of the shift register.

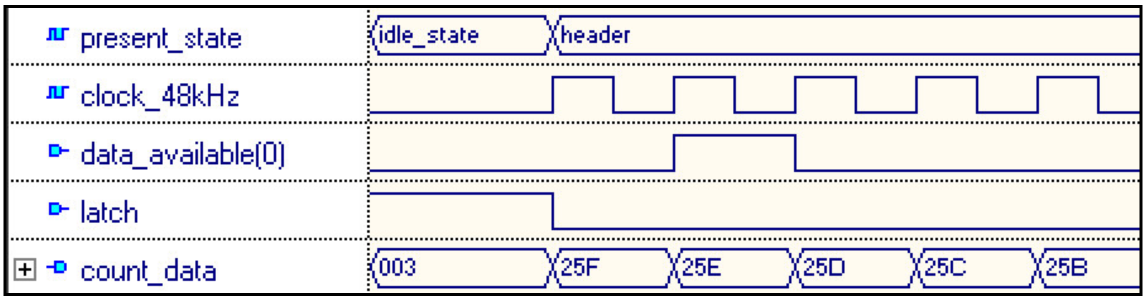


Figure 7-1: Waveform output when decoding the header

Following the header is a <DSE>. The first objective when decoding the <DSE> is to determine the number of data stream bytes in the element. *Dse_count* is read as 001h, indicating that two bytes follow. As *dse_count* is less than 255, the data bytes are read but not stored during *data_stream_element3*. The data correlates to the values determined in section 4.2, and the waveform for the decoding of the <DSE> is shown in Figure 7-2.

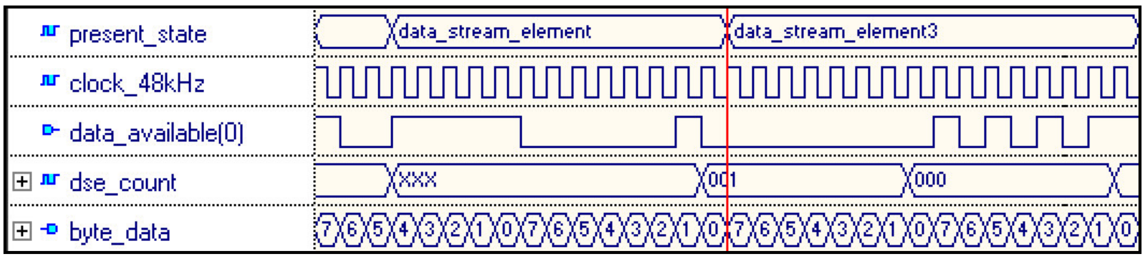


Figure 7-2: Waveform output when decoding a <DSE>

Also of note is the correct implementation of the byte_alignment() function. The red vertical line in Figure 7-2 indicates that the *byte_data* value to the left of the line was added to the counter value (not shown) on the transition to the *data_stream_element3* state. There are eight clock cycles while *dse_count* is equal to one, confirming that zero was added for the byte_alignment().

The next syntactical element in L1_fs_mod is a <SCE>. Figure 7-3 shows the waveform for decoding this element up to the end of ics_info(). The *global_gain* is 64h, the *window_shape* is 0h, the *window_sequence* is 01h and the *max_sfb* is 29h. The *scale_factor_grouping* remains undefined because the window sequence is not EIGHT_SHORT_WINDOWS. There five values are the same as the values obtained in section 4.3. The state transition within ics_info() was also correctly evaluated.

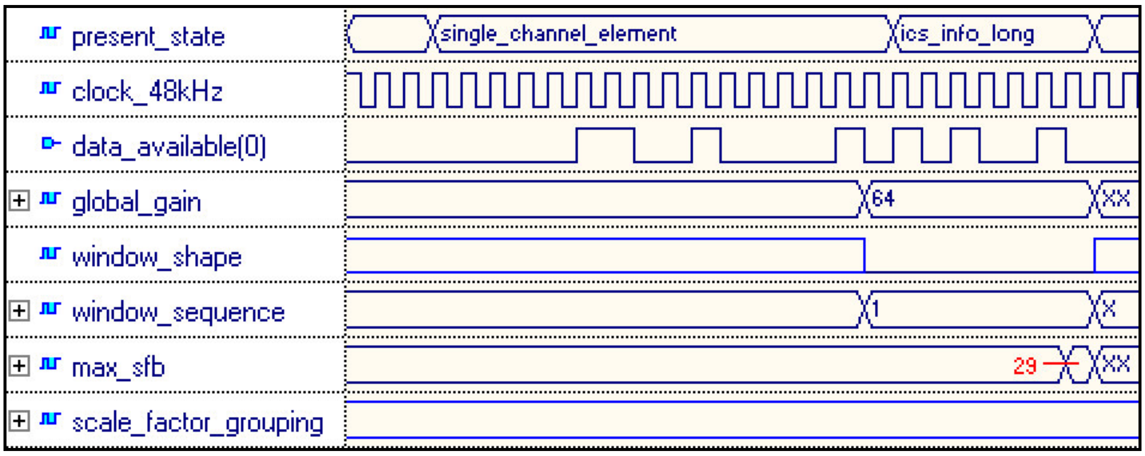


Figure 7-3: Waveform when decoding a <SCE>

7.1.1.2 Xilinx Foundation Synthesis and Implementation

The four VHDL files describing the bitstream demultiplexer (bitstream_top.vhd, bitstream.vhd, counter.vhd and shift_register.vhd) were sythesised in Xilinx Foundation for the XCV300PQ240 FPGA with a speed grade of 6. The synthesis generated seven warnings that a "variable is being read (inside a state machine) but is not in the process sensitivity list of the block which begin there (HDL-179)". These warnings were ignored, as the functionality of the state machine requires the variables to be omitted from the sensitivity list, and are not detrimental to the implementation.

A detrimental warning did occur however during mapping (FPGA-PADMAP-2). The synthesiser optimised the design by removing the input port *bits_in*, indicating that the port was not attached to a net and that it was redundant. Clearly, *bits_in* is not a redundant port – it is the input port for the AAC bitstream – but much investigation and debugging did not remedy the warning.

The implementation of the bitstream demultiplexer was successful, but the implementation did not include the input port *bits_in*. This ruled out any opportunity to develop a testing procedure for running the tool on the FPGA. However, the implementation results for the bitstream demultiplexer are available and included below as Figure 7-4.

```
Xilinx Mapping Report File for Design 'bitstream_top'
Copyright (c) 1995-2000 Xilinx, Inc.  All rights reserved.

Design Information
------------------
Command Line   : map -p xcv300-6-pq240 -o map.ncd decoder.ngd decoder.pcf
Target Device  : xv300
Target Package : pq240
Target Speed   : -6
Mapper Version : virtex -- D.27
Mapped Date    : Fri Oct 11 13:27:51 2002

Design Summary
--------------
   Number of errors:      0
   Number of warnings:    1
   Number of Slices:                 7 out of  3,072    1%
   Number of Slices containing
      unrelated logic:               0 out of      7    0%
   Number of Slice Flip Flops:      12 out of  6,144    1%
   Number of 4 input LUTs:           0 out of  6,144    0%
   Number of bonded IOBs:            2 out of    166    1%
   Number of GCLKs:                  1 out of      4   25%
   Number of GCLKIOBs:               1 out of      4   25%
Total equivalent gate count for design:  159
Additional JTAG gate count for IOBs:  144
```

Figure 7-4: Xilinx implementation output statistics for bitstream demultiplexer

In addition to the above results, the post-layout timing analysis showed a maximum clocking frequency of 215.8MHz. The implementation could be run on the XCV300 at the maximum clocking frequency while making very small demands on the FPGA's resources.

7.1.1.3 Summary

The partial implementation of the bitstream demultiplexer behaves correctly in a VHDL testbench. The first 688 bits of L1_fs_mod can be decoded in accordance with the documented standard ISO/IEC 13818-7. The FPGA implementation of the VHDL code does not function and was not tested due to an FPGA-PADMAP-2 warning.

7.1.2 Quantisation and Scalefactors

The quantisation and scalefactor tools were evaluated in four stages. Firstly, the accuracy of the fixed-point implementation was compared to alternate approximations and to the original floating-point calculations. Secondly, the correctness of the VHDL implementation was tested using Active-HDL's waveform editor. Thirdly, the design was synthesised and implemented with Xilinx Foundation. Finally, the design was run and verified on the FPGA.

7.1.2.1 Verification of Fixed-Point Accuracy

Section 6.3 explained that the quantisation and scalefactor tools used lookup tables to implement fixed-point arithmetic. Moreover, the values stored in the tables were multiplied by eight before rounding and the scalefactor tool was nested within the quantisation tool in an effort to increase the accuracy of the fixed-point approximations.

The accuracy of the implemented datapath was compared with some alternatives using Microsoft Excel. A spreadsheet was created that calculated the results of the quantisation and scalefactor tools, and three analysis tests were carried out with these results.

The first test compared the accuracy of the implemented fixed-point lookup tables with the original floating-point functions. The *scalefactor* value of 137 was used, and the results were calculated over the range 0 to 32 for *x_quant*. Figure 7-5 shows the percentage difference resulting from the implemented quantisation-scalefactor datapath replacing the floating-point calculations. The accuracy is excellent, within ±1%.

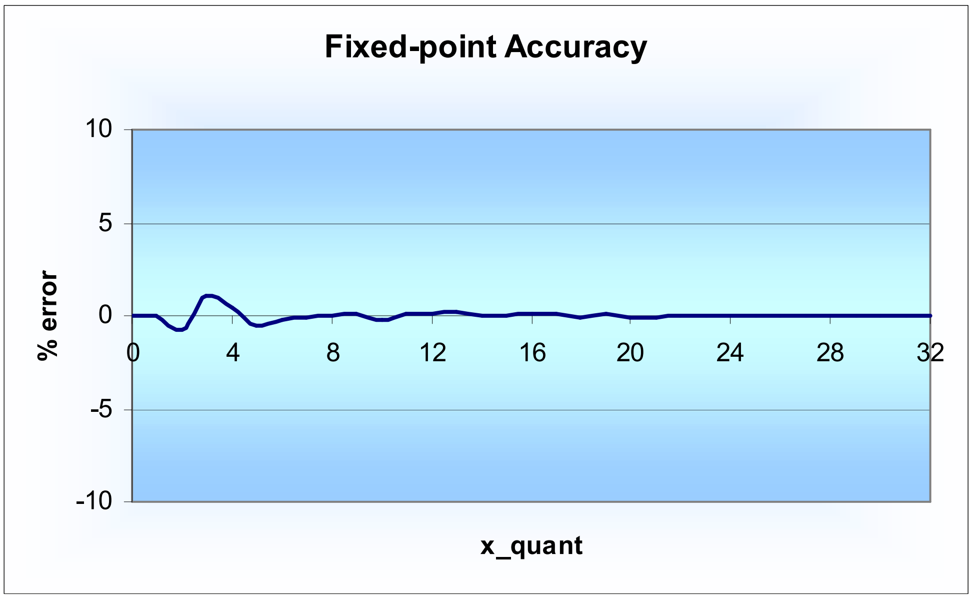


Figure 7-5: Accuracy of fixed-point implementation compared to floating-point

The second test was carried out to determine whether a multiplication factor of eight was ideal for the stored lookup table values of the scalefactor tool. Figure 7-6 shows the percentage error of using one, eight and sixteen as multiplying factors. It is conducted over the range of 85 to 125; outside of these values, the error is negligible.

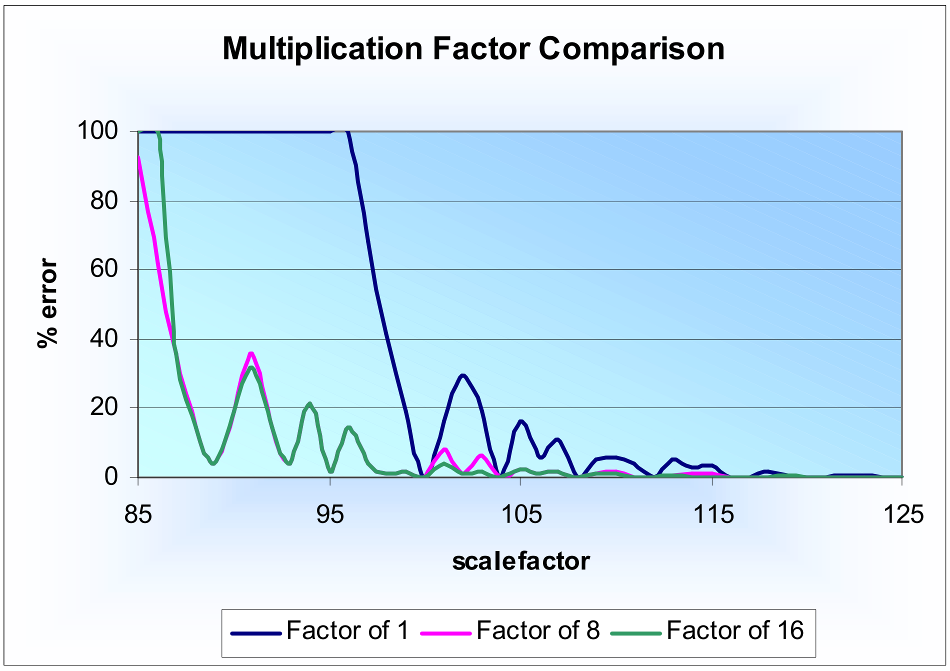


Figure 7-6: Accuracy achieved using different multiplication factors

The accuracy improvement of using a multiplication factor is clear. However, the difference between using sixteen in comparison with eight is small. Using sixteen instead of eight would increase the size of the lookup table and increase the complexity of the multiplier in the scalefactor datapath (Figure 6-10). Both of these tradeoffs are more significant than the resulting increase in accuracy, and a multiplication factor of eight is the ideal value.

The third test was carried out to determine the benefit of nesting the scalefactor tool within the quantisation tool. Figure 7-7 shows the percentage error of using a nested datapath versus a sequential datapath for each value of x_quant. It is clear that the performance increase due to nesting is significant, and verifies that it is the ideal solution.

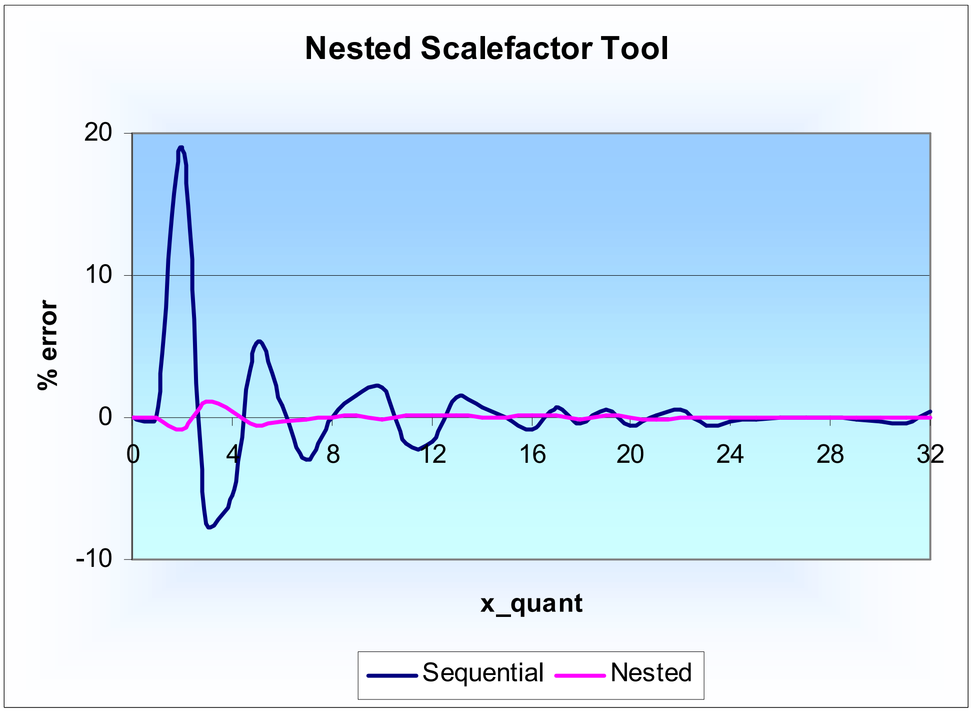


Figure 7-7: Accuracy improvement with a nested scalefactor tool

7.1.2.2 Active-HDL Waveform

The correctness of the VHDL implementation was verified using the waveform editor of Active-HDL. Table 7-1 summarises the test vectors used for the verification and Figure 7-8 shows the waveform output of the nested datapath. The results from the datapath are as expected using the given vectors.

INPUT				**OUTPUT**	
x_quant		**scalefactor**		**x_rescal**	
Decimal	Hex	Decimal	Hex	Decimal	Hex
0	00	0	00	0	000000000000
32	20	137	89	61876	00000000F1B4
-32	60	137	89	-61876	7FFFFFFF0E4C
32	20	118	76	2299	0000000008FB
-4	7C	118	76	-144	7FFFFFFFFF70

Table 7-1: Test vectors used for verification of datapath

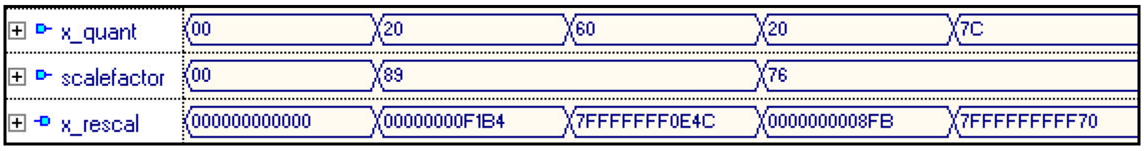


Figure 7-8: Waveform output of quantisation and scalefactor tools

7.1.2.3 Xilinx Foundation Synthesis and Implementation

The VHDL files quantisation_top.vhd, invquant.vhd, scaleapp.vhd and adjustquant.vhd were synthesised and implemented using Xilinx Foundation. No errors or warnings were encountered in either process, and a transcript of the implementation results is included as Figure 7-9.

```
Xilinx Mapping Report File for Design 'quantisation_top'
Copyright (c) 1995-2000 Xilinx, Inc.  All rights reserved.

Design Information
------------------
Command Line   : map -p xcv300-6-pq240 -o map.ncd q.ngd q.pcf
Target Device  : xv300
Target Package : pq240
Target Speed   : -6
Mapper Version : virtex -- D.27
Mapped Date    : Tue Oct 08 15:17:23 2002

Design Summary
--------------
   Number of errors:      0
   Number of warnings:    1
   Number of Slices:               479 out of  3,072   15%
   Number of Slices containing
      unrelated logic:               0 out of    479    0%
   Total Number 4 input LUTs:      937 out of  6,144   15%
      Number used as LUTs:                       936
      Number used as a route-thru:                 1
   Number of bonded IOBs:           60 out of    166   36%
Total equivalent gate count for design:  9,081
Additional JTAG gate count for IOBs:  2,880
```

Figure 7-9: Xilinx implementation output statistics for quantisation / scalefactor tools

The post-layout timing analysis reported that the maximum delay in the datapath was 34.89ns, indicating that the quantisation and scalefactor tools could produce 28.7 million results per second without pipelining the implementation. The timing performance and resource requirements of the tools are well within the capability of the FPGA.

7.1.2.4 Verification with XSV300 Development Board

Finally, the quantisation and scalefactor tools were verified on the XSV300 development board. Some modifications were made to the VHDL code for the purposes of testing and for allowing the inputs and outputs to be easily mapped on the board.

The datapath inputs were mapped to push buttons and switches. The *x_quant* input was reduced to a 4-bit input and mapped to the active-low pushbuttons. The upper three bits were tied low, reducing the range of testable x_quant values to 0 to +15. The *scalefactor* input was mapped to the eight-position DIP switch (ON = logic low, OFF = logic high) and retained its full range.

The datapath output, *x_rescal*, was mapped to the left and right expansion headers. These expansion headers are used to access the SRAM chips on the XSV Board, but can be used for general I/O if the SRAM chips are disabled. Therefore, two extra output signals were incorporated into quantisation_top.vhd that tied the SRAM chip enable pins high.

A UCF constraints file (see Appendix D) was written for the implementation that maps the synthesised I/O pads to the FPGA's pins. Xilinx Foundation was used to re-implement the synthesis with the constraints file and GSXLoad was used to download the bitstream 'q.bit' to the XSV Board. The design was tested and verified using a multimeter to check the response of the output pins to given input values.

7.1.2.5 Summary

The implementation of the quantisation and scalefactor tools is completed. The accuracy of the tools is ±1% compared to a floating-point implementation, and considering the tools together optimises the datapath. A full VHDL description behaves correctly under testing in Active-HDL, and synthesises and implements without errors or warnings in Xilinx Foundation. Finally, the design displays accuracy, robustness and soundness when executed on the XSV Board.

7.2 Evaluation of Personal Performance

Each stage of the project required the employment of technical skills and time management skills. This section evaluates the processes that resulted in the final product and design. For suggestions on how to improve the employed methodologies for future projects, see chapter eight.

7.2.1 Technical Skills

The major components of the technical skill set required for the project were research skills, digital design skills, VHDL coding skills, reporting skills and presentation skills. A critical evaluation of this skill set follows.

The process of researching the topic of audio coding algorithms, collaborating data in a meaningful manner and making informed decisions based on the collaborated data was done well. Many sources – technical books, journal articles, conference papers, Internet publications and technical documents – were accessed to provide the project with foundation and scope. The main deficiency in research methodology was the late acquisition of the ISO/IEC 13818-7 standard document. Originally, it was decided that this document would not be essential reading, but as the project progressed, the necessity of the document became glaringly obvious.

The process of digital design is a skill that was acquired in subjects studied previously, notably COMP3100 and COMP4100. Having gained a thorough understanding of the AAC decoding algorithm, state machines, counters, registers, adders and other logic units were used to implement a data path and control unit for sections of the decoder. The complexity of the AAC decoder required more precise design decisions – decisions that, in the end, were unable to be made to an optimal level (e.g. 'for loop' design solution).

Experience with the VHDL language in previous studies was sufficient to code the included parts of the decoder, but not sufficient to code the solution optimally. Notably, state machine coding and concurrent process coding were done with a functional goal in mind, not a performance goal.

Experience in reporting on large projects was minimal, and the reporting process was neglected until the end of the project. A workbook was kept throughout, making information integration a little easier, but at all points during the project when reporting was taking place, technical progress stopped. Similarly, though presentation skills had been acquired in previous studies, the presentations were always at the conclusion of the project. Preparation for presentations was at the expense of progress on the technical aspects of the project.

7.2.2 Time Management Skills

The thesis project, due to its duration and complexity, requires good time management skills in order for the project to be a success. Time management can be divided into two main categories, internal and external.

Internal time management refers to how the allocated time for the project was spent. To organise this time, a project plan was included with the project progress report and submitted on 19th April 2002. Since that submission, the project plan has not been referred to, indicating clearly that it was non-functional. The project plan was exceedingly optimistic, and even ignorant of the requirements of the project. It was written hastily and without a thorough consideration of the project's demands.

A critical evaluation of internal time management for the project concludes the following. Too much time was wasted before research commenced, and the research required much more time than was first thought. A seven-week delay on the delivery of the ISO/IEC 13818-7 standard (requested May 22, received July 19) pushed the whole project schedule back a number of weeks, resulting in implementation commencing on 9th September – six weeks later than an ideal date in early August. Contingency decisions were made to compensate for this unforseen delay, with the decoder implementation suffering as a result.

External time management refers to how the allocated time for the project was scheduled among the other demands on a 4th year engineering student. There is always the possibility of spending more time on the project, but from the outset it was decided that it was important to maintain a balanced academic, extra-curricular, social and spiritual life. Accordingly, the relative quality of the product is lower, but the relative quality of all other aspects of life is higher.

A critical evaluation of external time management for the project concludes the following. During first semester, too much time was given to other university subjects (particularly COMP4102) at the expense of time spent on the thesis project. This was detrimental to progress, and increased the second semester workload (alongside the above mentioned delay in document delivery). During second semester, the intentional decision to maintain a balanced lifestyle did not provide time to make up for the time lost in first semester.

7.3 Summary

This chapter evaluated the partial implementation of the bitstream demultiplexer (section 7.1.1) and the full implementation of the quantisation and scalefactor tools (section 7.1.2). The results of the evaluation show that the work done during the project is a solid foundation for a full implementation of an AAC decoder capable of decoding the conformance bitstream L1_fs and eventually, any AAC LC bitstream. Also, the processes and methodologies used in achieving the final results were evaluated (section 7.2).

8 FUTURE DEVELOPMENTS

This chapter provides suggestions for future work on the implementation of an AAC decoder. Also, suggestions for improvements regarding the approach to a similar project are made.

8.1 Product Developments

Four main areas of further work and development have been identified at the conclusion of this project. Firstly, the implementation of an adequate controller for the bitstream demultiplexer and the noiseless coding tool must be developed, as well as the quantisation and scalefactor tools. Secondly, work must be done on a complete approach to data storage during decoding. Thirdly, the filter bank tool must be developed. Finally, the FPGA's interaction with the PC – both for bitstream decoding and configuration programming – requires work.

8.1.1 Controller Implementation

The main conclusion drawn from the partial VHDL implementation of the bitstream demultiplexer (section 6.1.3) was that a method for implementing complicated 'for loops' and table accesses must be engineered. Similarly, the VHDL implementation of a controller for the quantisation and scalefactor tools (section 6.3.4) was abandoned due to the four nested 'for loops' required for processing. A recommendation is made for an approach toward a controller implementation that would solve the problem.

The control needed to demultiplex an AAC bitstream is not well suited to an FPGA implementation. As detailed in section 2.4, DSPs excel and FPGAs do not excel when extensive evaluations of conditional statements are required, and this is surely one reason why DSPs have been preferred for all the current commercial implementations of hardware-based AAC decoders. DSPs can be programmed with the C programming language (or equivalent) – a language much more proficient in dealing with conditional statements than VHDL. Similarly, the nested 'for loops' used for the quantisation and scalefactor controller are much better suited to a more traditional hardware architecture and implementation.

A combination DSP/FPGA solution for the decoder controller would be inappropriate if the DSP was used primarily as a decision maker. Perhaps the best solution for controlling the AAC decoder would be to use a small microcontroller. The bitstream demultiplexer could be controlled using code almost identical to the syntax descriptions in ISO/IEC 13818-7, and hence the noiseless coding tool could also be controlled using the microcontroller (see Table 4-3.)

A microcontroller could also handle the transfer of any externally stored data (Huffman tables, spectral values etc – see section 8.1.2) between the storage device and the FPGA. Certainly the controller for the quantisation and scalefactor tools only needs to ensure that the correct values in the interleaved array are modified and updated in a systematic manner.

A feasibility assessment of this recommendation for the incorporation of a microcontroller is beyond the scope of this project, as is any development of this recommendation. It is suggested for future work.

8.1.2 Data Storage

No formal work was done during the project on the issue of data storage. At many stages in the decoder's datapath, information is required to be retained. Whether this is the static storage of the Huffman tables, the storage of the spectral values attained from the noiseless coding tool, the various storage requirements within the filter bank or the storage of control information gathered from the bitstream demultiplexer, appropriate decisions about data handling must be made.

The storage requirements for the Huffman tables will be determined primarily by the manner in which the tables are coded. If simple lookup tables are used, the data will be 'stored' within the logic on the FPGA and consume chip area and resources. If a controller is used to access some external storage device, latency will be introduced.

After Huffman decoding has occurred, each raw data block yields 1024 samples of spectral data, and the trade-off between speed and area must again be considered when choosing the data storage method for these values.

The XSV board has on-board SRAM, but no work was done with the SRAM during this project. However, previous work has been done with the SRAM [41] and could form the basis for the data storage scheme. Alternatively, other implementations for external storage may need to be considered.

In summary, the data storage issues require more detailed attention than they were given in this project. A suitable solution for retaining data from the input to the output must be achieved if a full implementation of an AAC decoder is to be realised.

8.1.3 Filter Bank

An implementation of the filter bank tool was not attempted as part of this project, and only preliminary work was done toward such an implementation. However, the work did uncover challenges that would need to be overcome in order to make an FPGA implementation of the tool possible.

That the filter bank is suitable for implementation on an FPGA is beyond doubt; many multiply-accumulate (MAC) tasks are frequently done using FPGAs. Furthermore, Xilinx has published an application note [42] describing the implementation of an Inverse Discrete Cosine Transform (IDCT) for a Virtex FPGA. The IDCT is part of the MPEG video decoding standard and the implementation of the IDCT is appropriate for use in a real-time MPEG video decoder.

The main challenge is the same challenge that was faced in the design and implementation of the quantisation and scalefactor tools. By nature, the filter bank requires the incorporation of floating-point arithmetic, and as a further challenge, the incorporation of trigonometric functions, division, and square root. The cosine function is required for the IMDCT itself, the sine function is required for the application of the sine window, and division and square root are required for the application of the KBD window. Though the filter bank would be supplied with integer value inputs, a great deal of work and modification to the filter bank algorithms would be required to retain accurate integer outputs after the computational functions had been applied.

A popular method for implementing trigonometric functions on FPGAs is to use CORDIC (Coordinate Rotation Digital Computer) algorithms. Ray Andraka has published a survey of CORDIC algorithms [43] for FPGA-based computers and explains how vector rotation functions are applicable to hardware blocks such as DFTs (Discrete Fourier Transforms) and DCTs. Also, Vladimirova and Teggeler have published a paper [44] on fast, efficient implementations of CORDIC algorithms on FPGA. However, these papers deal with implementations using *n*-bit binary fractions, and insufficient work has been done to conclude whether or not they are suitable for use in the AAC decoder.

As an alternative to using trigonometric functions, Duhamel, Mahieux, and Petit have presented work [45] that equivocates an FFT-based algorithm using complex number arithmetic with the standard IMDCT. This alternate algorithm has been used in a VLSI implementation [22] of an AAC decoder with results indicating that the computational load of the filter bank was reduced by a factor of ten. However, the processing core was a DSP, and comment on the usefulness of using the alternate algorithm in this case cannot be made.

In summary, the filter bank tool is very appropriate for implementation on an FPGA, but due to its complex base algorithms, only preliminary work could be done during this project. An entire thesis could be written on the implementation of the filter bank tool alone, and this section included information that would be helpful for beginning such a project.

8.1.4 PC-FPGA Communication

There are two instances when the PC is required to communicate with the FPGA. In the first instance, the PC must transfer an AAC LC bitstream to the FPGA so that a 48.8kHz bitstream can be processed serially. However, this does not imply that the communication link between the PC and the FPGA must be a serial link. A suitable communication protocol detailing both the PC side of communication and the FPGA side of communication needs to be developed before the functionality of the decoder can be fully tested.

Previous work with the XSV Board has seen the development of a communication link between the PC and the board's SRAM chips via the parallel port [41]. Both the PC software (written in Visual Basic) and the FPGA configuration (in VHDL) are documented and freely available. This work could provide a starting point for the final PC-FPGA communication link.

The last stage of work on this project will be an attempt to adapt the PC-SRAM interface to demonstrate the functionality of the completed quantisation and scalefactor tools. PC software will be written and incorporated into the FAAD2 decoding algorithm, and VHDL files will be added to the implementation to provide capability for the FPGA to communicate with the PC.

In the second instance, the PC programs the FPGA with the configuration bitstream via the parallel port. A final improvement to the design of the decoder would be the incorporation of standalone operation.

Standalone operation is achieved by downloading the configuration bitstream into the on board Flash RAM and programming the CPLD to load the configuration bitstream from the Flash instead of the parallel port. Such a modification would negate the need for the FPGA to be reprogrammed each time it was powered down.

Standalone operation was successfully incorporated using the XSV Board by Jorgen Pedderson and is detailed as part of the thesis project he completed in 2001 [35]. This reference gives sufficiently detailed information for standalone operation to be incorporated into the AAC decoder.

8.2 Process Developments

Overall, as outlined in section 7.2, the technical skills and time management skills employed during the project worked with some success. However, many lessons were learned, and suggestions for improving technical abilities and time management follow.

With regard to technical skills, little can be done other than to anticipate the problems that may arise and take pre-emptive action. As a specific example, while the project was being delayed by the document delivery problems, reading about VHDL programming skills would have resulted in better preparation for the implementation of the decoder. As another example, recognising the need for the ISO/IEC 13818-7 document earlier – i.e., anticipating that it might be necessary – would cut down on the time spent researching for information on implementational approaches. In a future project, greater anticipation would result in better application of the required skill set.

With regard to time management, a more realistic project plan would have helped achieve a more successful product. The lack of small milestones in the plan lessened the plan's motivational capacity and usefulness. As a specific example, a task such as 'Implement bitstream module' could be broken down further into a number of achievable milestones such as 'Understand algorithms involved', 'Complete paper design', 'Write VHDL code', 'Simulate and debug code' and 'Test implementation'. Progress is then measurable and the plan is then useful.

In summary, a future project would require more thorough planning, both in time scheduling and contingency, to be successful. Also, setting time aside to review progress and involving someone else in that process, i.e. the supervisor, would facilitate a more purposeful approach to the project.

8.3 Summary

This chapter has made recommendations for future work and development in four main areas pertaining to an implementation of an AAC decoder (section 8.1). The primary recommendation was that a microcontroller be used to control the now implemented decoding tools. Also, the developments to the processes involved throughout the project were suggested to improve the performance of the engineer in the future (section 8.2).

9 CONCLUSION

The goal of the project was to investigate and implement an audio coding algorithm on an FPGA. For these purposes, the MPEG-2 AAC decoding algorithm was chosen, and the implementation environment centred around the Virtex XCV300 FGPA. The algorithm was analysed and an appropriate block diagram including the essential decoding tools was extrapolated. A modified conformance bitstream – L1_fs_mod – was selected for decoding and this finalised the design constraints for the decoder.

The L1_fs_mod bitstream was decoded manually by hand to provide the thesis with a foundational theoretic understanding of the decoding algorithm and as well as a comprehensive set of data that was used for testing purposes. The listing of the decoding is included with this thesis and will provide an excellent foundation for future work with the algorithm.

With a thorough understanding of the algorithm, an implementation of the decoder using VHDL was undertaken. Two of the five essential decoding tools were fully described and another was partially described. The final product is incapable of decoding an AAC bitstream, but is a first step toward a fully-functional implementation.

An evaluation of the implemented tools returned excellent results. The fixed-point implementation of the quantisation and scalefactor tools is robust, small in area and is accurate to within ±1% of a floating-point implementation. The bitstream demultiplexer also returned encouraging results after implementation.

An analysis of the problems encountered during the thesis project led to four major recommendations for future work and development with the implementation of the MPEG-2 AAC decoding algorithm.

References

[1] "Internet Audio – MP3", *Xilinx Web Page*, http://www.xilinx.com/esp/technologies/consumer/mp3_audio.htm (Last accessed 15 Oct. 02)

[2] "Napster's Day in Court", *C/NET News.com*, http://news.com.com/2009-1023-252407.html (Last accessed 15 Oct. 02)

[3] "Xilinx High Volume Programmable Logic Applications in Internet Audio Players" *Xilinx Web Page*, 17 January 2000, http://www.xilinx.com/publications/whitepapers/wp_pdf/wp103.pdf (Last accessed 15 Oct. 02)

[4] Bauvigne G, 2002. "MPEG-2/MPEG-4 AAC", *MP3 Tech Web Site*, http://www.mp3-tech.org, (Last accessed 19 Apr. 02)

[5] Purnhagen H, 2002. "MPEG Audio FAQ", *The MPEG Audio Web Page*, http://www.tnt.uni-hannover.de/project/mpeg/audio/faq/mpeg2.html (Last accessed 18 Apr. 02)

[6] "*ISO/IEC 13818-7: Information technology: generic coding of moving pictures and associated audio information. Part 7, Advanced audio coding (AAC)*", International Standards Organisation, Geneva, 1997.

[7] "*ISO/IEC 13818-3: Information technology: generic coding of moving pictures and associated audio information. Part 3, Audio*", International Standards Organisation, Geneva, 1998.

[8] Jansson D, 2001. "What is AAC?", *DJ-Media Web Site*, http://www.dj-media.com/doc/what_is_aac.asp (Last accessed 19 Apr. 02)

[9] Chen J, Tai H, 1999. "MPEG-2 AAC Coder on a fixed-point DSP", *IEEE International Conference on Consumer Electronics*, June 22-24, 1999. p. 24-5.

[10] Kirby, D. & Watanabe, K, 1996. "Report on the Formal Subjective Listening Tests of MPEG-2 NBC multichannel audio coding." TNT Institute Web Site, http://www.tnt.uni-hannover.de/project/mpeg/audio/public/w1419.pdf (Last accessed 15 Oct. 02)

[11] 2000. "AAC – Applications", Advanced Audio Coding Web Site, http://www.aac-audio.com/applications/ (Last accessed 2 Oct. 02)

[12] Brosch, O, 2002. “Introduction to FPGA Processors”, *The FPGA Processors Group Homepage*, http://www-li5.ti.uni-mannheim.de/fpga/?group/intro (Last accessed 15 Oct. 02)

[13] “ISO/IEC 11172-3: Information technology: coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s. Part 3, Audio”, International Standards Organisation, Geneva, 1993.

[14] 2001. “Converting MP3 Software to Hardware”, Celoxica Web Site, http://www.celoxica.com/products/technical_papers/case_studies/cs_001.htm (Last accessed 15 Oct. 02)

[15] TMS320VC5401 Data Manual, *Texas Instruments Web Site*, http://www-s.ti.com/sc/ds/tms320vc5401.pdf (Last accessed 15 Oct. 02)

[16] “Dolby Announces Sanyo’s Support for AAC in New Portable Internet Player”, *AAC Audio Web Site*, http://www.aac-audio.com/pdf/aac.pr.0101.AACSanyo.pdf, (Last accessed 15 Oct. 02)

[17] TMS320C6712 Data Manual, Texas Instruments Web Site, http://www-s.ti.com/sc/ds/tms320c6712.pdf (Last accessed 15 Oct. 02)

[18] PT8402 Datasheet, *Princeton Web Site*, http://www.princeton.com.tw/english/pdfile/dsp/8402s.pdf (Last accessed 24 Sep. 02)

[19] CS49400 Datasheet, *Cirrus Web Site*, http://www.cirrus.com/pubs/cs49400-2.pdf?DocumentID=893 (Last accessed 24 Sep. 02)

[20] MAS3509F Datasheet, *Micronas Web Site*, http://www.micronas.com/products/documentation/consumer/mas35x9f/downloads/mas35x9f_1pd.pdf (Last accessed 24 Sep. 02)

[21] “Move Technology Audio Components”, *ARM Web Site*, http://www.arm.com/armtech.nsf/iwpList73/C24352A48B8EE70580256B810056BA30?OpenDocument&style=IP_Solutions (Last accessed 24 Sep. 02)

[22] Lee, K. & Jeong, N. & Bang, K. & Youn, D. “*A VLSI Implementation of MPEG-2 AAC Decoder System*”, http://www.ap-asic.org/1999/proceedings/8-2.pdf (Last accessed 15 Oct. 02)

[23] Dick, C. “FPGAs: The High-End Alternative for DSP Applications.” *Hunt Engineering Web Site*, http://www.hunteng.co.uk/pdfs/tech/DSP1736FPGA.pdf (Last accessed 15 Oct. 02)

[24] "Choosing DSP or FPGA for your application", *Hunt Engineering Web Site*, http://www.hunteng.co.uk/info/fpga_dsp.htm (Last accessed 15 Oct. 02)

[25] "Mayah AAC Recorder", *Mayah Communication Web Site*, http://www.tranzicom.se/mayah-products/aac_recorder.html (Last accessed 21 Sep. 02)

[26] "QuickTime 6: The Digital Media Standard", *Apple Web Site*, http://a720.g.akamai.net/7/720/51/4c19caa9c78f5d/www.apple.com/quicktime/products/pdf/L19086B_QT6_DS.pdf (Last accessed 15 Oct. 02)

[27] "FAAC and FAAD2 AAC software", *Audiocoding.com Web Site*, http://www.audiocoding.com (Last accessed 15 Oct. 02)

[28] "MPEG-2 Audio Conformance Bitstreams", *AT&T Labs Web Site*, http://www.research.att.com/projects/mpegaudio/mpeg2.html (Last accessed 15 Oct. 02)

[29] "XSV Board v1.1 Manual", *Xess Corporation Web Site*, http://www.xess.com/manuals/xsv-manual-v1_1.pdf (Last accessed 13 Oct. 02)

[30] "Virtex 2.5V Field Programmable Gate Arrays", *Xilinx Web Site*, http://direct.xilinx.com/bvdocs/publications/ds003.pdf (Last accessed 15 Oct. 02)

[31] Chang, Y. & Wong, D. & Wong, C. "*Programmable Logic Devices*", http://cc.ee.ntu.edu.tw/~ywchang/Papers/pla.ps (Last accessed 15 Oct. 02)

[32] AK4520A Datasheet, *AKM Web Site*, http://www.asahi-kasei.co.jp/akm/usa/product/ak4520a/ek4520a.pdf (Last accessed 19 Apr. 02)

[33] Brennan, J. 2001. "*Audio Project*", http://www.itee.uq.edu.au/~peters/xsvboard/audio/audio.pdf (Last accessed 21 Sep. 02)

[34] "XC95108 In-System Programmable CPLD", *Xilinx Web Site*, http://direct.xilinx.com/bvdocs/publications/95108.pdf (Last accessed 15 Oct. 02)

[35] Pedderson, J. 2001. "*Hardware Implementation of Video Streaming*", http://innovexpo.itee.uq.edu.au/2001/projects/s369604/thesis.pdf (Last accessed 15 Oct. 02)

[36] Khosravipour, M. "*VHDL Introduction*", http://www.ecsi.org/earnest/digests/VHDL_intro/VHDL-Intro.pdf (Last accessed 15 Oct. 02)

[37] “Synthesis”, *Xilinx Web Site*, http://toolbox.xilinx.com/docsan/xilinx4/data/docs/xug/flow5.html (Last accessed 21 Sep. 02)

[38] “Design Implementation”, *Xilinx Web Site*, http://toolbox.xilinx.com/docsan/xilinx4/data/docs/dev/dsgnflow4.html (Last accessed 21 Sep. 02)

[39] XSTools. Available at http://www.xess.com/ho07000.html (Last accessed 19 Apr. 02)

[40] “VHDL Reference”, http://www.itee.uq.edu.au/~comp3100/vhdl/vhdlref.pdf (Last accessed 16 Oct. 02)

[41] Brennan, J. & Pedderson J. 2001. “*SRAM Interface Project*”, http://www.itee.uq.edu.au/~peters/xsvboard/sram/sram.pdf (Last accessed 15 Oct. 02)

[42] “An Inverse Discrete Cosine Transform Implementation in Virtex for MPEG Video Applications”, *Xilinx Web Site*, http://www.xilinx.com/xapp/xapp208.pdf (Last accessed 14 Oct. 02)

[43] Andraka, R. “*A survey of CORDIC algorithms for FPGA based computers*”, http://www.andraka.com/files/crdcsrvy.pdf (Last accessed 14 Oct. 02)

[44] Vladimirova, P. “*FPGA Implementation of Sine and Cosine Generators using the CORDIC Algorithm*”, http://klabs.org/richcontent/MAPLDCon99/Papers/A2_Vladimirova_P.pdf (Last accessed 14 Oct. 02)

[45] Duhamel, P. & Mahieux, Y. & Petit, J. “A fast algorithm for the implementation of filter banks based on `time domain aliasing cancellation'”, *Proc. ICASSP*, IEEE, Toronto, 1991, vol 3, p2209-12.

APPENDIX A. LC CONFORMANCE BITSTREAMS

	L1_fs	L2_fs	L3_fs	L4_fs	L5_fs	L6_fs
bitrate	40/64	40/64	40/64	40/64	80/128	120/192
# single channel elements	1	0	1	1	0	1
# channel pair elements	0	0	0	0	1	1
# LFE channels	0	0	0	0	0	0
# Dep coupling channels	0	0	0	0	0	0
Data stream elements	Yes					
Intensity					Yes	
M/S					Yes	Yes
Window shape				Yes		
TNS				Yes		Yes
Pulse data				Yes		
Buffer test		Yes				
Arithmetic torture						
Nonmeaningful transitions			Yes			
ADTS						

	L7_fs	**L8_fs**	**L9_fs**	**L10_fs**	**L11_fs**
Bitrate	240/380	1920/3072	40/64	40/64	40/64
# single channel elements	1	16	1	1	1
# channel pair elements	2	16	0	0	0
# LFE channels	1	0	0	0	0
# Dep coupling channels	1	0	0	0	0
Data stream elements				Yes	Yes
Intensity					
M/S	Yes	Yes			
Window shape					
TNS	Yes	Yes			
Pulse data					
Buffer test			Yes		
Arithmetic torture					Yes
Nonmeaningful transitions					
ADTS			Yes		

APPENDIX B. DETAILS OF 'L1_fs_mod'

ADIF_SEQUENCE()

ADIF_HEADER()

Variable Name	**Value**	**Comment**
adif_id	44 41 49 46 (hex)	ADIF
copyright_id_present	0	No
original_copy	1	Yes
home	0	
bitstream_type	0	Constant Rate
bitrate	000 1111 1010 0000 0000	64kbits/sec
num_program_config_elements	0000	None
adif_buffer_fullness	0001 0000 0010 0000	4136

PROGRAM_CONFIG_ELEMENT()

Variable Name	**Value**	**Comment**
element_instance_tag	0000	
profile	01	Low Complexity
sampling_frequency_index	0011	48kHz
num_front_channel_elements	0001	Single
num_side_channel_elements	0000	
num_back_channel_elements	0000	
num_lfe_channel_elements	00	
num_assoc_data_elements	001	
num_valid_cc_elements	0000	
mono_mixdown_present	0	
stereo_mixdown_present	0	
matrix_mixdown_idx_present	0	
front_element_is_cpe[0]	0	
front_element_tag_select[0]	0000	
assoc_data_element_tag_select[0]	1111	
byte_alignment()	00	
comment_field_bytes	0011 1011	59 bytes
comment_field_data[]	Encoded by AT&T Laboratories, GMT Fri Dec 21 14:30:29 2001 <CR>	

RAW_DATA_STREAM()

RAW_DATA_BLOCK()

Variable Name	Value	Comment
id_syn_ele	100	ID_DSE

DATA_STREAM_ELEMENT()

Variable Name	Value	Comment
element_instance_tag	1111	15
data_byte_align_flag	1	Yes
count	0000 0010	2
data_stream_byte[15][0]	0000 0000	Zero
data_stream_byte[15][1]	1010 1011	171

RAW_DATA_BLOCK() (continued)

Variable Name	Value	Comment
id_syn_ele	000	ID_SCE

SINGLE_CHANNEL_ELEMENT()

Variable Name	Value	Comment
element_instance_tag	0000	Zero

INDIVIDUAL_CHANNEL_STREAM[0]()

Variable Name	Value	Comment
global_gain	0110 0100	100

ICS_INFO()

Variable Name	Value	Comment
ics_reserved_bit	0	
window_sequence	01	long_start_seq
window_shape	0	sine
max_sfb	10 1001	41
predictor_data_present	0	No
Additional Information		
Variable Name	**Value**	**Comment**
num_windows	1	
num_window_groups	1	
window_group_length[0]	1	
num_swb	49	
sect_sfb_offset[0][i]	See table 8.4 [6, p33,34]	
swb_offset[i]	See table 8.4 [6, p33,34]	

SECTION_DATA()

Variable Name	**Value**	**Comment**
sect_cb[0][0]	0000	ZERO_HCB
sect_len_incr	11111	31
sect_len_incr	01010	10
→ **Result**: sfb_cb[0][0→ 40] is ZERO_HCB		

SCALE_FACTOR_DATA()

INDIVIDUAL_CHANNEL_STREAM[0]() (continued)

Variable Name	**Value**	**Comment**
pulse_data_present	0	No
tns_data_present	0	No
gain_control_data_present	0	No

SPECTRAL_DATA()

RAW_DATA_BLOCK() (continued)

Variable Name	**Value**	**Comment**
id_syn_ele	110	ID_FIL

FILL_ELEMENT()

Variable Name	**Value**	**Comment**
count	1111	15
esc_count	1101 0101	149
extension_type	0101	Default
other_bits[0 → 1299]	01 x 162.5	Lots of fill bits

RAW_DATA_BLOCK() (continued)

Variable Name	**Value**	**Comment**
id_syn_ele	111	ID_TERM
byte_alignment()		

RAW_DATA_BLOCK() (NEW)

Variable Name	**Value**	**Comment**
id_syn_ele	100	ID_DSE

DATA_STREAM_ELEMENT()

Variable Name	Value	Comment
element_instance_tag	1111	15
data_byte_align_flag	1	Yes
count	0000 0010	2
data_stream_byte[15][0]	0000 0000	Zero
data_stream_byte[15][1]	1001 1101	157

RAW_DATA_BLOCK() (continued)

Variable Name	Value	Comment
id_syn_ele	000	ID_SCE

SINGLE_CHANNEL_ELEMENT()

Variable Name	Value	Comment
element_instance_tag	0000	Zero

INDIVIDUAL_CHANNEL_STREAM[0]()

Variable Name	Value	Comment
global_gain	0111 0110	118

ICS_INFO()

Variable Name	Value	Comment
ics_reserved_bit	0	
window_sequence	11	eight_short
window_shape	0	sine
max_sfb	1011	11
scale_factor_grouping	110 0111	103
Additional Information		
Variable Name	**Value**	**Comment**
num_windows	8	
num_window_groups	1	
window_group_length[0] window_group_length[1] window_group_length[2]	3 1 4	
num_swb	14	
sect_sfb_offset[g][i]	See table 8.5 [6, p33,34]	
swb_offset[i]	See table 8.5 [6, p33,34]	

SECTION_DATA()

Variable Name	Value	Comment
sect_cb[0][0]	0000	ZERO_HCB
sect_len_incr	111	7
sect_len_incr	100	4
→ **Result**: sfb_cb[0][0→ 10] is ZERO_HCB		
sect_cb[1][0]	1010	HCB 10
sect_len_incr	001	1
→ **Result**: sfb_cb[1][0] is HCB 10		
sect_cb[1][1]	0100	HCB 4
sect_len_incr	111	7
sect_len_incr	011	3
→ **Result**: sfb_cb[1][1→ 10] is HCB 4		
sect_cb[2][0]	1011	HCB 11
sect_len_incr	011	3
→ **Result**: sfb_cb[2][0→ 2] is HCB 11		
sect_cb[2][1]	0110	HCB 6
sect_len_incr	100	4
→ **Result**: sfb_cb[2][3→ 6] is HCB 6		
sect_cb[2][2]	1000	HCB 8
sect_len_incr	001	1
→ **Result**: sfb_cb[2][7] is HCB 8		
sect_cb[2][3]	0110	HCB 6
sect_len_incr	011	3
→ **Result**: sfb_cb[2][8→ 10] is HCB 6		

SCALE_FACTOR_DATA()

Variable Name	Value	Decoded	Value
		sf[0][0 → 10]	0
hcod_sf[dpcm_sf[1][0 → 10]]	0	sf[1][0 → 10]	118
hcod_sf[dpcm_sf[2][0]]	0	sf[2][0]	118
hcod_sf[dpcm_sf[2][1]]	1100	sf[2][1]	120
hcod_sf[dpcm_sf[2][2]]	1011	sf[2][2]	118
hcod_sf[dpcm_sf[2][3]]	0	sf[2][3]	118
hcod_sf[dpcm_sf[2][4]]	111 1010	sf[2][4]	125
hcod_sf[dpcm_sf[2][5]]	1011	sf[2][5]	123
hcod_sf[dpcm_sf[2][6]]	100	sf[2][6]	122
hcod_sf[dpcm_sf[2][7]]	100	sf[2][7]	121
hcod_sf[dpcm_sf[2][8]]	0	sf[2][8]	121
hcod_sf[dpcm_sf[2][9]]	0	sf[2][9]	121
hcod_sf[dpcm_sf[2][10]]	1100	sf[2][10]	123

INDIVIDUAL_CHANNEL_STREAM[0]() (continued)

Variable Name	**Value**	**Comment**
pulse_data_present	0	No
tns_data_present	0	No
gain_control_data_present	0	No

SPECTRAL_DATA()

g = 0, ZERO_HCB → no spectral data

g = 1, HCB 10

hcod[10][y][z]	**pair_sign_bits**	**y**	**z**
11 1110 0000	11	-9	-8
101 0110	11	-5	-3

g = 1, HCB 4

hcod[4][y][z]	**quad_sign_bits**	**w**	**x**	**y**	**z**
1 1111 0000	0000	1	2	2	1
1110 1111	00	0	0	1	2
110 1000	0000	2	1	1	1
110 1011	0000	1	1	1	2
1 1111 0001	0000	2	2	1	1
0001	111	0	-1	-1	-1
0100	100	-1	0	1	1
0100	011	1	0	-1	-1
0000	1111	-1	-1	-1	-1
110 1101	110	-2	-1	0	1
0000	0000	1	1	1	1
0010	001	1	1	0	-1
0000	1100	-1	-1	1	1
0010	000	1	1	1	0
0001	111	0	-1	-1	-1
0101	1	-1	0	0	0
0000	1111	-1	-1	-1	-1
0011	111	-1	-1	-1	0
1 0110	1	0	0	0	-1

g = 2, HCB 11

hcod[11][y][z]	**pair_sign_bits**	**y**	**z**
111 1110 0111	10	-12	12
1011 1000	10	-2	8
1 1000 1111	01		
→ with escape sequence 10 0000		32	-15
1001 1110	00	5	5

1 1011 0000	00	10	4
1011 0000	10	-6	3
1001 0111	00		
→ with escape sequence 0000		16	4
11 1101 1111	1	0	-9
11 1010 0000	10	-11	1
0 0111	00	2	1
11 1100 1100	11	-15	-4
0001	00	1	1
11 1001 0100	11	-13	-4
01 1000	11	-2	-3
11 1001 0111	10	-13	5
011 1100	1	-3	0
0 1001	10	-2	2
1010 1011	11	-7	-1
0 1000	01	1	-2
011 1010	00	4	3
0001	00	1	1
011 0111	11	-1	-4
0 1000	11	-1	-2
1011 1100	10	-8	3

g = 2, HCB 6 (entire left column first then entire right column – even over the page…)

hcod[6][y][z]	**y**	**z**	**hcod[6][y][z]**	**y**	**z**
11 0010	-2	-2	0000	0	0
0110	-1	-1	0100	-1	0
10 1111	0	2	10 1111	0	2
0100	-1	0	10 0101	2	1
0111	1	-1	0100	-1	0
1000	-1	-1	10 1010	2	0
0101	1	1	0010	-1	1
1110 1011	-2	3	0100	-1	0
111 0001	3	3	10 1110	-1	-2
0010	0	-1	1000	-1	-1
0010	0	-1	10 0110	-2	1
0110	-1	1	0011	0	1
0100	-1	0	10 1010	2	0
0110	-1	1	111 0010	0	-3
1000	-1	-1	1000	-1	-1
1000	-1	-1	0110	-1	1
1000	-1	-1	0010	0	-1
0100	-1	0	1 1111 0011	-2	-4
1000	-1	-1	0111	1	-1
0111	1	-1	10 1100	1	2
0101	1	1	10 1111	0	2
10 0101	2	1	0011	0	1

0110	-1	1	10 1010	2	0
0001	1	0	0101	1	1

g = 2, HCB 8

hcod[8][y][z]	**pair_sign_bits**	**y**	**z**
0110	10	-2	2
11 0010	01	3	-4
111 0011	00	6	2
0010	01	2	-1
10 1011	00	4	1
0110	01	2	-2
0110	10	-2	2
000	01	1	-1
1 0011	00	3	2
0100	11	-1	-2
1 0000	1	0	-2
000	10	-1	1
111 0011	00	6	2
000	00	1	1
1 0010	00	1	3
0101	1	0	-1

g = 2, HCB 6 (entire left column first then entire right column – even over the page…)

hcod[6][y][z]	**y**	**z**	**hcod[6][y][z]**	**y**	**z**
0111	1	-1	0100	-1	0
0000	0	0	0100	-1	0
1000	-1	-1	1 1111 0110	4	-2
110 111o	1	-3	0100	-1	0
11 0001	-2	2	10 1011	1	-2
0110	-1	1	0000	0	0
11 0001	-2	2	0110	-1	1
10 0101	2	1	0111	1	-1
0110	-1	1	0101	1	1
0110	-1	1	1 1110 1000	-1	-4
0110	-1	1	1000	-1	-1
11 0000	2	-2	110 1000	-3	1
0110	-1	1	0001	1	0
10 1000	-2	0	10 1100	1	2
10 1111	0	2	0111	1	-1
0010	0	-1	0100	-1	0
110 1110	1	-3	0110	-1	1
1000	-1	-1	10 0100	2	-1
0011	0	1	0001	1	0
1000	-1	-1	10 1001	-1	2
11 0010	-2	-2	0001	1	0
1110 1110	2	-3	110 1110	1	-3
0111	1	-1	0001	1	0

10 1010	2	0	0010	0	-1
0010	0	-1	0001	1	0
0111	1	-1	0001	1	0
0111	1	-1	0100	-1	0
0010	0	-1	0000	0	0
10 0110	-2	1	10 1101	0	-2
10 0101	2	1	10 1111	0	2
11 0000	2	-2	10 0110	-2	1
11 0010	-2	-2	0010	0	-1
10 1011	1	-2	1000	-1	-1
0010	0	-1	110 1111	-1	-3
0010	0	-1	0111	1	-1
11 0001	-2	2	1000	-1	-1

RAW_DATA_BLOCK() (continued)

Variable Name	**Value**	**Comment**
id_syn_ele	111	ID_TERM
byte_alignment()		

APPENDIX C. VHDL SOURCE CODE

This appendix contains selected code samples showing the functionality

C.1 bitstream_top.vhd

```
----------------------------------------------------------------------
-- bitstream_top -----------------------------------------------------
----------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity bitstream_top is
      port (
      clk: in STD_LOGIC;
      reset: in STD_LOGIC;
      bits_in: in STD_LOGIC;
      clock_48: out STD_LOGIC
      );
end bitstream_top;

architecture bitstream_top of bitstream_top is

component shift_register_32 is
      port (
      clk: in STD_LOGIC;
      reset: in STD_LOGIC;
      shift_in: in STD_LOGIC;
      shift_data: out STD_LOGIC_VECTOR(31 downto 0);
      byte_data: out STD_LOGIC_VECTOR(2 downto 0)
      );
end component;

component bitstream_counter is
      port (
      clk: in STD_LOGIC;
      reset: in STD_LOGIC;
      latch: in STD_LOGIC;
      latch_data: in STD_LOGIC_VECTOR(10 downto 0);
      count_data: out STD_LOGIC_VECTOR(10 downto 0)
      );
end component;

component bitstream_state_machine is
      port(
      clk: in STD_LOGIC;
      reset: in STD_LOGIC;
      count_latch: out STD_LOGIC;
      count_value: in STD_LOGIC_VECTOR(10 downto 0);
      count_set: out STD_LOGIC_VECTOR(10 downto 0);
      byte_value: in STD_LOGIC_VECTOR(2 downto 0);
      data_available: in STD_LOGIC_VECTOR(31 downto 0)
      );
end component;

-- Clock generation signals
signal clock_count: STD_LOGIC_VECTOR(10 downto 0);
signal clock_48kHz: STD_LOGIC;

signal count_latch: STD_LOGIC;
signal count_value: STD_LOGIC_VECTOR(10 downto 0);
signal count_set: STD_LOGIC_VECTOR(10 downto 0);
signal byte_value: STD_LOGIC_VECTOR(2 downto 0);
signal data_available: STD_LOGIC_VECTOR(31 downto 0);

begin

      U1: shift_register_32 port map (
      clk => clock_48khz,
      reset => reset,
      shift_in => bits_in,
      shift_data => data_available,
      byte_data => byte_value
      );

      U2: bitstream_counter port map (
      clk => clock_48khz,
      reset => reset,
      latch => count_latch,
      latch_data => count_set,
      count_data => count_value
      );

      U3: bitstream_state_machine port map (
      clk => clock_48kHz,
      reset => reset,
      count_latch => count_latch,
      count_value => count_value,
      count_set => count_set,
      byte_value => byte_value,
      data_available => data_available
      );

      clock_48 <= clock_48kHz;

      process(clk,reset)
      begin

            if(reset = '0') then
                  clock_48khz <= '0';
                  clock_count <= "00000000000";
            else
                  if(clk'event and clk = '1') then
                        clock_count <= clock_count + 1;
                        clock_48khz <= clock_count(10);
                  end if;
            end if;
      end process;
end bitstream_top;
```

C.2 bitstream.vhd

```
----------------------------------------------------------------------
-- bitstream_state_machine -------------------------------------------
----------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity bitstream_state_machine is
      port(
      clk: in STD_LOGIC;
      reset: in STD_LOGIC;
      count_latch: out STD_LOGIC;
      count_value: in STD_LOGIC_VECTOR(10 downto 0);
      count_set: out STD_LOGIC_VECTOR(10 downto 0);
      byte_value: in STD_LOGIC_VECTOR(2 downto 0);
      data_available: in STD_LOGIC_VECTOR(31 downto 0)
      );
end bitstream_state_machine;

architecture bitstream_state_machine of bitstream_state_machine is

type state_type is
(header, raw_data_block,
 data_stream_element, data_stream_element2, data_stream_element3,
 single_channel_element, ics_info_long, ics_info_short,
 section_data,
 error_state, idle_state);

signal present_state: state_type;
signal next_state: state_type;

-- data_stream_element signals
signal data_byte_align_flag: STD_LOGIC;
signal dse_count: STD_LOGIC_VECTOR(8 downto 0);
signal temp: STD_LOGIC;

-- single_channel_element signals
signal global_gain: STD_LOGIC_VECTOR(7 downto 0);
signal window_sequence: STD_LOGIC_VECTOR(1 downto 0);
signal window_shape: STD_LOGIC;
signal max_sfb: STD_LOGIC_VECTOR(5 downto 0);
signal scale_factor_grouping: STD_LOGIC_VECTOR(6 downto 0);

begin

      process(clk,reset)
      begin

            if(reset = '0') then
                  present_state <= idle_state;
            else
                  if(clk'event and clk = '1') then
                        present_state <= next_state;
                  end if;
            end if;
end process;

process(present_state, count_value, data_available, byte_value)
begin

case present_state is

when idle_state =>
      next_state <= header;
      count_set <= "01001011111";   -- 608 bit header
      count_latch <= '1';
      data_byte_align_flag <= 'X';
      dse_count <= "XXXXXXXXX";
      global_gain <= "XXXXXXXX";
      window_sequence <= "XX";
      window_shape <= 'X';
      max_sfb <= "XXXXXX";
      scale_factor_grouping <= "XXXXXXX";

when header =>

      data_byte_align_flag <= 'X';
      dse_count <= "XXXXXXXXX";
      global_gain <= "XXXXXXXX";
      window_sequence <= "XX";
      window_shape <= 'X';
      max_sfb <= "XXXXXX";
      scale_factor_grouping <= "XXXXXXX";

      if(count_value = "00000000000") then
            count_latch <= '1';
            next_state <= raw_data_block;
            count_set <= "00000000010";
      else
            count_latch <= '0';
            next_state <= header;
            count_set <= "XXXXXXXXXXX";
      end if;

when raw_data_block =>

      data_byte_align_flag <= 'X';
      dse_count <= "XXXXXXXXX";
      global_gain <= global_gain;
      window_sequence <= window_sequence;
      window_shape <= window_shape;
      max_sfb <= max_sfb;
      scale_factor_grouping <= scale_factor_grouping;

      if(count_value = "00000000000") then
            count_latch <= '1';
            case data_available(2 downto 0) is
                  when "000" =>     -- ID_SCE
                        next_state <= single_channel_element;
                        count_set <= "00000001111";   -- 16
                  when "100" =>     -- ID_DSE
                        next_state <= data_stream_element;
                        count_set <= "00000001100";   -- 12
                  when "111" =>     -- ID_END
```

```
                        next_state <= raw_data_block;
                        count_set <= "00000000010";   -- 2
                  when others =>
                        next_state <= error_state;
                        count_set <= "XXXXXXXXXXX";   -- 2
            end case;
      else
            -- Wait for the counter to get to zero.
            next_state <= raw_data_block;
            count_set <= "XXXXXXXXXXX";
            count_latch <= '0';
      end if;

when data_stream_element =>

      global_gain <= global_gain;
      window_sequence <= window_sequence;
      window_shape <= window_shape;
      max_sfb <= max_sfb;
      scale_factor_grouping <= scale_factor_grouping;

      if(count_value = "00000000000") then
            data_byte_align_flag <= data_available(8);
            count_latch <= '1';
            dse_count(7 downto 0)
                  <= data_available(7 downto 0) - 1;
            dse_count(8) <= '0';
            if(data_available(7 downto 0) = "11111111") then
                  next_state <= data_stream_element2;
                  count_set <= "00000000111";   -- 7
            else
                  next_state <= data_stream_element3;
                  -- Check the byte alignment
                  if(data_byte_align_flag = '1') then
                        count_set <= "00000000110" + byte_value;
                  else
                        count_set <= "00000000110";   -- 7
                  end if;
            end if;
      else
            next_state <= data_stream_element;
            count_set <= "XXXXXXXXXXX";
            count_latch <= '0';
            dse_count <= "XXXXXXXXX";
            data_byte_align_flag <= 'X';
      end if;

when data_stream_element2 =>

      global_gain <= global_gain;
      window_sequence <= window_sequence;
      window_shape <= window_shape;
      max_sfb <= max_sfb;
      scale_factor_grouping <= scale_factor_grouping;

      if(count_value = "00000000000") then
            dse_count <= "100000000" +
                  data_available(7 downto 0) - 1;
            data_byte_align_flag <= data_available(16);
```

```
                    next_state <= data_stream_element3;
                    count_latch <= '1';
                    -- Check the byte alignment
                    if(data_byte_align_flag = '1') then
                          count_set <= "00000000110" + byte_value;
                    else
                          count_set <= "00000000110";   -- 7
                    end if;
              else
                    next_state <= data_stream_element2;
                    count_set <= "XXXXXXXXXXX";
                    count_latch <= '0';
                    data_byte_align_flag <= 'X';
                    dse_count <= "XXXXXXXXX";
              end if;

        when data_stream_element3 =>

              data_byte_align_flag <= 'X';
              global_gain <= global_gain;
              window_sequence <= window_sequence;
              window_shape <= window_shape;
              max_sfb <= max_sfb;
              scale_factor_grouping <= scale_factor_grouping;

              if(count_value = "11111111111") then
                    count_latch <= '1';
                    if(dse_count = "000000000") then
                          next_state <= raw_data_block;
                          count_set <= "00000000010";   -- 2
                          dse_count <= "XXXXXXXXX";
                    else
                          next_state <= data_stream_element3;
                          count_set <= "00000000110";   -- 7
                          dse_count <= dse_count - 1;
                    end if;
              else
                    next_state <= data_stream_element3;
                    count_set <= "XXXXXXXXXXX";
                    count_latch <= '0';
                    dse_count <= dse_count;
              end if;

        when single_channel_element =>

              data_byte_align_flag <= 'X';
              dse_count <= "XXXXXXXXX";
              max_sfb <= max_sfb;
              scale_factor_grouping <= scale_factor_grouping;

              if(count_value = "00000000000") then
                    count_latch <= '1';
                    global_gain <= data_available(11 downto 4);
                    window_sequence <= data_available(2 downto 1);
                    window_shape <= data_available(0);
                    if(window_sequence = "10") then
                          -- EIGHT_SHORT_SEQUENCE
                          next_state <= ics_info_short;
                          count_set <= "00000001010";   -- 10
```

```
            else
                  -- LONG WINDOWS
                  next_state <= ics_info_long;
                  count_set <= "00000000110";   -- 6
            end if;
      else
            next_state <= single_channel_element;
            count_set <= "XXXXXXXXXXX";
            count_latch <= '0';
            global_gain <= global_gain;
            window_sequence <= window_sequence;
            window_shape <= window_shape;
      end if;

when ics_info_long =>

      data_byte_align_flag <= 'X';
      dse_count <= "XXXXXXXXX";
      global_gain <= global_gain;
      window_sequence <= window_sequence;
      window_shape <= window_shape;
      scale_factor_grouping <= scale_factor_grouping;

      if(count_value = "00000000000") then
            next_state <= section_data;
            count_latch <= '1';
            count_set <= "00000000010";   -- 2
            max_sfb <= data_available(6 downto 1);
      else
            next_state <= ics_info_long;
            count_latch <= '0';
            count_set <= "XXXXXXXXXXX";
            max_sfb <= max_sfb;
      end if;

when ics_info_short =>

      data_byte_align_flag <= 'X';
      dse_count <= "XXXXXXXXX";
      global_gain <= global_gain;
      window_sequence <= window_sequence;
      window_shape <= window_shape;

      if(count_value = "00000000000") then
            next_state <= section_data;
            count_latch <= '1';
            count_set <= "00000000010";   -- 2
            max_sfb(3 downto 0) <= data_available(10 downto 7);
            max_sfb(5 downto 4) <= "00";
            scale_factor_grouping <= data_available(6 downto 0);
      else
            next_state <= ics_info_long;
            count_latch <= '0';
            count_set <= "XXXXXXXXXXX";
            max_sfb <= max_sfb;
            scale_factor_grouping <= scale_factor_grouping;
      end if;

when section_data =>
```

```
            next_state <= idle_state;
            count_latch <= '0';
            data_byte_align_flag <= 'X';
            dse_count <= "XXXXXXXXX";
            count_set <= "XXXXXXXXXXX";
            global_gain <= "XXXXXXXX";
            window_sequence <= "XX";
            window_shape <= 'X';
            max_sfb <= "XXXXXX";
            scale_factor_grouping <= "XXXXXXX";

      when error_state =>
            next_state <= idle_state;
            count_latch <= 'X';
            data_byte_align_flag <= 'X';
            dse_count <= "XXXXXXXXX";
            count_set <= "XXXXXXXXXXX";
            global_gain <= "XXXXXXXX";
            window_sequence <= "XX";
            window_shape <= 'X';
            max_sfb <= "XXXXXX";
            scale_factor_grouping <= "XXXXXXX";

      end case;
      end process;
end bitstream_state_machine;
```

C.3 counter.vhd

```
----------------------------------------------------------------------
-- bitstream_counter -------------------------------------------------
----------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity bitstream_counter is
      port (
      clk: in STD_LOGIC;
      reset: in STD_LOGIC;
      latch: in STD_LOGIC;
      latch_data: in STD_LOGIC_VECTOR(10 downto 0);
      count_data: out STD_LOGIC_VECTOR(10 downto 0)
      );
end bitstream_counter;

architecture bitstream_counter of bitstream_counter is

signal data: STD_LOGIC_VECTOR(10 downto 0);

begin
      count_data <= data;
      process(clk,reset)
      begin
            if (reset = '0') then
                  data <= "00000000011";
            else
                  if(clk'event and clk = '1') then
                        if (latch = '1') then
                              data <= latch_data;
                        else
                              data <= data - 1;
                        end if;
                  end if;
            end if;
      end process;
end bitstream_counter;
```

C.4 *shift_register.vhd*

```
----------------------------------------------------------------------
-- shift_register_32 -------------------------------------------------
----------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity shift_register_32 is
      port (
      clk: in STD_LOGIC;
      reset: in STD_LOGIC;
      shift_in: in STD_LOGIC;
      shift_data: out STD_LOGIC_VECTOR(31 downto 0);
      byte_data: out STD_LOGIC_VECTOR(2 downto 0)
      );
end shift_register_32;

architecture shift_register_32 of shift_register_32 is

signal data: STD_LOGIC_VECTOR(31 downto 0);
signal byte: STD_LOGIC_VECTOR(2 downto 0);

begin

      shift_data <= data;
      byte_data <= byte;
      process(clk,reset)
      begin

            if(reset = '0') then
                  data <= (others=>'0');
                  byte <= (others=>'0');
            else
                  if(clk'event and clk = '1') then
                        data(31 downto 1) <= data(30 downto 0);
                        data(0) <= shift_in;
                        byte <= byte - 1;
                  end if;
            end if;
      end process;
end shift_register_32;
```

C.5 scale_factor_table.vhd

```
----------------------------------------------------------------------
-- scalefactor_table -------------------------------------------------
----------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity scalefactor_table is
      port (
      index: in STD_LOGIC_VECTOR(5 downto 0);
      window: in STD_LOGIC_VECTOR(1 downto 0);
      swb_offset: out STD_LOGIC_VECTOR(10 downto 0)
      );
end scalefactor_table;

architecture scalefactor_table of scalefactor_table is

begin
      process(window, index)
      begin
            case window is
            -- SHORT_WINDOW
            when "10" =>
                  case index is
                        when "000000" => swb_offset <= "00000000000";
                        when "000001" => swb_offset <= "00000000100";
                        when "000010" => swb_offset <= "00000001000";
                        when "000011" => swb_offset <= "00000001100";
                        when "000100" => swb_offset <= "00000010000";
                        when "000101" => swb_offset <= "00000010100";
                        when "000110" => swb_offset <= "00000011100";
                        when "000111" => swb_offset <= "00000100100";
                        when "001000" => swb_offset <= "00000101100";
                        when "001001" => swb_offset <= "00000111000";
                        when "001010" => swb_offset <= "00001000100";
                        when "001011" => swb_offset <= "00001010000";
                        when "001100" => swb_offset <= "00001100000";
                        when "001101" => swb_offset <= "00001110000";
                        when "001110" => swb_offset <= "00010000000";
                        when others => swb_offset <= "XXXXXXXXXXX";
                  end case;

            -- LONG_WINDOW, LONG_START_WINDOW, LONG_STOP_WINDOW
            when others =>
                  case index is
                        when "000000" => swb_offset <= "00000000000";
                        when "000001" => swb_offset <= "00000000100";
                        when "000010" => swb_offset <= "00000001000";
                        when "000011" => swb_offset <= "00000001100";
                        when "000100" => swb_offset <= "00000010000";
                        when "000101" => swb_offset <= "00000010100";
                        when "000110" => swb_offset <= "00000011000";
                        when  "000111"  =>  swb_offset  <=  "00000011100";
                        when "001000" => swb_offset <= "00000100000";
                        when "001001" => swb_offset <= "00000100100";
                        when "001010" => swb_offset <= "00000101000";
                        when "001011" => swb_offset <= "00000110000";
                        when "001100" => swb_offset <= "00000111000";
                        when "001101" => swb_offset <= "00001000000";
                        when "001110" => swb_offset <= "00001001000";
                        when "001111" => swb_offset <= "00001010000";
                        when "010000" => swb_offset <= "00001011000";
                        when "010001" => swb_offset <= "00001100000";
                        when "010010" => swb_offset <= "00001101100";
                        when "010011" => swb_offset <= "00001111000";
                        when "010100" => swb_offset <= "00010000100";
                        when "010101" => swb_offset <= "00010010000";
                        when "010110" => swb_offset <= "00010100000";
                        when "010111" => swb_offset <= "00010110000";
                        when "011000" => swb_offset <= "00011000100";
                        when "011001" => swb_offset <= "00011011000";
                        when "011010" => swb_offset <= "00011110000";
                        when "011011" => swb_offset <= "00100001000";
                        when "011100" => swb_offset <= "00100100100";
                        when "011101" => swb_offset <= "00101000000";
                        when "011110" => swb_offset <= "00101100000";
                        when "011111" => swb_offset <= "00110000000";
                        when "100000" => swb_offset <= "00110100000";
                        when "100001" => swb_offset <= "00111000000";
                        when "100010" => swb_offset <= "00111100000";
                        when "100011" => swb_offset <= "01000000000";
                        when "100100" => swb_offset <= "01000100000";
                        when "100101" => swb_offset <= "01001000000";
                        when "100110" => swb_offset <= "01001100000";
                        when "100111" => swb_offset <= "01010000000";
                        when "101000" => swb_offset <= "01010100000";
                        when "101001" => swb_offset <= "01011000000";
                        when "101010" => swb_offset <= "01011100000";
                        when "101011" => swb_offset <= "01100000000";
                        when "101100" => swb_offset <= "01100100000";
                        when "101101" => swb_offset <= "01101000000";
                        when "101110" => swb_offset <= "01101100000";
                        when "101111" => swb_offset <= "01110000000";
                        when "110000" => swb_offset <= "01110100000";
                        when "110001" => swb_offset <= "10000000000";
                        when others => swb_offset <= "XXXXXXXXXXX";
                  end case;
            end case;
      end process;
end scalefactor_table;
```

C.6 forloop.vhd

```
----------------------------------------------------------------------
-- ics_info() --------------------------------------------------------
----------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity forloop is
```

```
      port (
      clk: in STD_LOGIC;
      reset: in STD_LOGIC;
      window_sequence: in STD_LOGIC_VECTOR(1 downto 0);
      scale_factor_grouping: in STD_LOGIC_VECTOR(6 downto 0);
      max_sfb: in INTEGER;
      index: out INTEGER;
      sf_table_value: in STD_LOGIC_VECTOR(10 downto 0)
      );
end forloop;

architecture forloop of forloop is

type LONG_ARRAY is array (48 downto 0) of STD_LOGIC_VECTOR(10 downto
0);
type SHORT_ARRAY is array (13 downto 0) of STD_LOGIC_VECTOR(10 downto
0);

signal num_windows: INTEGER;
signal num_window_groups: INTEGER;
signal window_group_length: INTEGER;
signal num_swb: INTEGER;
signal sect_sfb_offset: LONG_ARRAY;
signal swb_offset: LONG_ARRAY;

begin

      process(clk,reset)
      begin
            if (reset = '0') then

                  num_windows <= 0;
                  num_window_groups <= 0;
                  window_group_length <= 0;
                  num_swb <= 0;
                  for i in 0 to 48 loop
                        sect_sfb_offset(i) <= "00000000000";
                        swb_offset(i) <= "00000000000";
                  end loop;

            else
                  if(clk'event and clk = '1') then

                        case window_sequence is

                              when "10" =>
                              -- EIGHT_SHORT_WINDOWS

                              when others =>
                              -- LONG WINDOWS
                              num_windows <= 1;
                              num_window_groups <= 1;
                              window_group_length <= 1;
                              num_swb <= 49;

                              for i in 0 to 48 loop
                                    index <= i;
                                    sect_sfb_offset(i)
                                          <= sf_table_value;
                                    swb_offset(i) <= sf_table_value;
                              end loop;

                        end case;


                  end if;
            end if;
      end process;
end forloop;
```

C.7 quantisationTop.vhd

```
----------------------------------------------------------------------
-- Quantisation Top Level --------------------------------------------
----------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity quantisation_top is
      port (
      x_quant: in STD_LOGIC_VECTOR(6 downto 0);
      scalefactor: in STD_LOGIC_VECTOR(7 downto 0);
      x_rescal: out STD_LOGIC_VECTOR(46 downto 0)
      );
end quantisation_top;

architecture quantisation_top of quantisation_top is

-- Component declarations

component inverse_quantisation is
      port(
      x_quant: in STD_LOGIC_VECTOR(6 downto 0);
      x_invquant: out STD_LOGIC_VECTOR(9 downto 0)
      );
end component;

component apply_scalefactors is
      port(
      x_invquant: in STD_LOGIC_VECTOR(9 downto 0);
      scalefactor: in STD_LOGIC_VECTOR(7 downto 0);
      x_rescal: out STD_LOGIC_VECTOR(49 downto 0)
      );
end component;

component quantisation_adjustment is
      port(
      sign_bit: in STD_LOGIC;
      x_rescal: in STD_LOGIC_VECTOR(49 downto 0);
      x_adjust: out STD_LOGIC_VECTOR(46 downto 0)
      );
end component;
```

```
-- Type declarations

-- Signal declarations
signal invquant: STD_LOGIC_VECTOR(9 downto 0);
signal rescal: STD_LOGIC_VECTOR(49 downto 0);
signal sign_bit: STD_LOGIC;

-- Constant declarations
-- Attribute declarations

begin

      U1: inverse_quantisation port map (
      x_quant => x_quant,
      x_invquant => invquant
      );

      U2: apply_scalefactors port map (
      x_invquant => invquant,
      scalefactor => scalefactor,
      x_rescal => rescal
      );

      U3: quantisation_adjustment port map (
      sign_bit => sign_bit,
      x_rescal => rescal,
      x_adjust => x_rescal
      );

      sign_bit <= x_quant(6);

end quantisation_top;
```

C.8 *invquant.vhd*

```
----------------------------------------------------------------------
-- Inverse Quantisation ----------------------------------------------
----------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity inverse_quantisation is
      port(
      x_quant: in STD_LOGIC_VECTOR(6 downto 0);
      x_invquant: out STD_LOGIC_VECTOR(9 downto 0)
      );
end inverse_quantisation;

architecture inverse_quantisation of inverse_quantisation is

-- Signal declarations
signal twos_xquant: STD_LOGIC_VECTOR(6 downto 0);
signal xquant_in: STD_LOGIC_VECTOR(6 downto 0);

begin
```

```
      -- Concurrent assignments
      twos_xquant <= not x_quant + 1;

      with x_quant(6) select
      xquant_in <= x_quant when '0',
                   twos_xquant when '1',
                   "XXXXXXX" when others;

      with xquant_in select                              --    value
      x_invquant <=     "0000000000" when "0000000",  --      0
                        "0000001000" when "0000001",  --      8
                        "0000010100" when "0000010",  --     20
                        "0000100011" when "0000011",  --     35
                        "0000110011" when "0000100",  --     51
                        "0001000100" when "0000101",  --     68
                        "0001010111" when "0000110",  --     87
                        "0001101011" when "0000111",  --    107
                        "0010000000" when "0001000",  --    128
                        "0010010110" when "0001001",  --    150
                        "0010101100" when "0001010",  --    172
                        "0011000100" when "0001011",  --    196
                        "0011011100" when "0001100",  --    220
                        "0011110101" when "0001101",  --    245
                        "0100001110" when "0001110",  --    270
                        "0100101000" when "0001111",  --    296
                        "0101000011" when "0010000",  --    323
                        "0101011110" when "0010001",  --    350
                        "0101111001" when "0010010",  --    377
                        "0110010110" when "0010011",  --    406
                        "0110110010" when "0010100",  --    434
                        "0111001111" when "0010101",  --    463
                        "0111101101" when "0010110",  --    493
                        "1000001011" when "0010111",  --    523
                        "1000101010" when "0011000",  --    554
                        "1001001001" when "0011001",  --    585
                        "1001101000" when "0011010",  --    616
                        "1010001000" when "0011011",  --    648
                        "1010101000" when "0011100",  --    680
                        "1011001001" when "0011101",  --    713
                        "1011101010" when "0011110",  --    746
                        "1100001011" when "0011111",  --    779
                        "1100101101" when "0100000",  --    813
                        "XXXXXXXXXX" when others;

end inverse_quantisation;
```

C.9 scaleapp.vhd

```
----------------------------------------------------------------------
-- Scalefactor Application -------------------------------------------
----------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
```

```
entity apply_scalefactors is
      port(
      x_invquant: in STD_LOGIC_VECTOR(9 downto 0);
      scalefactor: in STD_LOGIC_VECTOR(7 downto 0);
      x_rescal: out STD_LOGIC_VECTOR(49 downto 0)
      );
end apply_scalefactors;

architecture apply_scalefactors of apply_scalefactors is

-- Signal declarations
signal gain: STD_LOGIC_VECTOR(42 downto 0);
signal x_rescal8: STD_LOGIC_VECTOR(52 downto 0);

begin

      -- Concurrent assignments
      with scalefactor select
gain <= "0000000000000000000000000000000000000000000" when "00000000",
      "0000000000000000000000000000000000000000000" when "00000001",
      "0000000000000000000000000000000000000000000" when "00000010",
      "0000000000000000000000000000000000000000000" when "00000011",
      "0000000000000000000000000000000000000000000" when "00000100",
      "0000000000000000000000000000000000000000000" when "00000101",
      "0000000000000000000000000000000000000000000" when "00000110",
      "0000000000000000000000000000000000000000000" when "00000111",
      "0000000000000000000000000000000000000000000" when "00001000",
      "0000000000000000000000000000000000000000000" when "00001001",
      "0000000000000000000000000000000000000000000" when "00001010",
      "0000000000000000000000000000000000000000000" when "00001011",
      "0000000000000000000000000000000000000000000" when "00001100",
      "0000000000000000000000000000000000000000000" when "00001101",
      "0000000000000000000000000000000000000000000" when "00001110",
      "0000000000000000000000000000000000000000000" when "00001111",
      "0000000000000000000000000000000000000000000" when "00010000",
      "0000000000000000000000000000000000000000000" when "00010001",
      "0000000000000000000000000000000000000000000" when "00010010",
      "0000000000000000000000000000000000000000000" when "00010011",
      "0000000000000000000000000000000000000000000" when "00010100",
      "0000000000000000000000000000000000000000000" when "00010101",
      "0000000000000000000000000000000000000000000" when "00010110",
      "0000000000000000000000000000000000000000000" when "00010111",
      "0000000000000000000000000000000000000000000" when "00011000",
      "0000000000000000000000000000000000000000000" when "00011001",
      "0000000000000000000000000000000000000000000" when "00011010",
      "0000000000000000000000000000000000000000000" when "00011011",
      "0000000000000000000000000000000000000000000" when "00011100",
      "0000000000000000000000000000000000000000000" when "00011101",
      "0000000000000000000000000000000000000000000" when "00011110",
      "0000000000000000000000000000000000000000000" when "00011111",
      "0000000000000000000000000000000000000000000" when "00100000",
      "0000000000000000000000000000000000000000000" when "00100001",
      "0000000000000000000000000000000000000000000" when "00100010",
      "0000000000000000000000000000000000000000000" when "00100011",
      "0000000000000000000000000000000000000000000" when "00100100",
      "0000000000000000000000000000000000000000000" when "00100101",
      "0000000000000000000000000000000000000000000" when "00100110",
      "0000000000000000000000000000000000000000000" when "00100111",
      "0000000000000000000000000000000000000000000" when "00101000",
"0000000000000000000000000000000000000000000" when "00101001",
"0000000000000000000000000000000000000000000" when "00101010",
"0000000000000000000000000000000000000000000" when "00101011",
"0000000000000000000000000000000000000000000" when "00101100",
"0000000000000000000000000000000000000000000" when "00101101",
"0000000000000000000000000000000000000000000" when "00101110",
"0000000000000000000000000000000000000000000" when "00101111",
"0000000000000000000000000000000000000000000" when "00110000",
"0000000000000000000000000000000000000000000" when "00110001",
"0000000000000000000000000000000000000000000" when "00110010",
"0000000000000000000000000000000000000000000" when "00110011",
"0000000000000000000000000000000000000000000" when "00110100",
"0000000000000000000000000000000000000000000" when "00110101",
"0000000000000000000000000000000000000000000" when "00110110",
"0000000000000000000000000000000000000000000" when "00110111",
"0000000000000000000000000000000000000000000" when "00111000",
"0000000000000000000000000000000000000000000" when "00111001",
"0000000000000000000000000000000000000000000" when "00111010",
"0000000000000000000000000000000000000000000" when "00111011",
"0000000000000000000000000000000000000000000" when "00111100",
"0000000000000000000000000000000000000000000" when "00111101",
"0000000000000000000000000000000000000000000" when "00111110",
"0000000000000000000000000000000000000000000" when "00111111",
"0000000000000000000000000000000000000000000" when "01000000",
"0000000000000000000000000000000000000000000" when "01000001",
"0000000000000000000000000000000000000000000" when "01000010",
"0000000000000000000000000000000000000000000" when "01000011",
"0000000000000000000000000000000000000000000" when "01000100",
"0000000000000000000000000000000000000000000" when "01000101",
"0000000000000000000000000000000000000000000" when "01000110",
"0000000000000000000000000000000000000000000" when "01000111",
"0000000000000000000000000000000000000000000" when "01001000",
"0000000000000000000000000000000000000000000" when "01001001",
"0000000000000000000000000000000000000000000" when "01001010",
"0000000000000000000000000000000000000000000" when "01001011",
"0000000000000000000000000000000000000000000" when "01001100",
"0000000000000000000000000000000000000000000" when "01001101",
"0000000000000000000000000000000000000000000" when "01001110",
"0000000000000000000000000000000000000000000" when "01001111",
"0000000000000000000000000000000000000000000" when "01010000",
"0000000000000000000000000000000000000000000" when "01010001",
"0000000000000000000000000000000000000000000" when "01010010",
"0000000000000000000000000000000000000000000" when "01010011",
"0000000000000000000000000000000000000000001" when "01010100",
"0000000000000000000000000000000000000000001" when "01010101",
"0000000000000000000000000000000000000000001" when "01010110",
"0000000000000000000000000000000000000000001" when "01010111",
"0000000000000000000000000000000000000000001" when "01011000",
"0000000000000000000000000000000000000000001" when "01011001",
"0000000000000000000000000000000000000000001" when "01011010",
"0000000000000000000000000000000000000000010" when "01011011",
"0000000000000000000000000000000000000000010" when "01011100",
"0000000000000000000000000000000000000000010" when "01011101",
"0000000000000000000000000000000000000000011" when "01011110",
"0000000000000000000000000000000000000000011" when "01011111",
"0000000000000000000000000000000000000000100" when "01100000",
"0000000000000000000000000000000000000000101" when "01100001",
"0000000000000000000000000000000000000000110" when "01100010",
"0000000000000000000000000000000000000000111" when "01100011",
"0000000000000000000000000000000000000001000" when "01100100",
"0000000000000000000000000000000000000001010" when "01100101",
"0000000000000000000000000000000000000001011" when "01100110",
"0000000000000000000000000000000000000001101" when "01100111",
"0000000000000000000000000000000000000010000" when "01101000",
"0000000000000000000000000000000000000010011" when "01101001",
"0000000000000000000000000000000000000010111" when "01101010",
"0000000000000000000000000000000000000011011" when "01101011",
"0000000000000000000000000000000000000100000" when "01101100",
"0000000000000000000000000000000000000100110" when "01101101",
"0000000000000000000000000000000000000101101" when "01101110",
"0000000000000000000000000000000000000110110" when "01101111",
"0000000000000000000000000000000000001000000" when "01110000",
"0000000000000000000000000000000000001001100" when "01110001",
"0000000000000000000000000000000000001011011" when "01110010",
"0000000000000000000000000000000000001101100" when "01110011",
"0000000000000000000000000000000000010000000" when "01110100",
"0000000000000000000000000000000000010011000" when "01110101",
"0000000000000000000000000000000000010110101" when "01110110",
"0000000000000000000000000000000000011010111" when "01110111",
"0000000000000000000000000000000000100000000" when "01111000",
"0000000000000000000000000000000000100110000" when "01111001",
"0000000000000000000000000000000000101101010" when "01111010",
"0000000000000000000000000000000000110101111" when "01111011",
"0000000000000000000000000000000001000000000" when "01111100",
"0000000000000000000000000000000001001100001" when "01111101",
"0000000000000000000000000000000001011010100" when "01111110",
"0000000000000000000000000000000001101011101" when "01111111",
"0000000000000000000000000000000010000000000" when "10000000",
"0000000000000000000000000000000010011000010" when "10000001",
"0000000000000000000000000000000010110101000" when "10000010",
"0000000000000000000000000000000011010111010" when "10000011",
"0000000000000000000000000000000100000000000" when "10000100",
"0000000000000000000000000000000100110000011" when "10000101",
"0000000000000000000000000000000101101010000" when "10000110",
"0000000000000000000000000000000110101110100" when "10000111",
"0000000000000000000000000000001000000000000" when "10001000",
"0000000000000000000000000000001001100000111" when "10001001",
"0000000000000000000000000000001011010100001" when "10001010",
"0000000000000000000000000000001101011101001" when "10001011",
"0000000000000000000000000000010000000000000" when "10001100",
"0000000000000000000000000000010011000001110" when "10001101",
"0000000000000000000000000000010110101000001" when "10001110",
"0000000000000000000000000000011010111010001" when "10001111",
"0000000000000000000000000000100000000000000" when "10010000",
"0000000000000000000000000000100110000011100" when "10010001",
"0000000000000000000000000000101101010000010" when "10010010",
"0000000000000000000000000000110101110100010" when "10010011",
"0000000000000000000000000001000000000000000" when "10010100",
"0000000000000000000000000001001100000111000" when "10010101",
"0000000000000000000000000001011010100000101" when "10010110",
"0000000000000000000000000001101011101000101" when "10010111",
"0000000000000000000000000010000000000000000" when "10011000",
"0000000000000000000000000010011000001110000" when "10011001",
"0000000000000000000000000010110101000001010" when "10011010",
"0000000000000000000000000011010111010001010" when "10011011",
"0000000000000000000000000100000000000000000" when "10011100",
"0000000000000000000000000100110000011100000" when "10011101",
"0000000000000000000000000101101010000010100" when "10011110",
"0000000000000000000000000110101110100010100" when "10011111",
"0000000000000000000000001000000000000000000" when "10100000",
"0000000000000000000000001001100000111000000" when "10100001",
"0000000000000000000000001011010100000101000" when "10100010",
"0000000000000000000000001101011101000101000" when "10100011",
"0000000000000000000000010000000000000000000" when "10100100",
"0000000000000000000000010011000001101111111" when "10100101",
"0000000000000000000000010110101000001001111" when "10100110",
"0000000000000000000000011010111010001010000" when "10100111",
"0000000000000000000000100000000000000000000" when "10101000",
"0000000000000000000000100110000011011111110" when "10101001",
"0000000000000000000000101101010000010011110" when "10101010",
"0000000000000000000000110101110100010100000" when "10101011",
"0000000000000000000001000000000000000000000" when "10101100",
"0000000000000000000001001100000110111111100" when "10101101",
"0000000000000000000001011010100000100111101" when "10101110",
"0000000000000000000001101011101000100111111" when "10101111",
"0000000000000000000010000000000000000000000" when "10110000",
"0000000000000000000010011000001101111111000" when "10110001",
"0000000000000000000010110101000001001111010" when "10110010",
"0000000000000000000011010111010001001111110" when "10110011",
"0000000000000000000100000000000000000000000" when "10110100",
"0000000000000000000100110000011011111110000" when "10110101",
"0000000000000000000101101010000010011110011" when "10110110",
"0000000000000000000110101110100010011111101" when "10110111",
"0000000000000000001000000000000000000000000" when "10111000",
"0000000000000000001001100000110111111100001" when "10111001",
"0000000000000000001011010100000100111100110" when "10111010",
"0000000000000000001101011101000100111111010" when "10111011",
"0000000000000000010000000000000000000000000" when "10111100",
“0000000000000000010011000001101111111000001" when "10111101",
"0000000000000000010110101000001001111001101" when "10111110",
"0000000000000000011010111010001001111110011" when "10111111",
"0000000000000000100000000000000000000000000" when "11000000",
"0000000000000000100110000011011111110000011" when "11000001",
"0000000000000000101101010000010011110011010" when "11000010",
"0000000000000000110101110100010011111100110" when "11000011",
"0000000000000001000000000000000000000000000" when "11000100",
"0000000000000001001100000110111111100000101" when "11000101",
"0000000000000001011010100000100111100110011" when "11000110",
"0000000000000001101011101000100111111001101" when "11000111",
"0000000000000010000000000000000000000000000" when "11001000",
"0000000000000010011000001101111111000001010" when "11001001",
"0000000000000010110101000001001111001100110" when "11001010",
"0000000000000011010111010001001111110011001" when "11001011",
"0000000000000100000000000000000000000000000" when "11001100",
"0000000000000100110000011011111110000010100" when "11001101",
"0000000000000101101010000010011110011001101" when "11001110",
"0000000000000110101110100010011111100110011" when "11001111",
"0000000000001000000000000000000000000000000" when "11010000",
"0000000000001001100000110111111100000101001" when "11010001",
"0000000000001011010100000100111100110011010" when "11010010",
"0000000000001101011101000100111111001100101" when "11010011",
"0000000000010000000000000000000000000000000" when "11010100",
"0000000000010011000001101111111000001010010" when "11010101",
"0000000000010110101000001001111001100110100" when "11010110",
"0000000000011010111010001001111110011001011" when "11010111",
"0000000000100000000000000000000000000000000" when "11011000",
"0000000000100110000011011111110000010100011" when "11011001",
      "0000000000101101010000010011110011001101000" when "11011010",
      "0000000000110101110100010011111100110010110" when "11011011",
      "0000000001000000000000000000000000000000000" when "11011100",
      "0000000001001100000110111111100000101000110" when "11011101",
      "0000000001011010100000100111100110011010000" when "11011110",
      "0000000001101011101000100111111001100101011" when "11011111",
      "0000000010000000000000000000000000000000000" when "11100000",
      "0000000010011000001101111111000001010001100" when "11100001",
      "0000000010110101000001001111001100110100000" when "11100010",
      "0000000011010111010001001111110011001010111" when "11100011",
      "0000000100000000000000000000000000000000000" when "11100100",
      "0000000100110000011011111110000010100011001" when "11100101",
      "0000000101101010000010011110011001101000000" when "11100110",
      "0000000110101110100010011111100110010101101" when "11100111",
      "0000001000000000000000000000000000000000000" when "11101000",
      "0000001001100000110111111100000101000110010" when "11101001",
      "0000001011010100000100111100110011001111111" when "11101010",
      "0000001101011101000100111111001100101011011" when "11101011",
      "0000010000000000000000000000000000000000000" when "11101100",
      "0000010011000001101111111000001010001100011" when "11101101",
      "0000010110101000001001111001100110011111110" when "11101110",
      "0000011010111010001001111110011001010110110" when "11101111",
      "0000100000000000000000000000000000000000000" when "11110000",
      "0000100110000011011111110000010100011000111" when "11110001",
      "0000101101010000010011110011001100111111101" when "11110010",
      "0000110101110100010011111100110010101101011" when "11110011",
      "0001000000000000000000000000000000000000000" when "11110100",
      "0001001100000110111111100000101000110001110" when "11110101",
      "0001011010100000100111100110011001111111010" when "11110110",
      "0001101011101000100111111001100101011010111" when "11110111",
      "0010000000000000000000000000000000000000000" when "11111000",
      "0010011000001101111111000001010001100011011" when "11111001",
      "0010110101000001001111001100110011111110100" when "11111010",
      "0011010111010001001111110011001010110101101" when "11111011",
      "0100000000000000000000000000000000000000000" when "11111100",
      "0100110000011011111110000010100011000110111" when "11111101",
      "0101101010000010011110011001100111111100111" when "11111110",
      "0110101110100010011111100110010101101011010" when "11111111",
      "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" when others;

      x_rescal8 <= gain * x_invquant;
      x_rescal <= x_rescal8(52 downto 3);

end apply_scalefactors;
```

C.10 quantadjust.vhd

```
----------------------------------------------------------------------
-- Quantisation Adjustment -------------------------------------------
----------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity quantisation_adjustment is
      port(
      sign_bit: in STD_LOGIC;
      x_rescal: in STD_LOGIC_VECTOR(49 downto 0);
      x_adjust: out STD_LOGIC_VECTOR(46 downto 0)
      );
end quantisation_adjustment;

architecture quantisation_adjustment of quantisation_adjustment is

-- Signal declarations
signal x_div8: STD_LOGIC_VECTOR(46 downto 0);
signal twos_xdiv8: STD_LOGIC_VECTOR(46 downto 0);

begin

      -- Concurrent assignments
      x_div8 <= x_rescal(49 downto 3);

      twos_xdiv8 <= not x_div8 + 1;

      with sign_bit select
      x_adjust  <=  x_div8  when  '0',  twos_xdiv8  when  '1',  (others  =>
'0') when others;

end quantisation_adjustment;
```

C.11 testbench.vhd

```
----------------------------------------------------------------------
-- testbench ---------------------------------------------------------
----------------------------------------------------------------------

library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;

entity Test_bench is
      port (
      clk: in STD_LOGIC;
      reset: in STD_LOGIC
      );
end Test_bench;

architecture TB_ARCHITECTURE of Test_bench is

component bitstream_top is
      port (
      clk: in STD_LOGIC;
      reset: in STD_LOGIC;
      bits_in: in STD_LOGIC;
      clock_48: out STD_LOGIC
      );
end component;

component bitstream_source
      port (
      clk: in STD_LOGIC;
      reset: in STD_LOGIC;
      bit_out: out STD_LOGIC
      );
end component;

signal bitstream : std_logic;
signal clock_48khz: STD_LOGIC;

begin

      -- Unit Under Test port map
      Decoder : bitstream_top port map (
            clk => clk,
            reset => reset,
            bits_in => bitstream,
            clock_48 => clock_48khz
      );

      Source: bitstream_source port map (
            clk => clock_48khz,
            reset => reset,
            bit_out => bitstream
      );

end TB_ARCHITECTURE;
```

C.12 bitstream_source.vhd

```
----------------------------------------------------------------------
-- bitstream_source --------------------------------------------------
----------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity bitstream_source is
      port (
      clk: in STD_LOGIC;
      reset: in STD_LOGIC;
      bit_out: out STD_LOGIC
      );
end bitstream_source;

architecture bitstream_source of bitstream_source is

constant test_stream: STD_LOGIC_VECTOR(687 downto 0) :=
x"41444946401F40000205009880040003C3B456E636F6465642062792041542654204C
61626F7261746F726965732C20474D5420467269204465632032312031343A33303A32
392032303031 0A9F0200AB00C85483EA1B";

signal test_stream_next: STD_LOGIC_VECTOR(687 downto 0);

begin

      process(clk,reset)
      begin
            if (reset = '0') then
```

```
                  test_stream_next <= test_stream;
                  bit_out <= test_stream(687);
            else
                  if(clk'event and clk = '1') then
                        test_stream_next(687 downto 1)
                              <= test_stream_next(686 downto 0);
                        bit_out <= test_stream_next(686);
                  end if;
            end if;
      end process;
end bitstream_source;
```

APPENDIX D. XSV BOARD PINOUT FOR TESTING

```
#PINLOCK_BEGIN

#Sat Oct 12 13:45:29 2002

#File:   pinout.ucf

#Author: Ryan Linneman

#Disable SRAM
NET "SRAM_CE_Left"   LOC =  "p186";
NET "SRAM_CE_Right"  LOC =  "p109";

#Push buttons
NET "x_quant<3>"     LOC =  "P185";
NET "x_quant<2>"     LOC =  "P176";
NET "x_quant<1>"     LOC =  "P175";
NET "x_quant<0>"     LOC =  "P174";

#DIP Switches
NET "scalefactor<7>"  LOC =  "P140";
NET "scalefactor<6>"  LOC =  "P142";
NET "scalefactor<5>"  LOC =  "P146";
NET "scalefactor<4>"  LOC =  "P149";
NET "scalefactor<3>"  LOC =  "P153";
NET "scalefactor<2>"  LOC =  "P155";
NET "scalefactor<1>"  LOC =  "P159";
NET "scalefactor<0>"  LOC =  "P161";

#Expansion Header Left
NET "x_rescal<45>"   LOC =  "P199";
NET "x_rescal<44>"   LOC =  "P195";
NET "x_rescal<43>"   LOC =  "P194";
NET "x_rescal<42>"   LOC =  "P193";
NET "x_rescal<41>"   LOC =  "P192";
NET "x_rescal<40>"   LOC =  "P191";
NET "x_rescal<39>"   LOC =  "P189";
NET "x_rescal<38>"   LOC =  "P188";
NET "x_rescal<37>"   LOC =  "P187";
```

```
#Expansion Header Right
NET "x_rescal<36>"   LOC =  "P108";
NET "x_rescal<35>"   LOC =  "P107";
NET "x_rescal<34>"   LOC =  "P103";
NET "x_rescal<33>"   LOC =  "P102";
NET "x_rescal<32>"   LOC =  "P101";
NET "x_rescal<31>"   LOC =  "P100";
NET "x_rescal<30>"   LOC =  "P99";
NET "x_rescal<29>"   LOC =  "P97";
NET "x_rescal<28>"   LOC =  "P96";
NET "x_rescal<27>"   LOC =  "P95";
NET "x_rescal<26>"   LOC =  "P94";
NET "x_rescal<25>"   LOC =  "P93";
NET "x_rescal<24>"   LOC =  "P87";
NET "x_rescal<23>"   LOC =  "P86";
NET "x_rescal<22>"   LOC =  "P85";
NET "x_rescal<21>"   LOC =  "P84";
NET "x_rescal<20>"   LOC =  "P82";
NET "x_rescal<19>"   LOC =  "P81";
NET "x_rescal<18>"   LOC =  "P80";
NET "x_rescal<17>"   LOC =  "P79";
NET "x_rescal<16>"   LOC =  "P78";
NET "x_rescal<15>"   LOC =  "P74";
NET "x_rescal<14>"   LOC =  "P73";
NET "x_rescal<13>"   LOC =  "P72";
NET "x_rescal<12>"   LOC =  "P71";
NET "x_rescal<11>"   LOC =  "P70";
NET "x_rescal<10>"   LOC =  "P68";
NET "x_rescal<9>"   LOC =  "P67";
NET "x_rescal<8>"   LOC =  "P66";
NET "x_rescal<7>"   LOC =  "P65";
NET "x_rescal<6>"   LOC =  "P64";
NET "x_rescal<5>"   LOC =  "P63";
NET "x_rescal<4>"   LOC =  "P57";
NET "x_rescal<3>"   LOC =  "P56";
NET "x_rescal<2>"   LOC =  "P55";
NET "x_rescal<1>"   LOC =  "P54";
NET "x_rescal<0>"   LOC =  "P53";

#PINLOCK_END
```