



TÉCNICO
LISBOA

MPEG-1/2 audio decoder using a System-on-Chip with a RISC-V processor

Henrique Alves Gonçalves

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Prof. Doutor José João Henriques Teixeira de Sousa

Examination Committee

Chairperson: Prof. Doutor Pedro Filipe Zeferino Aidos Tomás

Supervisor: Prof. Doutor José João Henriques Teixeira de Sousa

Member of the Committee: Prof. Doutor Marcelino Bicho dos Santos

December 2024

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I want to express my appreciation and gratitude for my family's unconditional support and motivation: my parents, grandparents, uncles, cousins, and brother. I also want to recognize my friends, Ricardo and Cavaco, for the incessant discussions and companionship.

A special thanks to the professors who assisted through my studies and to teacher Lúcia, who encouraged me to pursue and continue. To my supervisor, Professor José Teixeira de Sousa, for the valuable guidance and advice and the IOBundle team for the essential help during the development of the project.

Resumo

Este trabalho consiste da elaboração de um sistema para decodificar áudio de acordo com o padrão de compressão MPEG-1/2. O formato é amplamente utilizado na indústria de dispositivos multimídia ou interfaces para som, tipicamente em infraestrutura dedicada a processamento de sinal digital ou sistemas digitais e micro-controladores de referência como processadores x86 e ARM. O objetivo principal do estudo foi uma implementação do decodificador em um sistema embebido com uma unidade central de processamento do tipo RISC-V. Este tipo de especificação simplificada disponibiliza extensões criadas para diferentes domínios de computação e diferencia-se pela adoção de licença de código aberto. A estrutura do sistema é baseada na plataforma IOB-SoC, implementado em linguagem de descrição de *hardware* Verilog. O *software* tem como recurso a biblioteca de código aberto *LibMAD*, que é integrada na arquitetura base do sistema. No âmbito de prototipagem em suporte FPGA foram realizados testes para estimar o tempo de execução computacional do algoritmo, revelando uma utilização reduzida de componentes da estrutura do dispositivo. O resultado da implementação é o suficiente para conseguir o objetivo de descompressão e integridade de dados com a arquitetura PicoRV e processamento em tempo-real com as unidades DarkRV e VexRV.

Palavras-chave: Licença de código aberto, MPEG, Decodificador de áudio, Sistema embebido, RISC-V, Matriz de Portas Programável em Campo

Abstract

This work consists of developing a system for audio decoding according to the MPEG 1/2 compression standard. The format is widely used in the multimedia device industry and sound interfaces, typically in infrastructure dedicated to digital signal processing or digital systems and reference microcontrollers such as x86 and ARM processors. The study's primary objective was implementing the decoder in an embedded system with a central processing unit with RISC-V ISA. This type of simplified specification provides extensions created for different computing domains and is distinguished by adopting an open-source license. The system's structure is based on the IOB-SoC platform, implemented in the Verilog hardware description language. The software integrates the open-source library *LibMAD* into the system's base architecture. In the context of FPGA-supported prototyping, tests were carried out to estimate the computational execution time of the algorithm, revealing minimal usage of components in the device structure. The implementation result is sufficient to meet the objective of successful decompression and data integrity with the PicoRV and real-time processing with the DarkRV and VexRV processors.

Keywords: Open-source, MPEG, Audio decoder, System-on-Chip, RISC-V, Field-programmable gate array

Contents

Acknowledgments	v
Resumo	vii
Abstract	ix
List of Tables	xiii
List of Figures	xv
List of Acronyms	xvii
1 Introduction	1
1.1 MPEG Digital Audio Coding Standard	1
1.2 Motivation	1
1.3 Topic overview	2
1.4 Definition of objectives and deliverables	3
1.5 Outline	3
2 Background	5
2.1 The fundamentals of signal processing	5
2.2 Principles of digital audio compression	7
2.3 ISO/ IEC 11172 & 13818	8
2.3.1 Compressed audio characteristics and applications	8
2.3.2 Bitstream coding format	9
2.3.3 Encoding process	10
2.3.4 Decoding process	11
a) Bitstream unpacking	11
b) Frequency-to-time inverse mapping	11
c) Sub-band synthesis	12
2.4 Development technologies	13
2.4.1 Field Programmable Gate Array	13
2.4.2 System-on-chip	14
2.4.3 The IOB-SoC project template	14
a) Project workflow	15
b) SoC architecture	16
c) Software	17
d) Peripherals	18
e) Primary parameters and SoC configuration	21
2.4.4 RISC-V Instruction Set Architecture	23
2.5 Related work	25

3	System design	27
3.1	Hardware structure	28
3.1.1	Embedded system components	29
a)	Central processing unit	29
b)	AXI-Stream bus protocol	30
c)	Clock timer	30
3.2	Software architecture	32
3.2.1	LibMAD - MPEG Audio Decoder Library	33
3.2.2	Application runtime analysis	33
3.2.3	Fixed-point numerical representation	37
3.2.4	Procedure in the context of the IOb-MPEG-Decoder	38
a)	Integration strategies	39
b)	Firmware customisation	40
c)	Profiling	44
4	System Verification	47
4.1	Testing platform	48
4.1.1	IOb-SoC-SUT	50
4.1.2	Audio samples evaluation set	52
5	Results	53
5.1	Profiling	53
5.2	Performance metrics	55
5.3	FPGA system resources utilization	56
5.4	Evaluation on different types of audio content	56
5.5	Real-time requirements	57
6	Conclusions	59
6.1	Achievements	59
6.2	Future work	60
	References	61

List of Tables

2.1	Comparison of FPGA device resources	13
2.2	RISC-V integer register set	23
2.3	RISC-V floating-point register set	24
3.1	Arithmetic and logic operations used in the decoding & scaling algorithms	37
3.2	SoC configuration parameters for the IOb-MPEG-Decoder	43
3.3	Description of configuration options for the compilation flags	44
3.4	Configuration parameters and descriptions for profiling	45
4.1	Audio file examples and respective specifications	52
5.1	Execution time per function related to the MP2 decoding procedure (frame average) [μs] .	53
5.2	Execution time per function related to the MP3 decoding procedure (frame average) [μs] .	54
5.3	FPGA resource usage for the IOb-MPEG-Decoder	56
5.4	Execution time on average per frame for MP2 and MP3 files [μs]	57
5.5	Halting difference for decoding time per frame of MP2 and MP3 files [μs]	57
5.6	Attained speedup and required values for MP2 and MP3 files	58

List of Figures

2.1	Frequency spectrum representation of a quantised signal $X_s(f)$	5
2.2	Spectral masking effects on audibility threshold Source: Psychoacoustic Models for Perceptual Audio Coding [8]	7
2.3	MPEG audio frame header structure	9
2.4	MPEG bitstream	9
2.5	MPEG layer II frame scheme	9
2.6	MPEG layer III frame scheme	9
2.7	MPEG audio encoder	10
2.8	MPEG perceptual decoder system	11
2.9	High-level block diagram of the IObundle-SoC Source: PEPC Online Workshop Presentation 25/01/2022	14
2.10	System-on-chip hardware structure	16
2.11	Block diagram of the AXI hardware component	19
2.12	Block diagram of the cache memory IP core	20
2.13	Block diagram of the UART hardware component	21
3.1	Block diagram of the IOB-MPEG-Decoder	28
3.2	Block diagram of the AXI-Stream peripheral	30
3.3	Block diagram of the timer peripheral	30
3.4	Software implementation flow diagram	32
3.5	MPEG layer II decoder example with functions call graph	35
3.6	MPEG layer III decoder example with functions call graph	36
3.7	Fixed-point binary number representation	37
3.8	High-level SoC diagram with communication modules	38
3.9	Decoder firmware pseudocode	42
4.1	Test environment infrastructure diagram	48
4.2	IoB-SoC Tester & MPEG Audio Decoder diagram as unit under test	50
4.3	Tester firmware pseudocode	51

List of Acronyms

AAC	Advanced Audio Coding
ACK	Acknowledgement
ADC	Analog-to-digital converter
ALU	Arithmetic logic unit
API	Application programmable interface
ASCII	American Standard Code for Information Inter- change
ASIC	Application-specific integrated circuit
AXI	Advanced eXtensible Interface
BPS	Bits per second
BRAM	Block Random Access Memory
CBR	Constant bitrate
CD	Compact disk
CGRA	Coarse-Grained Reconfigurable Arrays
CI/ CD	Continuous Integration and Continuous Deliv- ery
CLB	Configurable logic block
CLI	Command line interface
CPI	Cycles per instruction
CPU	Central processing unit
CRC	Cyclic redundancy check
DAB	Digital audio broadcasting
DCT	Discrete Cosine Transform
DDR RAM	Double data rate random-access memory
DMA	Direct memory access
DSP	Digital signal processing
ENQ	Enquiry
EOT	End-of-transmission
FFT	Fast Fourier Transform
FF	Flip-flop
FIFO	First in first out
FPGA	Field programmable gate array
FRX	File reception request
FSDCT	Fast standard discrete cosine transform
FTX	File transmission request
HAL	Hardware abstraction layer

HDL	Hardware description language
Hz	Hertz
I2S	Inter-IC Sound
IEC	International Electrotechnical Commission
IMDCT	Inverse modified discrete cosine transform
IP	Intellectual property
ISA	Instruction set architecture
ISO	International Organization for Standardization
Kbps	Kilobit per second
LSF	Low Sampling Frequency
LUT	Lookup table
MAD	MPEG audio decoder
MCU	Microcontroller unit
MP1, MP2 & MP3	MPEG layer I/ II and III
MPEG	Moving Picture Experts Group
Mbps	Megabit per second
OS	Operating system
PCM	Pulse code modulation
PL & PS	Programmable logic and programmable system
PMod	Peripheral Module interface
RISC	Reduced instruction set computer
RMS	Root mean square
ROM	Read-only memory
RS232	Recommended Standard 232 (Computer serial interface)
RTL	Register transfer level
RTOS	Real-time operating system
RV32IMA	RISC-V ISA 32-bit architecture with integer, multiplication/ division and atomic instructions extensions
SRAM	Static random-access memory
SSH	Secure shell
SUT	System under test
SoC	System-on-chip
UART	Universal asynchronous receiver-transmitter
UIMSBF	Unsigned integer most significant bit first
VBR	Variable bitrate
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
WAV	Waveform Audio File Format

Chapter 1

Introduction

1.1 MPEG Digital Audio Coding Standard

The Moving Picture Experts Group was founded in 1988 by the International Standards Organization to develop coding standards for multimedia, including digital audio and video formats for compression and data transport. The first version of the codec standard, MPEG-1 [1], identified by ISO/ IEC 11172-3, was completed in 1992 and describes three compression layers. A more advanced version, MPEG-2, identified by ISO/ IEC 13818 [2], followed, further refining these standards and enabling improved efficiency and performance across various applications, including broadcasting and DVDs.

The layers present increasing complexity and higher compression rates, making the last layer the most efficient. This, combined with a broader, more flexible coding format than the anterior layers and a range of available configuration settings related to signal processing attributes, such as bit rate and sample frequencies, leads to more coverage regarding audio data content and its applications, one of the reasons why MP3 became extremely popular for digital audio files.

Higher bit rates offer increased audio quality, while lower bit rates are used to save space, albeit at the cost of audio quality. This definition made it possible to maintain high-quality standards relative to the audio experience and support fidelity retention with less information to represent the uncompressed data.

1.2 Motivation

According to the current state of the intellectual property ecosystem, open-source libraries are a helpful resource for developing semiconductor hardware, which could potentially promote more regular competition and may increase accessibility to a more significant number of companies able to research, create, and test new forms of microelectronic devices, resulting in an innovation and adaptation convergence to redefine technological standards and functionalities for computing systems.

In today's rapidly evolving technological landscape, there is a discernible shift towards hybrid approaches in computing system development from lower-end systems, such as microcontrollers, to more complex computing machines, for example, with machine learning or hardware acceleration dedicated modules. Such advancements transcend conventional boundaries, permeating diverse domains and driving transformative change across various use cases. This perpetual evolution necessitates agile methodologies and adaptable frameworks to accommodate the ever-changing demands of contemporary applications.

This approach challenges proprietary models involving integrated circuit design, design automation tools, composable processor extensions and incremental instruction set development approaches for microprocessor architectures, encouraging broader adoption of innovative practices in the industry.

Embedded system design was once a matter of manually transposing an intention expressed using mnemonics into machine code (usually represented using hexadecimal or binary values) and storing those values in a persistent storage device readable by a microprocessor. This technology is integral to numerous modern devices, consisting of processing units, memory, and input/ output peripherals. It is developed to perform dedicated functions within larger systems, materializing the convergence of software and hardware.

The SoC paradigm embodies the composition of diverse intellectual property blocks into a cohesive design, culminating in a versatile platform that can orchestrate intricate functionalities for data processing and operating systems environments. Tailored to specific applications, it leverages targeted optimizations and to fulfil specialized requirements with precision and efficiency. By aligning the system capabilities with application-specific demands, developers can harness the full capability of hardware resources, thus improving performance and efficacy.

With the development of processor design and semiconductor technology, RISC's digital signal processing capability has approached the DSP level. Therefore, it became a reality to implement MP3 decoding in real-time on a single RISC core [3].

1.3 Topic overview

This thesis report defines the proposed implementation of an MPEG-1/2 audio decoder in a system-on-chip composed of a central processing unit which implements the 32-bit RISC-V instruction set architecture (ISA), internal and external memory and peripheral interface components used for data transmission.

Audio data compression techniques allow audio engineers to save memory space by compressing sensitive and perceptually irrelevant signal data and quantizing the difference between a masking amplitude envelope and the actual sound level. Sound decoders have become increasingly popular and economically available from a production cost standpoint, and they are included in various audiovisual consumer electronics products.

The subsequent topics and chapters encapsulate the core aspects of the dissertation about the work on developing the decoder submodule implemented in the system-on-chip platform developed by *IOBundle, Lda* [4]. In the scope of the work, an open-source instruction set architecture (ISA) based on RISC-V consolidates the hardware mechanism and scheme for the implementation unit as a software attachment that can be moulded for the referred architecture. For audio applications in a broad spectrum of consumer electronics, these encoding types and decoding IP cores became a standard solution for multimedia processing techniques.

The role of the Field-Programmable Gate Array (FPGA) is to be the deployment platform of the SoC as a whole, with software and hardware definitions, utilizing the device fabric, constituted by configurable and reprogrammable components and memory. Implementation is conducted to adequately port the decoder application to the SoC, create a testing environment, measure execution time on software elements, and evaluate the results regarding the system-under-test's properties.

1.4 Definition of objectives and deliverables

In this project, a software implementation of the MPEG-1/2 audio layers I, II and III decoder uses a RISC-V scalar unit with dedicated modules for fixed-point multiply and division instructions and a five-stage pipeline processor installed on an FPGA for design, simulation, and test purposes.

Custom and modular firmware, debugging tools, and project build automation are essential elements in the development process of embedded systems, ensuring efficient compilation, synthesis, and deployment.

For the current project, a software library is used for the decoding process, *LibMAD* (Underbit Technologies) [5]. This repository will be implemented with scalar and pipelined central processing units based on RISC-V instruction set architecture and evaluated to guarantee the correct operation of the software implemented in the context of the IOB-SoC.

The objectives of the current work address the description of the integration of the decoding process as a software element with the system-on-chip architecture template, developed by *IOBundle, Lda*, and a representation of the conduct and behaviour between the central core with its memory and interconnect properties with other components. The system is put through a set of evaluation audio samples to produce results regarding the execution time of the decompression procedure, which is used for quality requirements verification and program workload performed in the decoding procedure.

1.5 Outline

This report presents a chapter on each major topic of the work.

The second chapter discusses a theoretical background of signal processing and data compression, particularly sound modulation and analysis, in the scope of intellectual property development, according to existing solutions in the market for the corresponding computer processing unit architectures. Following the MPEG introduction and theoretical background, the report also contains a presentation of technologies such as the FPGA and its logic resources, the RISC-V open software architecture, the decoding library built in C language, and its features and functionalities that sum up the complete integration of the decoder implementation.

The third chapter contains a more detailed view of the decoding scheme, and some introduced designs will be compared to the proposed implementation. This section describes the software implementation concerning the proposed instruction set architecture, including the available interfaces and submodules.

The fourth chapter describes the simulation process and communication of the working framework and the core implementation for testing the integrated MPEG audio decoder functionality. It is followed by a study of the possible optimizations that can be applied, the test of real-time decoding and its relation to the system's resource requirements and the maintenance of audio quality facing the different forms of audio files regarding the referred layers, different audio profiles, and other codec features.

The fifth chapter presents the results regarding the central processing unit models used to run the system and evaluates the time profiling of the software components involved in the algorithm execution.

The sixth chapter provides conclusions about the research made in the project, what was accomplished and a proposal for future work related to continuous development for this thesis.

Chapter 2

Background

2.1 The fundamentals of signal processing

- **Sampling theorem** The conversion from a continuous-time signal, $x(t)$, to a discrete-time signal is achieved through a process known as sampling. This method involves the measurement of the amplitude of the signal at discrete intervals, typically uniform and periodic. The mathematical expression for the continuous-time sampled signal, $x_s(t)$, and the discrete-time representation, $x[n]$, can be written as:

$$1. \quad x_s(t) = \sum_{n=-\infty}^{\infty} x(nT_s) \cdot \delta(t - nT_s) \quad 2. \quad x[n] = x(nT_s) \quad (2.1)$$

where T_s is the sampling period, the time interval between successive samples, $\delta(t - nT_s)$ is the Dirac delta function, creating impulses at each sampling instant $t = x(nT_s)$ and n is an integer representing the index of the sample. In the frequency domain, this multiplication corresponds to a convolution between the signal's spectrum $X(f)$ and the Fourier Transform of the impulse train, which is another impulse train:

$$X_s(f) = X(f) * \left(f_s \cdot \sum_{k=-\infty}^{\infty} \delta(f - kf_s) \right) \quad (2.2)$$

This convolution results in replicas of the original spectrum $X(f)$ shifted by multiples of the sampling frequency f_s .

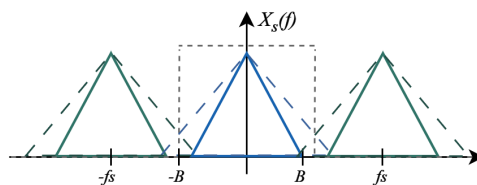


Figure 2.1: Frequency spectrum representation of a quantised signal $X_s(f)$

The Nyquist-Shannon theorem provides a criterion for the minimum sampling rate that must be used to ensure that the continuous-time signal can be fully reconstructed from its samples without any information loss. Aliasing is a phenomenon that causes different signals to become indistinguishable (or aliases of one another) when sampled. It occurs when the sampling frequency f_s is insufficient to capture the signal's frequency content.

Assuming the signal is band-limited to a maximum frequency B (i.e., $X(f) = 0$ for $|f| > B$), the components in the frequency domain due to sampling will start at $\pm f_s$ and extend from $f = \pm f_s - B$ to $f = \pm f_s + B$. To prevent overlap between the original spectrum and the transformed signal, it is required:

$$f_s - B > B \implies f_s > 2B \quad (2.3)$$

This inequality ensures that the highest frequency component of the original signal does not interfere with the lowest frequency component of the adjacent replica, thereby preventing aliasing, which can be described by the folding of frequency components beyond the Nyquist frequency ($f_{\text{Nyquist}} = f_s/2$) back into the range $[0, f_{\text{Nyquist}}]$ [6].

- **Quantization and encoding** After sampling, the continuous amplitude values must be quantised into discrete levels. In PCM, the sampled value $x[n]$ is approximated by the nearest value from a finite set of discrete amplitude levels. This step is necessary because digital systems can only represent a finite number of amplitude values due to their binary nature. It is the standard analogue audio conversion format used in digital systems. The difference between the actual and quantised values introduces quantisation error, a form of noise defined as the difference between the original sample and the quantised values. Following quantisation, each quantised sample $x_q[n]$ is encoded into a binary sequence for digital representation. The encoding process assigns a unique binary code to each quantisation level. The number of bits b used in encoding determines the resolution and the number of quantisation levels L . The quantisation level set is determined by the quantisation resolution, which depends on the number of bits b used to represent each sample. The total number of discrete levels is given by $L = 2^b$.

- **Fourier transform and its applications** The Fourier Transform is a fundamental mathematical tool used to analyse and process signals in the frequency domain. It decomposes a time-domain signal into its constituent frequency components, providing a frequency spectrum that describes the signal's content [7]. For a continuous-time signal $x(t)$, the Fourier Transform $X(f)$ is defined as:

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft} dt \quad (2.4)$$

This transformation converts the time-domain signal into a continuous function of frequency, revealing the amplitude and phase of each frequency component present in $x(t)$. For discrete-time signals obtained through sampling, the Discrete-time Fourier Transform (DTFT) provides a continuous frequency spectrum for a sampled signal $x[n]$. The DFT is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}kn}, \quad k = 0, 1, \dots, N-1 \quad (2.5)$$

where N is the number of samples and $X[k]$ represents the frequency component at index k . In signal processing, the discrete signal transformation is often computed using the Fast Fourier Transform. This algorithm method takes advantage of the periodic and symmetric properties of the complex exponentials (roots of unity) in the DFT formula. It recursively splits the DFT of N points into two DFTs of $N/2$ points, one for the even-indexed inputs and one for the odd-indexed inputs. The algorithm reduces the complexity of computing the DFT from $O(N^2)$ to $O(N \log_2(N))$, making it much faster for larger N . This efficiency gain is crucial for applications requiring real-time or high-speed processing of large datasets, such as audio/ video processing and telecommunications.

2.3 ISO/ IEC 11172 & 13818

The MPEG standards for audio compression, specifically ISO/ IEC 11172-3 and subsequent modification, ISO/ IEC 13818-3, present a digital audio framework for efficient storage, transmission and quality requirements. These standards became essential because they laid the groundwork for widespread audio compression formats. The directives include the syntax of the coded bitstream, the definitions of the encoding and decoding process and compliance tests for assessing the accuracy of the decoder. This guarantees that, regardless of origin, any fully compliant MPEG audio decoder will be able to decode any MPEG audio bitstream with a predictable result.

Both standards employ a layered approach, with layers presenting increasing complexity and higher compression rates, making the last layer the most efficient. This, combined with a broader, more flexible coding format than the anterior layers and a range of available configuration settings related to signal processing attributes, such as bit rate and sample frequencies, leads to more coverage regarding audio data content and its applications, one of the reasons why MP3 became extremely popular for digital audio files.

The introduction of MPEG-2 has expanded the utility of these standards by including a greater amplitude of bit rate and sample frequency configurations² enhance their application in a broader range of media technology, essential for various applications like digital television and DVD. While MPEG-1 was designed to deliver high-quality audio at higher bit rates, greater than 192 Kbps³, MPEG-2 allows for lower bit rates, causing it more adaptable in constrained bandwidth scenarios and supports 32, 44.1, and 48 kHz sample frequencies, which are well-suited for various devices but primarily target is CD audio quality. This standard version supports additional options like 16 kHz, 22.05 kHz, and 24 kHz. These are particularly useful for audio formats requiring less detail, occupying less storage space, or demanding less transfer rate throughput in data streaming services.

2.3.1 Compressed audio characteristics and applications

From communication to audiovisual consumer electronics and even with the emergence of the World Wide Web, along with the adoption of audio interfaces in consumer electronics of all application types, the decreased compressed data size becomes a necessity to encode audio information as memory capacity on a chip is reduced. Maintaining this homogeneity and information integrity between the sender and receiver is essential for real-time audio and visual multimedia transmission to secure service quality at a reasonable bandwidth.

By integrating the mentioned functionalities using platform-based approaches, companies can develop dedicated hardware and/ or software to implement coding algorithms with multiple devices and environments. Low power consumption and the area associated with the hardware appliances are two main features of digital and logic systems design and deployment in small-scale designs. Audio decoding solutions as a software component on a system-on-chip offer a good relationship between application abstraction and hardware resources or required memory space.

Compressed audio standards have also paved the way for advancements in mobile and streaming applications. As streaming platforms became more popular, efficient compression algorithms that balance audio quality and data size became critical. MPEG-2's flexibility in handling lower bit rates makes it especially useful in limited bandwidth, such as mobile networks or online streaming services. Moreover, the compatibility of these standards with various playback devices—ranging from low-power embedded systems in portable devices to high-end multimedia systems illustrates their adaptability across different environments.

²Number of samples obtained in one second, over a sampling period T , thus $f_s = 1/T$. Units expressed in Hertz (Hz)

³Number of bits conveyed or processed per unit of time. Units are expressed in Kbps (kilo-bit per second)

2.3.2 Bitstream coding format

The encoded bitstream comprises a sequence of frames containing information necessary to generate output audio samples. Each frame consists of a header and a data block. The header marks the beginning of the frame and is represented by a four-byte integer, 32 bits. It includes fields such as synchronisation word, MPEG version identifier, and layer description. Additionally, it contains a bit allocation index that specifies whether the audio data is for a single or dual channel, as well as an optional Cyclic Redundancy Check (CRC) for error detection and correction, ensuring the decoders' accurate interpretation of audio frames. Further fields define the sample frequency, bitrate, and various parameters related to the audio content, such as padding and emphasis. Each frame holds audio information, scale factor information, side information related to stereo channels, and ancillary data, which may include metadata or application-specific information.

Frame	
Header	32 bit
Synchronization word	11 bit
MPEG Version identifier	2 bit
Layer description	2 bit
Protection bit	1 bit
Bit-rate index	4 bit
Sample frequency index	2 bit
Padding bit	1 bit
Private bit	1 bit
Channel mode (1/ 2 way)	2 bit
Mode extension	2 bit
Copyright	1 bit
Original	1 bit
Emphasis	2 bit

Figure 2.3: MPEG audio frame header structure

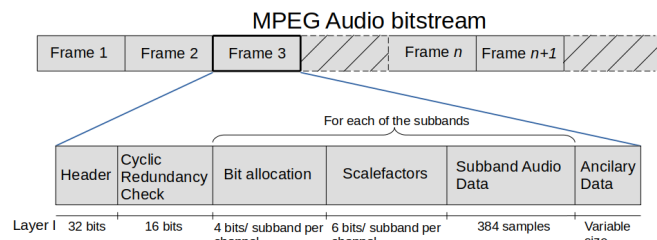


Figure 2.4: MPEG bitstream

The MPEG audio bitstream is built from several package structures with encoded information, as illustrated in Figure 2.2. Each frame incorporates a header followed by data blocks that vary in length depending on the audio layer (I, II, or III) used. Compared to MP1, layer II adds more detailed bit allocation fields (larger subbands) and scaling information.

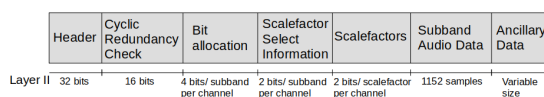


Figure 2.5: MPEG layer II frame scheme

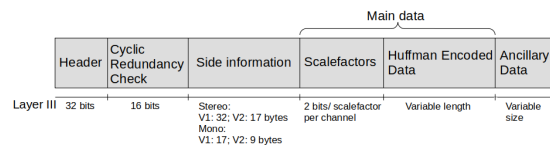


Figure 2.6: MPEG layer III frame scheme

Layer III frame structure further refines the compression process by incorporating Huffman encoded data, side information related to stereo channels, and scale factor selection. The scale factor selection allows for adaptive control over the quantisation noise. The encoder can minimise audible distortions by selecting appropriate scale factors for different frequency bands, especially in frequency ranges where the human ear is most sensitive. This perceptual coding approach leverages psychoacoustic models to allocate bits that align with human auditory perception. Each successive layer improves the compression performance at the cost of greater encoder and decoder complexity.

2.3.3 Encoding process

In the encoding process, each layer deals with different aspects of frequency resolution regarding the respective quantisation properties. An analysis filter bank retrieves the frequency information and organises it in a bank of 32 frequency sub-bands modelled by a *Quadrature Mirror Filter*. The output of each filter is downsampled by 32 (1 sample out for every 32 samples of the input), where each one contains a group of 12 samples for layer I, giving a total of 384 sub-band samples and three times 12 samples for the layer II, that corresponds to a total of 1152 sub-band samples which forms an allocation frame that represents each audio fragment encoded in conjunction.

Based on the perceptual model that employs a time-to-frequency mapping to get finer frequency resolution for an accurate calculation of the masking thresholds, the sound level information above the masking level is analysed to determine which frequencies to exclude and define the quantisation step size for the remnant ones that are to be encoded. A *fast Fourier transform* performs the spectral analysis separately for layers II and III, and dynamic bit allocation, scale-factor calculation, and selection are performed. The quantisation data is coded into a digital stream of compressed audio that defines the features that will be used to save the samples from the original signal for each frame. The layer encoding and decoding are backwards compatible, meaning the most complex can process the anterior ones.

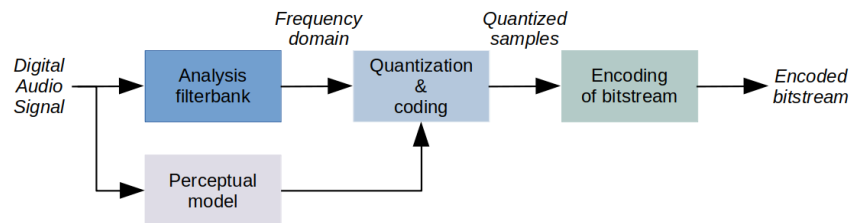


Figure 2.7: MPEG audio encoder

Similarly to layer II, the latter layer also uses the same FFT algorithm from the input to the perceptual model. It adds a *Modified Discrete Cosine Transform* window for each filtered sub-band generated after the polyphase filter bank. Considering these characteristics, the encoder generally takes the raw audio data through pulse code modulation.

Time-to-frequency mapping applies a subdivision of the signal frequency components, representing the input sample set and then appropriately allocating the available bit pool. For example, high-tonal representative signals have frequency components that change slowly in time, so it is advantageous over time-domain techniques to describe these signals with significantly less data involved in directly capturing the signal's shape as time passes. A scalar quantisation is achieved where an outer-iteration loop block is used. It is possible to mitigate the quantisation noise associated with the difference regarding the masking level from the audio signal being compensated in the scale factor for the corresponding frequency band, which is applied for rate and distortion control and finally generates a bitstream formatting and correction of possible data losses or transmission errors.

The MPEG audio coding process involves signal processing techniques and the perceptual modelling approach. The encoder achieves significant data compression by decomposing the audio signal into frequency sub-bands, applying perceptual criteria to determine bit allocation, and employing advanced quantisation methods [9]. The backward compatibility of the layers ensures that advancements in encoding techniques do not render older equipment obsolete, promoting longevity and widespread endorsement of the standard.

An MPEG audio encoder system implementation on a RISC-V processor with hardware accelerators using TwoLAME [10] software was developed along with IObundle in the scope of a master thesis [11].

2.3.4 Decoding process

In the domain of data compression, a decoder is an element that transforms compressed data into the raw format that originated it as accurately as possible.

Audio decoding aims to accurately reconstruct the original audio signal from the compressed format, with lossless or lossy information integrity between processes. This process involves inverse operations of the encoding process, where the compressed data is expanded back into a form that closely approximates the original audio waveform, ideally with indistinguishable differences to the human ear.

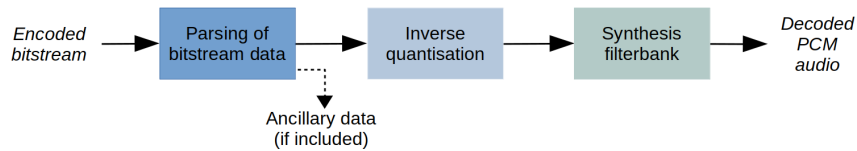


Figure 2.8: MPEG perceptual decoder system

The first step is error verification, which protects the integrity of information by examining it with a cyclic redundancy code [12]. The next step is to unpack the compressed data in the bitstream, from the header to the side and the primary information for the quantised samples that will be inversely quantised with the corresponding number of bits. Finally and analogously, a synthesis filter bank combines the resulting dequantized sub-band samples into a single or dual channel frequency domain to reproduce the output and recovered signal.

a) Bitstream unpacking

At first, the decoder should synchronize the bitstream at the beginning of the frame for decoding. Then, the side information of the MPEG audio basic channel is extracted. This information is utilized to dequantize and synthesize the subband samples. According to the channel information and dequantized subband samples, the multi-channel process can reconstruct another three channels (left, right and centre). After each channel's subband samples are ready, the synthesis filter bank can be started to generate the PCM samples.

The decoder performs entropy decoding, then reconstructs the quantized subband values and transforms subband values into a time-domain audio signal. Scale factors are used during encoding to adjust the quantisation step sizes for different frequency bands and are extracted for inverse quantisation. The decoder reads the bitstream and maps the variable-length codes to the quantised values using the code tables defined in the MPEG standard.

b) Frequency-to-time inverse mapping

Frequency-to-time inverse mapping converts a signal from the frequency to the time domain. An inverse transform, such as the Inverse Modified Discrete Cosine Transform (IMDCT), converts the frequency-domain coefficients back to time-domain samples. The inverse transform is defined as:

$$x[n] = \sum_{k=0}^{N-1} X[k] \cos \left[\frac{\pi}{N} \left(n + \frac{1}{2} + \frac{N}{2} \right) \left(k + \frac{1}{2} \right) \right], \quad n = 0, 1, \dots, 2N - 1 \quad (2.6)$$

Where $X[k]$ represents the DCT coefficients for the k -th frequency component, N is the block size of the input data, the cosine term that describes the oscillating frequency components contribution for each value of n and k , ensuring appropriate frequencies relationships are combined for each output value, and $x[n]$ are the reconstructed time-domain samples.

In audio codecs like MP3, the IMDCT is essential in the reassembly of audio signals after processing in the frequency domain. It operates by overlapping and adding subsequent blocks, which reduces interferences at the boundaries between blocks and ensures smooth audio reconstruction. The overlapping approach allows smoother transitions between successive audio frames, minimizing discontinuities and avoiding audible distortions. It provides a high-fidelity reconstruction and is highly effective for perceptual audio coding.

After synchronizing with this sequence, the decoder determines other relevant information, such as the sampling rate of uncompressed data and the data rate of the compressed data stream. The decoder then uses this information to decode the other frames. The coding tables translate the variable length-coded symbols in the inverse quantisation phase. These symbols would have been generated by the encoder based on the frequency of occurrence of each symbol. The primary goal of Huffman coding is to minimize the average code length, thus reducing the overall bit rate of the encoded data. The average code length, L , can be represented as:

$$L = \sum_{k=0}^{K-1} p_k l_k \quad (2.7)$$

Where L is the average code length represents the expected number of bits required to encode a symbol, p_k is the probability of occurrence of the k -th symbol. This value indicates how often a particular symbol appears in the data set. The length of the code, l_k , is assigned to the k -th symbol. It represents the number of bits used to encode that symbol and k the number of unique symbols in the data set [13].

c) Sub-band synthesis

In sub-band synthesis, the original signal is split into several frequency bands using a series of filters. Each band is then processed independently, including compression, modification, or other forms of manipulation. After processing, these bands are recombined and synthesised) to reconstruct the signal. A synthesis filter bank reconstructs the audio signal by combining the time-domain samples from each subband to generate the final time-domain audio signal.

In the synthesis/ filterbank stage, two steps are performed. In the first step, groups of 32 subband samples provided by the initial decoding phase are converted to 64 entry arrays using a discrete cosine transform. There are 36 such groups, and each sample in a subband represents the amplitude for a particular frequency. At any point, the synthesis/ filterbank phase keeps a set of sixteen 64-entry arrays in a rotating window fashion. In the second step, the 64-entry arrays are windowed using a set of 512 coefficients to produce 32 PCM samples. Thus, the 36 groups per channel in a frame produce 1152 decoded audio samples. The signal is represented as a finite sequence of data points as a sum of cosine functions oscillating at different frequencies.

The synthesis filter bank in MPEG audio decoding employs a polyphase filterbank architecture to split and subsequently recombine the subbands smoothly. Polyphase filtering ensures seamless transitions between subbands, thus minimizing artifacts and maintaining audio quality during reconstruction. Relying on Quadrature Mirror Filters (QMF), specialized filters are designed to split the input signal into multiple subbands without introducing aliasing and maintain the signal's integrity during subband splitting. A QMF splits the input signal into two parts: one passes through a low-pass filter and the other through a high-pass filter. These two filters are "mirror images" of each other in frequency response, which helps ensure that the original signal can be perfectly reconstructed from the subbands. In MPEG decoding, this approach helps ensure that the split frequency components remain in phase and that energy is preserved across the bands.

2.4 Development technologies

This section provides a brief overview of the technologies and libraries that will form the base for the current framework and its associated methodology. It also presents key features and their central role in the software implementation of the decoding procedure for the MPEG audio codec.

2.4.1 Field Programmable Gate Array

Field-programmable gate array devices encapsulate the physical computing system. It is a hardware interface made of digital components like a standard computer system, but with the advantage of being composed of a dedicated fabric of reprogrammable logic that allows developers to create digital systems designs, implement complex logic gate blueprints using Hardware Description Languages such as VHDL or Verilog, and simultaneously leverage the FPGA's potential to achieve optimised computational throughput and low latency.

Resource type	Intel Cyclone-V GT	Xilinx Kintex (UltraScale KU040)	Xilinx Artix-7 (Basys 3)
Logic elements/ cells	301 000 LEs	444 480 Cells	33 280 Cells
Flip-flops	110 000	406 720	20 480
Logic blocks (ALM/ LUT)	113 560 ALMs	203 360 LUTs	20 800 LUTs
BRAM	7 437 Kbits	21 120 Kbits	1 800 Kbits
DSP	342	1 700	90

Table 2.1: Comparison of FPGA device resources

- **Intel Cyclone-V GT**

The development board includes integrated transmitters/ receivers that support data rates up to 6.144 Gbps and support external memory DDR3 SDRAM interfaces. The FPGA also offers a rich set of I/O options, including LVDS interfaces, a memory unit of 384 MB, 32-bit DDR3 SDRAM with 8-bit error correction code via hard memory controller and 512 MB x64 DDR3 SDRAM via soft memory controller, plus 1 GB x16 synchronous flash. The communication interfaces comprise four PCIe connections, gigabit Ethernet and a USB-Blaster II (JTAG) for configuration.

- **Kintex UltraScale AES KU040 DB-G**

It offers high performance with a system that has 1 GB DDR4 SDRAM (x32 @ 1600 Mbps), 32 MB QSPI flash for configuration and user code/ data, two 10/ 100/ 1000 Ethernet interfaces (RGMII), and two gigabit transceiver interfaces with subminiature A, a type of coaxial connector commonly used in radio frequency (RF) interfaces, supporting up to 12.5 Gbps. Additionally, a programmable LVDS clock source with 250 MHz and a UART-USB interface is available. It is designed for applications demanding substantial computational capabilities and high-speed data processing.

- **Digilent Basys 3 Artix-7**

Features the Xilinx Artix-7 FPGA, optimised for low power and cost-effectiveness and is suitable for educational purposes and prototyping. The development kit includes a 100 MHz crystal oscillator with a 32 Mbit serial flash for configuration and data storage, a USB-UART bridge for programming and communication, 12-bit VGA output, and three PMod ports, including one for Xilinx ADC signals. While it has fewer resources than the other two devices, it provides a sufficient platform for developing and testing functionalities for embedded systems applications.

2.4.2 System-on-chip

The SoC design approach has become a key enabler for advancements across the integrated computer chip industry. This has led to the emergence of platform-based design techniques, in which a more flexible, programmable or reconfigurable medium is reused across a set of designs within a specific application domain [14]. SoC designs, particularly in FPGA-based environments, now drive the development and adoption of industry-wide standards by offering a flexible and customizable architecture [15], making it instrumental in the evolution of embedded systems.

2.4.3 The IOB-SoC project template

The system-on-chip developed by *IObundle, Lda* provides a RISC-V instruction set architecture-based platform appropriate for hardware and software development designated as IoB-SoC [16], a SoC template coded with Verilog hardware description language. *IObundle* provides development tools such as interfaces and computing hardware to facilitate the necessary communications with the development boards. These resources are suited for testing and simulating the provided SoC architecture.

A high-level diagram is provided in figure 2.9, containing the main components of the SoC used in the project. It is constituted by a memory system with an internal RAM and external DDR controller conducting L1/ L2 type cache systems, a central processing unit, a list of available peripherals that can be used according to the desired functionalities, such as Ethernet port interface, I²S (Inter-IC Sound) and AXI-4 (Advanced eXtensible Interface - full, lite and stream) components to link with subsystem devices. The system can be ported to FPGA or CGRA and deployed as an ASIC device.

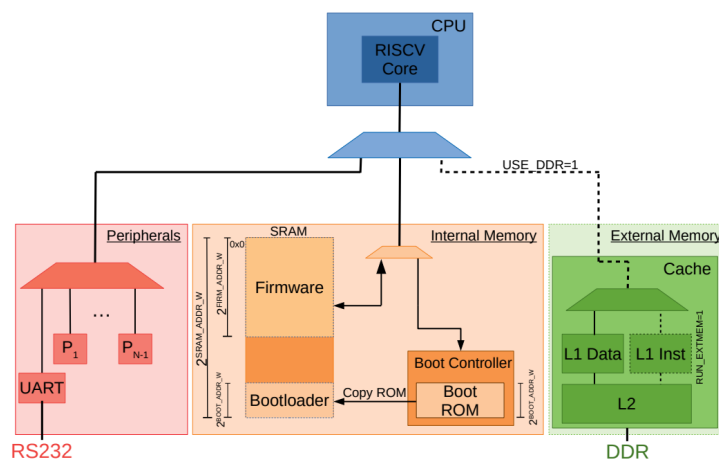


Figure 2.9: High-level block diagram of the IObundle-SoC
Source: PEPC Online Workshop Presentation 25/01/2022

An overlay design, a virtual layer of architecture that is conceptually located between the user application and the underlying physical FPGA, is implemented to allow greater flexibility by abstracting hardware details, enabling easier reuse and development of the system for different applications [17]. This approach is further supported by a type of hardware abstraction layer which provides a uniform interface for software development independent of the underlying hardware. Using abstraction facilitates portability across different hardware configurations, thus enhancing the versatility and scalability of the SoC. Therefore, this project template can be extended to implement customized processing systems with hardware and software solutions.

a) Project workflow

Managing directories and their source file contents should be consistent across teams of developers that contribute separately to a joint project. A typical issue can arise in that scenario, where nested folders depend on multiple-level project directories and should be coherent to keep a stable, compilable system version. The SoC template project repository is organised using Git submodule references, allowing users to maintain an online repository as a subdirectory of another project, keeping each set of commits and modifications independent within the entire project as much as possible.

This methodology facilitates a parallel and recursive adjustment of the main branch and continuous integration of new features with *pull requests* from developers and their local projects merged with the upstream and stable version.

a).1 Build environment and toolchain integration

Building software is a distinct and essential development activity requiring specialised tools. The term "build system" often refers to software that automates processes in a software project, explicitly focusing on source code processing. The underlying build environment adheres to this definition and is designed to facilitate this unique aspect of software development. A substantial degree of automation is backed up by a building structure using Makefile [25] and Python scripts, which define the execution tasks and operations for compilation and testing purposes.

A build automation system uses Makefile and Python scripts to streamline development. This automation facilitates various stages, including computer emulation on *Linux* [26] environments, system simulation, *Field Programmable Gate-Array* deployment, and document generation. The Makefile is configured to execute a series of commands that compile the code, simulate the performance, deploy it on an FPGA, and generate necessary documentation. This setup significantly reduces the manual effort involved in repetitive tasks and minimises the risk of errors, ensuring a smooth and efficient workflow. The automation encompasses different environments and targets, adapting to the needs of each stage in the development process, from initial testing on a computer emulator to the final deployment on an FPGA.

A template linker script controls the memory layout of the compiled program. It specifies how code and data sections are mapped to the SoC's memory, ensuring each segment is placed in the correct location. With this script, developers can manage the allocation of memory resources and the arrangement of executable code within the system. The template is designed to be adaptable, allowing for easy adjustments to accommodate different memory configurations or system requirements.

The compilation process generates an Executable and Linkable Format (ELF) file, a common standard for executable files in Unix-like systems. The ELF file contains the machine code and information about the program's entry point, data sections, and symbol table. This format is ideal for the detailed analysis and manipulation of the code, providing valuable insights during the debugging and testing phases.

After generating the ELF file, the next step involves converting it into a binary file format. This binary file represents a more compact and direct machine code version that can be loaded onto the FPGA. Following the binary file creation, it is further converted into a hexadecimal file type. This hexadecimal representation is often used for programming FPGAs, as it provides a clear and straightforward format for loading the program into the DDR memory.

Nix-shell is employed to manage dependencies and provide a consistent development environment. Nix-shell [27] creates a reproducible environment that isolates dependencies specific to the project. This tool ensures that all developers and systems involved in the project use identical versions of tools and

libraries. It streamlines the setup process for new developers and aligns the development environment across different platforms, enhancing collaboration and efficiency in the development process.

a).2 Hardware/ software co-synthesis

Cross-compilation plays an elementary role in both hardware and software synthesis. It involves compiling code on one platform (e.g., a developer's workstation) to run on a different target platform. This process is crucial for efficient development, as it allows for rapid iteration and testing of code before deploying it on the actual hardware.

For hardware synthesis, cross-compilation involves generating binary files compatible with the FPGA hardware, ensuring that the synthesised hardware design behaves as intended. In software synthesis, it is used to produce executables that run effectively on the specific architecture of the SoC, considering its unique characteristics and constraints. The compiler transforms the high-level and assembly code into machine code that can be executed efficiently by the RISC-V processor, considering factors such as processor architecture, memory management, and specific instruction sets available on the RISC-V platform.

In addition to software compilation, the development process involves compiling Verilog components to implement custom digital and logical systems. Verilog, a hardware description language, describes the digital circuits from the gate-array fabric and system architecture. These Verilog components are compiled into a cycle-accurate behavioural model to verify that the hardware design meets the required specifications, particularly in timing and synchronisation, which are critical in audio processing applications.

This model precisely emulates the behaviour of the digital circuits on a cycle-by-cycle basis, providing an accurate representation of how the hardware will perform in real-time. This level of precision in the behavioural model aids in identifying issues early in the development cycle, facilitating corrections before the actual hardware synthesis.

b) SoC architecture

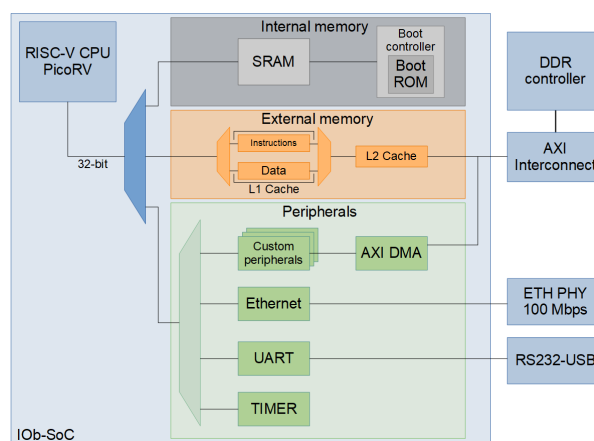


Figure 2.10: Block diagram of the system-on-chip hardware structure.
Source: IOB-SoC-based Tester System [43]

The hardware domain of the IOB-SoC template is constituted by a set of definitions written in Verilog language. The system-on-chip admits a memory-mapped input/ output peripheral interface to communicate between the CPU and other components/ devices. The memory scheme addresses both memory

sections and I/O devices, where each address represents a singular memory word or register existing in the SoC. Every individual peripheral device is given an address space to be accessed.

The IOB-SoC configuration file also contains the definitions of the macro parameters used to set up the SoC's hardware. It takes into account the directives for a conditional compilation that defines the operating modes for initialising internal or external memory with the firmware binary file and data bus selection bits: valid, extra memory, peripherals, and boot controller select.

The instantiation of the modules of the core system is present in the top source file `system_core.v`, such as the CPU along with data and instruction buses, the internal SRAM, and external DDR memory units with the respective data and instruction split bus and the peripheral instances. Once the programmer incorporates the necessary peripherals or other adaptations, a new file is created, `system.v`, upon the execution of the make procedure during the template build.

b).1 Source

It contains a set of Verilog source files where each instantiates a different module. In particular, bootloader and ROM initialisation are defined in the `boot.hex` file, `iob_soc_boot_ctr.v`; internal `iob_soc_int_mem.v`, and external along with IOB-Cache instance, `iob_soc_ext_mem.v`, memory-related buses using AXI interface standard; also the SRAM component declared in the file `iob_soc_sram.v` with instruction and data ports implementing a native interface with a dual-port RAM instantiation; the system module in `iob_soc.v` that is used as a foundation for the top level module of the design when the various peripherals are included.

b).2 Simulation

It consists of the combination of Makefile targets that consider the SoC macro definitions to build and test the customised template with the modified components within a simulation environment, namely the *Icarus* [18] and *Verilator* [19] Verilog logic converter for simulation and synthesis tools.

b).3 FPGA

The directory definition incorporates source code files for Vivado or Quartus logic device design software. The synthesis and implementation scripts acknowledge a top module system and device encapsulating the IOB-SoC and the respective compiled application software pre-loaded on SRAM or DDR memory, according to the specified external memory option. The Makefile targets are configured in `fpga_build.mk` and serve to define the necessary sequence of commands to build the software and hardware components. The resulting bitstream holds the information arrangement and is loaded onto the FPGA.

c) Software

Software for the IOB-SoC is primarily based on C and Python languages, providing both low-level hardware interaction and higher-level scripting and automation capabilities. The C code is used for firmware development and direct interaction with hardware registers, while Python scripts facilitate the build process, simulation, and interaction with the SoC. It supports a hybrid setup, integrating low-level programming routines alongside higher-level abstractions. This makes it possible to customize performance-critical sections while keeping other parts more easily maintainable.

The Makefile targets defined in `sw_build.mk` handle the compilation of the bootloader, peripheral drivers and firmware components. They also support building PC-based emulation for custom settings and debugging purposes.

c).1 Console

It is a software section that serves as an input and output file management interface. It enables data delivery to and from the target computer system, with the IOB-SoC inside the FPGA sending that application's binary file to be executed during its boot course and receiving standard output messages from the system-on-chip.

c).2 Bootloader

It is a program running in the system-on-chip that gets executed at the startup to manage the system initialization. It is also responsible for loading the firmware binary file using the console via UART and rebooting and executing the firmware program. The cache initialisation is also performed, given that the user configurations are set to use DDR memory and run the program from external memory instead of the SRAM memory section. This feature allows the user to send only the firmware component through the hardware communication module to the system-on-chip without recompiling the bitstream.

c).3 Firmware

This software segment is a layer of hardware-dependent software that provides a necessary set of instructions for device communications and data transferences with other computer hardware. Often, the word firmware is used as a synonym for device-specific software. The user can implement the desired application, define the required lower-level operations, and perform all monitoring, control, and data manipulation functions.

c).4 PC-emulation

The emulation procedure is a standalone software version of the system-on-chip template that runs on the local machine. This software section allows the programmer to test the application behaviour singly with the option to debug more easily and verify memory leaks and possibly existing data conflicts.

d) Peripherals

The IOB-SoC project incorporates a library of peripherals written in Verilog, available through the IOB-Lib repository [20]. These peripherals are designed to be used as subsystem IP cores, allowing easy integration with the system-on-chip. The library includes hardware components that enhance the SoC's capabilities, providing a foundation for building a fully functional embedded system.

These peripherals are implemented using the AXI4 and AXI4-Lite protocols, which provide high-performance memory-mapped communication interfaces between the CPU and peripheral devices. The modular nature of these components allows developers to easily add or remove specific peripherals as needed, ensuring that the SoC can be tailored to meet the particular requirements of a given application.

d).1 AXI Interconnect

The AXI4 interconnect is designed to establish communication between peripherals and memory controllers. The community of embedded system developers highly adopts this standard interface, ensuring compatibility and interoperability among various components from different manufacturers.

The *Verilog-AXI* [21], based on the interconnect template collection of AXI4 and AXI4-lite bus components, implements a high-performance memory-mapped data and address interface. The full version is capable of burst access to linked devices. In contrast, the AXI4-Lite subset, which lacks burst transfer access capability and has a more straightforward interface than the full AXI4, is also available. Instances are fully parameterisable regarding interface widths and the number of linked devices from both sides.

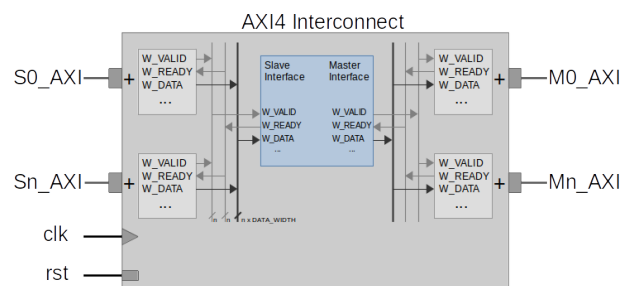


Figure 2.11: Block diagram of the IObundle AXI-Interconnect hardware component

Supports multiple initiators and targets, operating on separate channels designated for read and write operations. This approach further divides the communication into address/control and data phases. Such segmentation enables more efficient pipelining and overlapping of read and write operations and data transactions between components, thus enhancing throughput.

d).2 Central Processing Unit

The core of computing systems is responsible for executing instructions and managing system resources. It consists of several fundamental components, including the instruction register, decoder, program counter, arithmetic and logic unit, and connections to memory. It communicates with other components through data and address buses and control lines that manage the flow of information. The memory bus connects the CPU to the memory subsystem, while the peripheral bus allows communication with various peripherals.

A CPU wrapper converts the processor interface signals to the native interface format used internally throughout the system. This wrapper contains the necessary combinational and sequential logic to implement this interface, enabling efficient module interconnection.

Instruction execution within the CPU typically follows a series of stages for a pipelined processor: instruction fetch, decode, execute, and writeback. In each clock cycle, the central processing unit performs one of these fundamental operations, breaking down each program instruction into micro-operations carried out across multiple stages. This structured approach allows the processing core to manage the instruction flow and ensure accurate and effective operation. In more advanced configurations, specialized co-processors or accelerators can be integrated to offload specific tasks, such as floating-point arithmetic or cryptography computing, enhancing the system's overall performance. The floating point units handle floating-point operations, which are essential for many scientific and multimedia applications, reflecting on the efficient execution of complex arithmetic involving non-integer values.

While scalar architectures provide a single operation per clock cycle, the pipelined architecture allows multiple instructions to be decoded and executed in parallel, increasing instruction throughput. For instance, multiply and add operations can be queued simultaneously, separating each processing stage to dispatch unprocessed instructions. The PicoRV32 [22], an open-source processor that implements the RV32IMC instruction set, is used by default with the SoC template. However, other architectures are supported for more complex scalar/ multi-issue processors, such as the VexRISC.

d).3 Cache

Instruction and data caches are essential elements in a system-on-chip because they minimise the latency between the CPU and main memory. These caches store frequently accessed data and instructions close to the processor, reducing the need to fetch them from the slower main memory repeatedly. This proximity significantly speeds up data access times, taking advantage of the locality of reference principle, where recently accessed data is likely to be acquired repeatedly during the processor's operation.

The *IOb-Cache* [23] is a configurable IP core that administers cache controller and memory accesses. This module can handle data processing in stages (pipeline architectures), allowing it to handle one instruction (read or write) per clock cycle for faster performance. It can connect to different memory systems through either a custom high-speed connection, IOBundle's *Native Pipelined Interface* for the processor-side interface (front-end), and an AXI4 interface for the memory-side interface (back-end), which can also be configured with NPI.

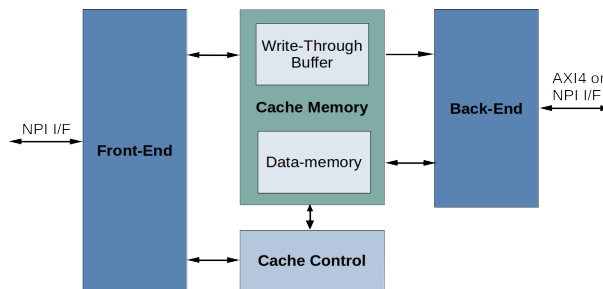


Figure 2.12: Block diagram of the cache memory IP core
Source: IObundle component documentation

Suppose the external memory interface is selected, allowing access to a larger DRAM memory (DDR4). In that case, the hardware description of the SoC includes a two-level hierarchy with an instruction and data L1 cache and a shared L2 cache that is located there. The L2 cache communicates with a third-party DDR memory controller IP core using an AXI4 master bus.

d).4 UART

The Universal Asynchronous Receiver / Transmitter protocol (UART) is used to transmit and receive serial data with external systems. It also serves the purpose of data transfer and debugging programs by printing valuable information in the console. A UART module is integrated into the system-on-chip to enable communications between the target system and the host.

The *IOb-UART* [24] peripheral is a RISC-V open-source-based hardware component written in Verilog and includes a C software driver to control the device. It provides full-duplex transmission and configurable baud rate in addition to clock frequency. The UART peripheral is used for text output,

console presentation display and program debugging. A *printf* API is adapted in the SoC to substitute regular implementation with a leaner version tailored for embedded systems.

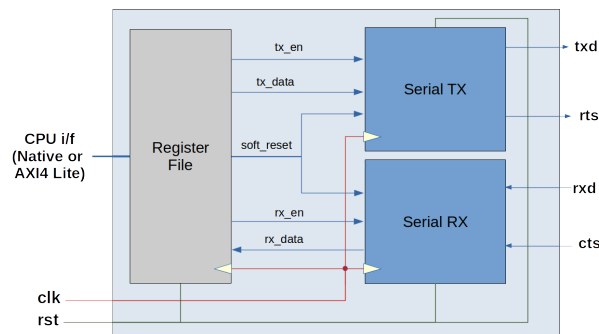


Figure 2.13: Block diagram of the IObundle UART hardware component
Source: IObundle component documentation

Another feature is controlling the system's transmission state under testing, orchestrated by the tester submodule. It can receive and send messages to the external element that should initiate the decoder operation, as well as the dispatch into the memory region used to store the compressed file to be interpreted and the subsequent digitally representative sampled signal given as the product of the decoding process. The serial interface device also manages communication outside the decoder loop. It loads the bootloader (to the BOOT ROM) and firmware (SRAM or DDR external memory) binaries to each of the specific memory regions used by both software sections.

e) Primary parameters and SoC configuration

Given the different sizes and complexities of digital computing applications and dynamics between peripheral components or onboard-computing units, it is essential to define and constrain the most critical areas of the workset relative to the hardware system to specify memory regions or implementations on logic-level procedures accurately. To conduct the dedicated software course of action, the programmer should consider a collection of environmental variables intending to delineate the hardware's intrinsic properties, adapt to the implemented software routines and make design decisions based on the available parametrization. The system is explicitly defined, synthesized and assembled using models of available system components.

With the integration of application software with integrated chip environments comes the dimensioning of the system at different levels of abstraction. This aspect allows the user to define the system's main characteristics regarding the size of memory elements, central processing unit bus width and instruction set options, transmission rate, and input/ output hardware device or the simulation board choice. The options can be taken as environmental variables defined in a set of Makefile directives in each directory.

Configuring the IOb-SoC system through these parameters allows for a highly customized system-on-chip design, balancing performance requirements and resource constraints. These parameters offer a versatile framework that facilitates tuning the system for different application needs, optimizing processing speed and expediting peripheral configuration. By precisely defining the hardware and software interaction, the SoC can be tailored for specific tasks, enhancing efficiency and ensuring compatibility with the intended application environment.

This list represents a set of mandatory and optional definitions of the setup options for the customized SoC, determined by the developer's choice of memory-mapped dimensions and hardware properties such as system clock frequency and transmission rate, choosing the FPGA board or the simulator, and enabling CPU instruction set extensions.

The following declarations in `iob_soc.py` are derived from these primary parameters, such as the peripherals submodules and both software, hardware and documentation paths for the SoC.

- **Pre-initialise memory: INIT_MEM**

Option to enable storing a program received by the UART and start-up execution from it, or boot-loader, firmware and data binaries are loaded to the pre-defined memory type (SRAM or DDR). The flag indicates whether the firmware is already loaded in the FPGA or if the *Console* needs to transfer the firmware to the SoC.

- **DDR usage option: USE_EXTMEM**

Habilitation of DDR layout and memory usage alternatives to specify the address location of the software application program being run either from internal or external memory in the IOB-SoC.

- **Transmission rate : BAUD**

Communication channel transference default speed value of 115200 bits/s.

- **Frequency: FREQ**

System clock with a default frequency value of 100MHz.

- **CPU architecture: DATA_W & ADDR_W**

Data and address width relative to the input and output of the AXI interface and peripherals port map.

- **Firmware address size: FIRM_ADDR_W**

Logarithmic base two approximation relative to the firmware binary size to assign into memory space a specific region to load the resulting file from the compilation procedure.

- **SRAM address size: SRAM_ADDR_W**

Logarithmic base two approximation relative to the size of SRAM to assign into memory space the addresses specific to the region for the internal memory.

- **DDR memory address size: MEM_ADDR_W**

Memory bus address width.

- **Bootloader ROM address size: BOOTROM_ADDR_W**

Logarithmic base two approximation relative to the size of the boot ROM to assign into memory space a specific region for the allocation of the bootloader program, which should be sufficient to hold the bootloader program and data.

- **RISC-V instructions: USE_MUL_DIV & USE_COMPRESSED**

Variables to define the instruction set extensions on multiply and division operations for values held in the integer registers and compressed RISC-V instructions.

- **Simulation default: SIMULATOR**

Definition of the default simulator. The simulator runs locally if SIM_SERVER and SIM_USER are not declared.

- **Board option: BOARD**

Representation of the default target board to link with the local machine that can be used to run the template implemented design for automatic testing.

2.4.4 RISC-V Instruction Set Architecture

The RISC-V Foundation was created in 2015 by the Parallel Computing Laboratory (Par Lab) at the University of California at Berkeley. It is a free and open-source Instruction Set Architecture that provides a method to create intellectual property and a more accessible approach to the digital processing products and devices market. The system-level organization of a RISC-V hardware platform can vary significantly, ranging from single-core microcontrollers to large-scale clusters of many-core server nodes with shared memory. Even small systems-on-a-chip can adopt a hierarchical structure, incorporating multicomputer or multiprocessor configurations. This modular approach not only simplifies the development process but also enables secure isolation between subsystems, enhancing flexibility and security in various embedded and general-purpose applications [28].

This architecture type is composed of 32 general-purpose integer registers from x0 to x31, plus a program counter (instruction pointer), with an additional 32 floating-point registers for the implementation of the respective extension, except in the embedded subset variant, which has only 16 integer registers. These assembler mnemonics are based on register-address instruction roles, except for memory-access instructions that reside outside the scope of the integer registers. The register types are defined according to the following table 2.2:

Register name	Symbolic name	Description
32 integer registers		
x0	zero	Hardwired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary or alternate return address
x6-x7	t1-t2	Temporaries
x8	s0/ fp	Saved register or frame pointer
x9	s1	Saved register
x10-x11	a0-a1	Function arguments or return values
x12-x17	a2-a7	Function arguments
x18-x27	s2-s11	Saved registers
x28-x31	t3-t6	Temporaries
pc	-	Program counter

Table 2.2: RISC-V integer register set

The RISC-V ISA provides a straightforward encoding format, which breaks the instructions into fields that can be efficiently parsed and executed by the CPU. It also allows for the inclusion of optional extensions to the base ISA, providing additional capabilities.

Processor instructions include a set of arithmetic, logical, memory access, and control transfer instructions. Arithmetic and logical operations include addition, subtraction, shifts, and bitwise operations. The floating-point extension adds another set of 32 registers, which are crucial for efficiently handling arithmetic operations involving real numbers, particularly for applications such as scientific computation and multimedia processing. The floating-point registers are detailed in the following table 2.3:

Register name	Symbolic name	Description
32 floating-point extension registers		
f0-f7	ft0-ft7	Temporaries
f8-f9	fs0-fs1	Saved registers
f10-f11	fa0-fa1	Arguments or return values
f12-f17	fa2-fa7	Arguments
f18-f27	fs2-fs11	Saved registers
f28-f31	ft8-ft11	Temporaries

Table 2.3: RISC-V floating-point register set

Memory access instructions, such as load and store, interact with the memory subsystem to move data between registers and memory locations. Control transfer instructions include branches and jumps, which update the program counter (PC) and control the execution flow.

Control and status registers are another essential part of the RISC-V ISA. They manage the system state, handle exceptions, and perform privileged operations. These registers are integral for handling exceptions and managing system control, particularly in complex SoCs with multiple processing elements and peripherals.

Exception handling in RISC-V follows a well-defined model. Exceptions can be triggered by illegal operations, such as dividing by zero or accessing unauthorized memory. When an exception occurs, control is transferred to a specific handler, which processes the exception and determines the appropriate response. This could involve recovering from errors, halting execution, or notifying higher-level software components. The handling of exceptions is facilitated through control and status registers, which keep track of the system state and allow for effective management of exceptional conditions.

The architecture is well-suited for embedded systems, providing a small footprint and a simple set of instructions that are easy to implement. The ISA's minimalism and modular nature allow for flexible implementations, where different subsets and extensions can be chosen based on the application's specific needs. In contrast, more general-purpose RISC-V implementations can include a broader range of extensions, enabling them to run complex operating systems and handle high-performance computing tasks.

RISC-V's extensibility is one of its defining features. The base integer instruction set, often referred to as RV32I or RV64I depending on the bit width, can be augmented with various optional extensions to meet specific application requirements:

- **Standard extensions** RISC-V's standard extension type is an optional component that can be added to the base integer ISA to improve its capabilities. These include extensions for atomic operations (A), integer multiplication and division (M), single and double-precision floating-point operations (F and D), among others. Each of these extensions is denoted by a single letter and can be combined in various ways to create a customised ISA that meets the requirements of a particular hardware or application.
- **Compressed subsets** The compressed subset aims to reduce the size of programs and increase performance in systems where memory bandwidth and size are limited [29]. This subset defines a set of 16-bit compressed instructions, which are more compact than the standard 32-bit instructions. These compressed instructions are crucial for applications in embedded systems or other environments where memory efficiency is a high priority.

- **Embedded subset** The embedded subset is made for embedded systems and offers a downsized version of the base integer ISA. This subset is designed for implementations requiring minimal resources, such as microcontrollers and other low-power devices. Instead of the 32 integer registers of the regular extensions, only 16 (x0-x15) are supported. This reduction reflects an estimated saving of 25% core area with a corresponding core power reduction [28].
- **Arithmetic and logic sets** These sets encompass extensions that provide additional arithmetic and logical operations beyond the base ISA. They include operations for complex number arithmetic, bit manipulation, and various specialised mathematical functions. These sets are vital for applications that require advanced mathematical processing, such as signal processing, cryptography, and scientific computing.
- **Privileged instruction set** The privileged instruction set defines instructions for controlling hardware and managing system-level functions. This set includes instructions for handling exceptions and interrupts, virtual memory support, and system control operations. It is fundamental for operating system development and implementing virtualisation & security features in more complex computing systems.

2.5 Related work

This section briefly introduces some solutions for ARM (Advanced RISC Machines) and TI (Texas Instruments) platforms, with an example of RISC-V support.

- Picus Tech developed the first decoder example. It is available for ARM platforms and several supported operating systems: Linux, Android, WinCE, Symbian, and Windows Mobile. It also supports all standard bit rate configurations for the MPEG-1, 2, and 2.5 versions.
- A decoder provided by AllGo Systems is also available for the three versions of the codec standard (1, 2, 2.5) layer III (a.k.a. MP3), in every sampling rate, from 8 to 48 kHz, and the defined bit rates from 8 to 320 Kbps. It also presents CBR and VBR (constant/ variable bit rate) playback features where the decoder adjusts the data rate according to the media file needs. Otherwise, these parameters remain consistent throughout the decompression process. It also supplies analysis for mono, dual, stereo, and joint stereo channels and bitstream error handling. It is accessible for the ARM9, ARM11, Cortex M3, and M4 platforms.
- Ittiam System developed a solution for the ARM and Texas Instruments nodes and provides a decoding module for the layers I, II, and III of the MPEG-1 ISO/ IEC 11172-3 and ISO/ IEC 13818-3 standards and is also developed for multi-channel information with a 24-bit pulse code modulation output.
- A fixed-point MP3 decoder developed by RealNetworks offers support for layer three but not for the first or second, and the MPEG-1, MPEG-2, and MPEG2.5 versions with CBR, VBR and FBR (free bit rate) modes for the media files, as well for all the mono and stereo modes. It is statically linkable and reentrant, meaning that it can run on a single processor system and be safely called after an interruption before the preceding invocations finish execution. The supported toolchains and processors are x86 (Microsoft Visual C++), ARM (ARM Developer Suite, Microsoft Embedded Visual C++, GNU toolchain), and RISC-V (GNU toolchain).
- The CWda75 developed by CoreWorks is a high-performance, fixed-point audio decoder IP core designed to support MPEG-1 and MPEG-2 Layer I/ II audio decoding with full compliance, offering robust decoding capabilities for bit rates ranging from 32 Kbps to 448 Kbps and sampling

frequencies of 16 kHz to 48 kHz. The decoder supports mono, stereo, and joint stereo channels and includes error detection and concealment mechanisms to ensure reliable audio output even in bitstream errors. Additionally, it is highly configurable and is designed for easy integration into various ASIC and FPGA platforms with minimal resource usage. The audio input and output interfaces use a native parallel interface. Standard audio interfaces, such as I2S/ TDM and SPDIF, are also available.

Chapter 3

System design

This chapter presents the composition between the development environment provided and the proposed implementation of the decoder algorithm. In practice, a set of objectives for integrating and evaluating the application deployment based on the system-on-chip architecture, prerequisites and implementation considerations are described for each development task.

The integration strategy combines hardware and software elements, certifying that the MPEG audio decoder operates efficiently within the RISC-V CPU architecture. Prominence is placed on exploiting the proposed SoC structure's available options and capabilities, tailoring it to accommodate the computational requirements of the application process.

Considering the flexibility characteristic in software-defined applications, the study aimed to optimise the all-inclusive frame decoding delay. This delay stems from the sequential nature of the audio codec specification in its different layers, with the available and detailed *Reduced-Instruction-Set-Computer* resources with parallelised computing strategies, with a dedicated hardware module as the secondary goal with the purpose of acceleration, to be able to build a reliable product for audio processing that can be used in multimedia streaming platforms.

This requirement proposes a set of problems to resolve, including analysing the decoder application's primary and most intensive tasks to get to a comprehensive design solution that is compliant with the initial specifications. Using the GNU [30] profile tools capabilities, an a priori study was performed to examine the computational complexity of the audio decoding algorithm based on an isolated software implementation running locally on a computer system.

Relative to the design process, from the prototyping to the application deployment in FPGA boards, a set of pre-configured verification sequences and tests with different audio content is performed. The referred *IObundle, Lda.* system-on-chip iterations, which can be adapted for a wide variety of embedded system programming applications, can be employed later for an ASIC environment and manufactured with the concurrent developed model.

Derived from the characteristics of segmentation and modularity of the system-on-chip, with the use of specific interfaces such as the AXI Stream, the IOb-SoC-SUT [31] is used for the validation stage of the unit under test development cycle, in this case, enclosing the decoder firmware with the core components of the native IOb-SoC template and the respective interconnection elements that allows the establishment of an operation command based on requests and provision of tester information.

Subsequent chapter sections decompose the main modules concerning software, hardware, and test methodology. They also describe the system components and thoroughly explain how both domains are integrated with the audio decoding functionality present in the design.

3.1 Hardware structure

When using the system in an isolated mode, in a self-contained operation mode, it is essential to specify communication structures to correctly read encoded audio data from a specified region in memory and output the PCM audio to its corresponding memory space that is contiguous with the input address space. The process is inherently coupled with shared information between the software core and its functional hardware unit counterparts.

This section addresses the system's hardware structure based on the IOb-SoC template. It can be used as a stand-alone functional system or integrated as a unit under test (UUT) that can be attached to a tester. In the most recent version regarding the IOb-SoC, an abstraction layer based on Python scripts is utilised to streamline the configuration and integration of peripherals and automate the creation of a build directory with all necessary files and build makefiles for running emulation, RTL simulation and FPGA. Notable improvements feature the automated generation of Control and Status Register (CSR) files, covering both hardware components and bare-metal software, along with the production of standard interfaces like AXI4. Additionally, incorporating the UART and LIB submodules directly into IOb-SoC minimises reliance on git submodules and mitigates related issues. The IOb-MPEG-Decoder system is derived from the IOb-SoC template in terms of structure, with the addition of two AXISTREAM interfaces for input and output data. The following figure 3.1 shows a high-level block diagram of the IOb-MPEG-Decoder system.

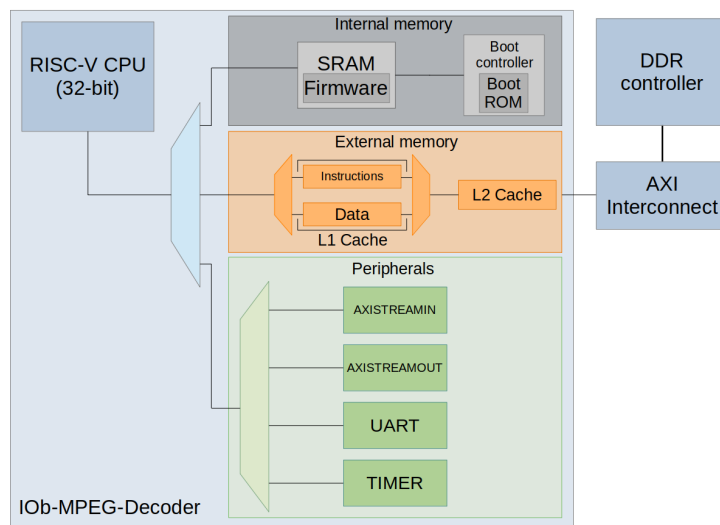


Figure 3.1: Block diagram of the IOb-MPEG-Decoder

This setup ensures a separate build directory to run simulations and tests, providing a clean environment to develop code and organise the repository. The Python code is part of a more extensive system that defines and manages an SoC's setup for development and testing purposes. It abstracts hardware details into configurable software entities, facilitating more accessible simulation and modifications.

The file, `iob_soc_mp2d.py`, is based on the `iob_soc_opencryotlinux.py` template. It showcases the definition of the class conveying the specific configuration of the decoder system, with import statements to include necessary modules for handling hardware logic, submodule list creation to organise and interconnect components, peripheral port mapping to define connections between ports and system interfaces, and post-setup and file management to ensure the alignment of software components with the setup. The `iob_soc_mp2d` class extends from the parent class, `iob_soc`, part of the hardware design configuration framework. The code is structured to configure and manage different peripheral interfaces

that are instantiated as attributes from the superclass in the form of submodules. These submodules are designed to manage and coordinate the interactions between different hardware components within the system-on-chip. The first step was to set the instances for the Timer and AXI Stream Input & Output peripherals since the DDR controller, UART, and AXI interconnect are inherently defined in the architecture.

3.1.1 Embedded system components

The baseline architecture for the IOB-MPEG-Decoder comprises six main components, namely the CPU and cache, AXI4 interconnect, the UART16550 and AXI-Stream bus interfaces for data input/output and a timer to measure the clock cycles count of algorithm routines, described in the following paragraphs and the template presentation, in section 2.4.3. These are implemented over the dedicated software-based decoder to ensure the correct execution and integration with other devices.

a) Central processing unit

The default processing core used in the IOB-SoC is an adaptation of the PicoRV32 [35], a 32-bit CPU which implements the RV32IMC instruction set (integer, multiplication & division and compressed extensions) at a frequency of operation measuring 100Mhz.

Onset implementation is done with the mentioned processor. Software profiling measurements are used to create a baseline coverage of the behaviour of the decoder application running on the SoC, and improvements are compared between the default model and the following ones. This element provides most functionalities but comes at the expense of less performance. Whilst this system operates over a fixed-point unit, the same numerical format as the software, it may not be the best choice compared to the other options, and additional factors must be considered.

The replacement of this component is similar to other submodules and, therefore, straightforward to change since the IOB-SoC is CPU-agnostic. Alternatively, the following architectures are provided by *IOBundle* and available to use in the context of this project:

- **VexRISC-V**

Is a 32-bit Linux-capable CPU [36] with an RV32IM (integer and multiplication & division extensions) instruction set pipelined on five stages (fetch, decode, execute, memory, write-back). It also employs a barrel shifter for a cycle-efficient bit shift operation: a cascaded multiplexer to execute logical shifts in binary objects with reduced decoding logic for the shift count. These features make this one of the best candidates for the proposed system to meet the real-time requirements. A repository provided by IObundle [37] is available with the hardware necessary to integrate the VexRiscv CPU on IOB-SoC.

- **DarkRISC-V**

Is a 32-bit CPU [38] that implements the RV32IE instruction set (integer and embedded extensions) and supports a three-stage pipeline. A custom 16x16-bit MAC, register-to-register instruction (R-type), was defined in the reserved OP code space with the identifier *b01010_11*, for computations attending to the digital signal processing aspect of the system. This processor also contains dedicated modules for arithmetic computing instructions (floating-point, multiply and divide). IObundle provides a properly verified and implemented architecture [39] prepared to attach to the SoC.

b) AXI-Stream bus protocol

AXI4-Stream is a communication protocol used for unidirectional data transferences. In this system, a specified width of bits, known as TDATA, is transacted synchronously with every clock cycle provided to the module. A set of control signals is defined and mirrored, the only difference being the direction regarding the sender and the receiver (out to in).

The transmission begins when the sender issues the TVALID signal, and the receiver acknowledges readiness by returning the TREADY signal after processing the initial TDATA. Following this, the sender continuously sends TDATA along with TLAST. The TLAST indicator marks the end of the data stream, prompting the receiver to continue accepting TDATA until TLAST is activated. Figure 3.2 depicts a high-level block diagram of the AXI-Stream peripheral.

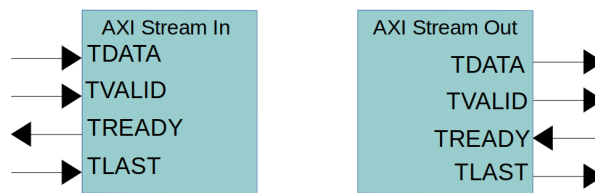


Figure 3.2: Block diagram of the AXI-Stream peripheral

In the scope of data transference allying the IOb-MPEG-Decoder and the IOb-SoC-Tester, a pair of *AXI-Stream* [40] interfaces corresponding to the input and output channel of the hardware block are accountable for dealing with the incoming encoded data from the tester system and the resulting decoded audio data. More details are provided in the following chapter.

c) Clock timer

The IOb-Timer [41] is a hardware timer with a 64-bit counter written in Verilog, with enable, soft reset and sampling capabilities. Implements an IOb-native interface with the CPU to a register file, configuration, control and status registers accessible by the software. It also includes a set of driver functions for reset, initialisation and sample timer counter, with virtualisation of software registers written for PC emulation, converting clock values from PC clock frequency to embedded frequency. The high-level block diagram of the Timer is shown in 3.3.

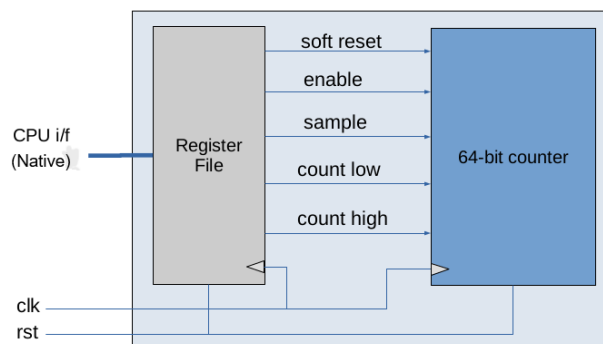


Figure 3.3: Block diagram of the timer peripheral
Source: IObundle component documentation

For system performance monitoring, as described in sub-section c), the effective use of this component allows for time tracking of the software behaviour in several parts of the system regarding the firmware and the underlying decoder application files.

The number of clock cycles between reset and sampling instances gives the execution interval when divided by the operational frequency of the system, set by default to 100MHz. Units are expressed in microseconds (μS).

3.2 Software architecture

The current section presents a more detailed delineation of the MPEG audio decoder application and the methodology of combining the elemental application interface parcels in the context of the development architecture domain of the system-on-chip. Consequently, conditions must be analysed to qualify the audio decoder operation in real-time.

The decoder in the system is implemented by combining the existing SoC firmware with the application programmable interface (API), which is natively implemented with the library. For that effect, the audio decompression source files collection is added as a *git submodule* present in the SoC's highest-level directory, making available the required functions, structures, variables and macro definitions in the upper layers of software of the system-on-chip. To streamline this process, developers use automated build tools and continuous integration techniques that mitigate the problems arising from changes in a single part of the code and do not disrupt existing system functionalities, thereby enhancing the robustness of the system's software infrastructure. Figure 3.4 shows a flow diagram representing the stages of the development/ verification tasks.

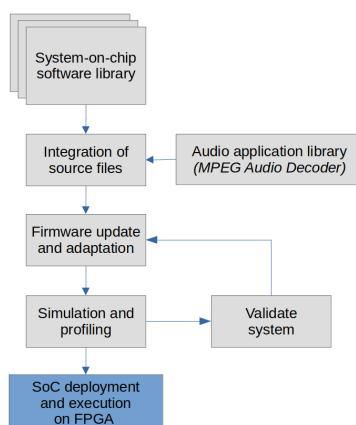


Figure 3.4: Software implementation flow diagram

The porting of the MAD decoder, originally composed in fixed-point, coded in ANSI C, entailed a task of code compilation, followed by running the program on a simulator. This step was instrumental in gathering crucial performance data, which was then analysed to identify potential areas for optimisation. Subsequently, this cycle of compilation, execution, and evaluation was methodically repeated in both development instances and involves ensuring the integration of these components within the broader SoC framework, which necessitates testing and validation procedures to confirm that the audio output meets the expected quality standards without compromising system performance. These iterative refinements aim to minimise resource utilisation while maximising the decoder's efficiency, making it well-suited for real-time audio processing tasks within the resource constraints of the SoC.

Deploying the specified application in the IOB-SoC is accomplished by integrating the source files that constitute the existing software segment with the application's native library. One of the reasons for the successful deployment of application programs in the system-on-chip architecture is its adaptability: it is designed to be modular, allowing for seamless incorporation of software and hardware components. Furthermore, the architectural design supports scalability, enabling the addition of more sophisticated functionalities or updates without significant rework, which is critical for maintaining technological relevance and accommodating future enhancements.

3.2.1 LibMAD - MPEG Audio Decoder Library

The *LibMAD* [5], an acronym for "library of MPEG audio decoder", is an open-source project for high-quality decompression written in standard C. All three audio layers are fully implemented (layers I, II and III). It is distributed under the terms of the GNU General Public License (GPL). It does not support MPEG-2 multi-channel audio, nor is it compatible with AAC. The proposed library provides the following characteristics:

- 24-bit PCM output
- 100% fixed-point (integer) computation
- Implementation based on the ISO/ IEC standards
- Hardware-agnostic, supports a collection of RISC ISAs
- Support for MPEG-1 (joint/ stereo channel mode) / 2 (LSF, single channel mode)

The results from MAD meet the computational accuracy standards required for compliance with ISO/ IEC 11172-4. Typically, MAD functions as a complete Layer III ISO/ IEC 11172-3 audio decoder, adhering to the standard's specifications. Optionally, MAD's configuration allows for adjustments in accuracy beyond the standard settings, balancing accuracy with performance.

For optimal performance, it is necessary to choose an assembly language version of the fixed-point multiplication routines. Various assembly language versions have been developed for different CPU types. MAD's subband synthesis routine, which is computationally demanding, could be made more efficient at the cost of slightly lower accuracy by modifying the fixed-point multiplication method using a small windowing constant [5].

Although this adjustment improves performance, and the accuracy reduction is typically imperceptible, it is not activated by default and must be manually enabled. Depending on the architecture, additional specific optimisations might also be available.

The repository version used was the *0.15.1b*, and the source code was modified only for code readability, debugging, and software behaviour profiling.

3.2.2 Application runtime analysis

A process involving profiling and analysing the runtime behaviour, time complexity, or memory space is crucial to an overview examination of the software's most computer-intensive functions or tasks. These features determine and influence the microprocessor system performance.

This approach can be helpful for several reasons, including a clear understanding of how the application behaves regarding CPU usage, memory consumption, and input/ output operations. Such insights are fundamental when adapting the software to a different system [32]. Although the original profiling is done on a distinct architecture, many performance characteristics, like algorithmic complexity and memory access patterns, will likely be consistent across platforms. This information can guide optimisations in the RISC-V implementation.

By exploiting the PC emulation setting of the *IObundle, Lda.* system-on-chip, taking advantage of a virtual version regarding the physical components and peripheral drivers through adapted filesystem resources. The present case used a duration measurement for each function that composes the MPEG audio decoder software.

The decoder provides two modes of operation: synchronous and asynchronous. The synchronous mode operates on the stream sequentially using a single process, characterised by a blocking behaviour,

making it impossible to perform other tasks coincidentally with decoding. On the contrary, the asynchronous use case is suitable when implementing non-blocking behaviour for concurrent processing systems and managing inter-process communication mechanisms. In this mode, different stages of the decoding process can be handled in separate threads or processes, allowing for parallel processing of other stream parts and improving performance, especially on systems with multiple cores or processors.

A synchronous approach is used in this case because the embedded systems environment does not have multi-thread support and relies exclusively on single-thread processing. In essence, the behaviour is reflected using a linear execution mode method that processes data consecutively, ensuring stable operation within these constraints.

In environments where resource constraints are significant, such as embedded systems with limited processing capabilities and memory, the predictability of a synchronous method greatly aids in minimising runtime errors and enhancing system reliability. It also reduces the overhead associated with complex thread management and synchronisation mechanisms, which are often necessary in asynchronous systems.

Both MP1 and MP2 use similar encoding methods. They employ a method of audio compression that involves dividing the audio spectrum into small frequency bands and then encoding them. This similarity in the foundational practice makes it easier for an MP2 decoder to understand and decode MP1 files. Given the definition of audio layer N decoder, where N denotes layers I, II or III of the MPEG audio standard, it is backwards compatible with bitstream data encoded in layer N and all layers below N. The analysis used MP2 and MP3 files to create a preliminary study of the decoder software's most intrinsic and demanding properties.

There are four main categories of execution concerning the decoding application:

- An initialisation phase, where the necessary components and communication peripherals are defined and instantiated. The decoding routine's essential elements are the decoder, buffer and stream structures and functions for the high-level application programmable interface.
- The input stage is responsible for managing the flow of data acquisition and buffering into the decoder, as well as with the initial error detection and handling for the incoming data stream, ensuring the integrity of the data before processing. Reading the encoded MPEG audio data, often buffered to ensure smooth processing, could be sourced from a file, a network stream, a digital audio channel or another input method.
- Decompression is the core of the decoding process, taking into concern the audio synthesis, reconstructing the signal waveform from the decoded data into a raw audio format (PCM or WAV), and post-processing, which may include operations like normalisation, equalisation or other audio enhancements. It involves various sub-steps, such as Huffman decoding, inverse quantisation, and inverse discrete cosine transform (IDCT).
- The output stage involves sending the decoded audio data quantities to an output device and synchronising them with other transport channels or systems for correct playback timing. Handling the continuous flow of the audio stream to the output device ensures smooth and uninterrupted playback.

The provided graph in figure 3.5, which represents a function call graph for an MP2 file decoding practical example, is a visualisation from Gprof [33], a performance analysis tool for Unix-based systems. It shows a diagram of function invocations and quantitative attributes that help reveal the decoder application's internal software units.

Contents on each element represent each function, identified by its corresponding name. The percentage at the top is the program's total running time used by the function and the related nested call(s).

The percentual value in parentheses below the previously mentioned is the time spent exclusively in the function, not including the succeeding invoked functions. At the bottom of the box, the number of times the function was reached is listed. The arrows between the boxes show the function relationships, and the arrow's thickness indicates the frequency or the time spent on these calls.

The audio specifications for the current example relative to the MP2 file are as follows: the total duration is 7 minutes and 14.15 seconds, and it is coded with a joint stereo (middle-side and intensity) channel mode. The bit rate is constant at 320 Kbps, and the sample rate is 44100 Hz. Each frame contains 1152 quantised samples, equating to a corresponding duration of 26122 microseconds, which gives a total of approximately 16633 audio segments.

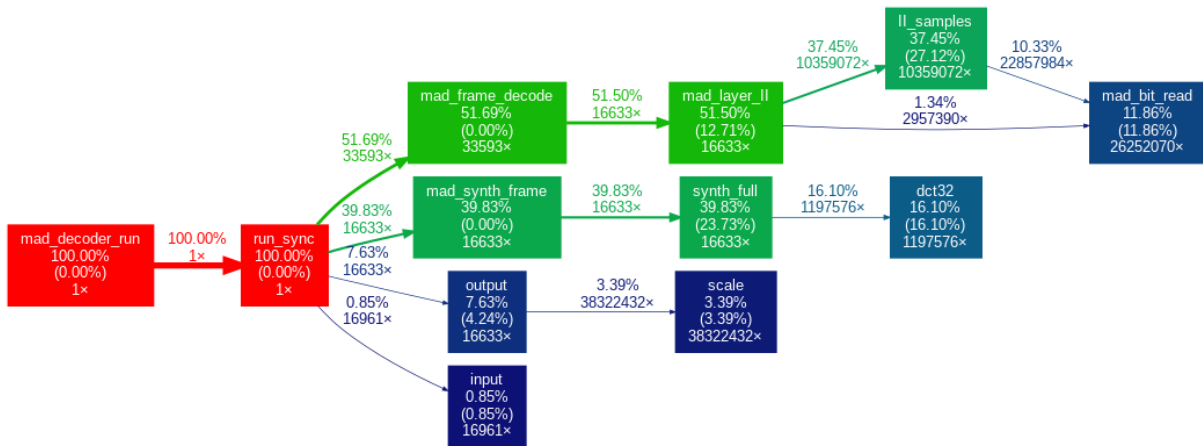


Figure 3.5: MPEG layer II decoder example with functions call graph

From left to right, the ramifications show how the reliance on lower-level functional routines is spread in the general decoding procedure. The *run_sync* function contributes to the majority of the execution time, considering that the two methods (*mad_frame_decode* and *mad_synth_frame*) that are used to decode the actual header and body of the audio frame are rooted from that point, that counts as the most significant part of the process.

Given that 91.33% is spent between *mad_layer_II* (51.50%), which have recourse to reading the bit-stream contents through the *mad_bit_read* method, and *synth_full* (39.83%), to conduct data dequantisation and inverse transformation from frequency subband components to the raw audio format, by using *II_samples* and *dct32*. The other section is related to the input and output callbacks, which convey data to be processed and the product of the decoding procedure. Coupling these two components, working reciprocally, the profiling analysis covers the most time-consuming fractions of code and interaction with the external device.

For the following example regarding the MP3 type decompression, the audio file specifications are defined as a total duration of 10 minutes and 43.24 seconds, an operating mode of regular left-right stereo, a constant bit rate of 320 Kbps and a sample rate of 44100 Hz. The frame contains 1152 quantised samples, corresponding to a duration of 26122 microseconds. This results in roughly 24624 audio segments in total. The following image 3.6 represents a function call graph for an MP3 file decoding practical example.

Based on the similarities with the previous example, the MP3 decoding process involves the same functions for the synthesis and decompression of audio frames. The difference is that the decoding component is more prominent than the synthesis, which is notable in terms of complexity, where most of the computational load is found due to the increased resolution and fidelity in the compression process.

The process also applies Huffman and scale factor decoding, which is essential for deconstructing the compressed data into a form that can be converted back into audio signals. Huffman coding allows for efficient data compression without loss of information, while scale factors adjust the decoded audio data to match the original dynamic range.

Moreover, MP3 decoding handles the potential for overlapping frames, frequency inversion, and alias reduction. Overlapping of frames can prevent audible gaps between frames, maintaining an ideal auditory experience. Frequency inversion and alias reduction are part of the process that ensures the output audio does not contain distortions that can arise from the encoding process.

Lastly, transform functions such as *dctIV* (Discrete Cosine Transform), *imdct36* (Inverse Modified Discrete Cosine Transform), *sdct11* (Short Discrete Cosine Transform), and *fastsdct* (Fast Short Discrete Cosine Transform) play critical roles in converting the frequency domain data back into the time domain. These transforms are essential in reproducing the high-quality audio that MP3 is known for, maintaining the integrity of the sound through various levels of compression and encoding.

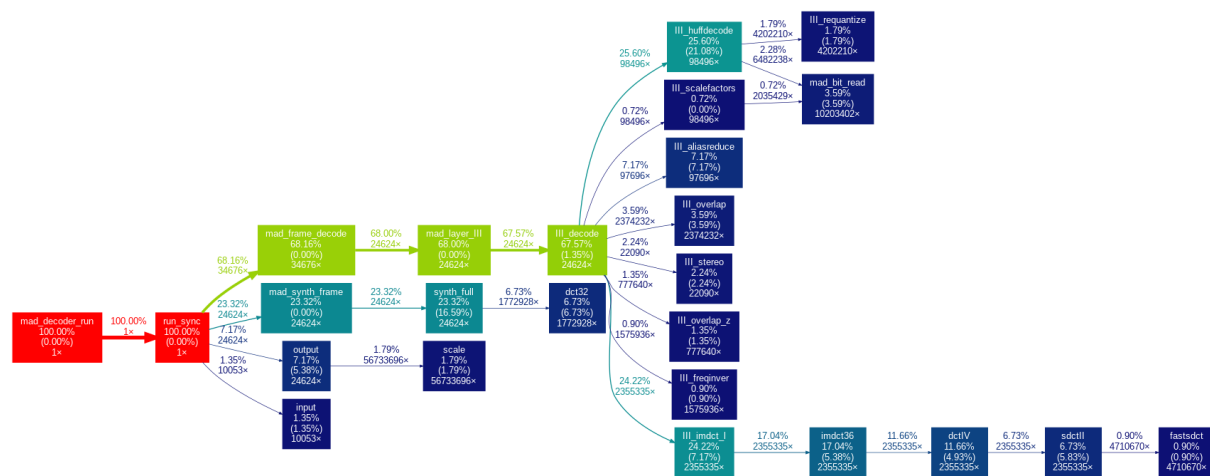


Figure 3.6: MPEG layer III decoder example with functions call graph

The algorithmic functional routines are supported by the most basic mathematical/ logical operations, with tailored implementations regarding the instructions available for several central processing unit architectures, such as SPARC (Scalable Processor ARChitecture), PowerPC (Performance Optimisation With Enhanced RISC–Performance Computing), MIPS (Microprocessor without Interlocked Pipelined Stages), ARM (Advanced RISC Machines), Intel and a 64-bit version (classified as the most accurate if the type is supported by the compiler, although not being the most efficient).

With this in mind, the profiling data and diagram serve as a valuable tool in identifying bottlenecks and ensuring that optimization techniques, such as assembly-level adjustments or algorithmic improvements, can be applied effectively across different CPU architectures. This allows MP3 decoders to remain versatile and adaptable, even as hardware evolves, ensuring high-quality audio decoding across multiple platforms.

In this project’s scope, the default 32-bit fixed-point method is implemented, taking advantage of some RISC-V extensions and CPU architectures as a conveyor of improvement and adaptability of the audio decoding system requirements to real-time operation [34].

A breakdown of the numerical format and the most frequently accessed arithmetic and logical operations is portrayed in the following subsection.

3.2.3 Fixed-point numerical representation

Fixed-point numerical representation is a cornerstone of efficient operation in embedded computing, particularly in environments where hardware constraints preclude the use of floating-point units. This approach is tailored for systems without a specialised floating-point unit and uses integer computation. Nonetheless, this necessitates a detailed consideration of certain compromises regarding the embedded system infrastructure since a tradeoff exists on the dynamic range of values, precision and performance. *LibMAD* exemplifies this approach by leveraging integer-based arithmetic to perform precise data processing. The figure 3.7 illustrates the numerical format.

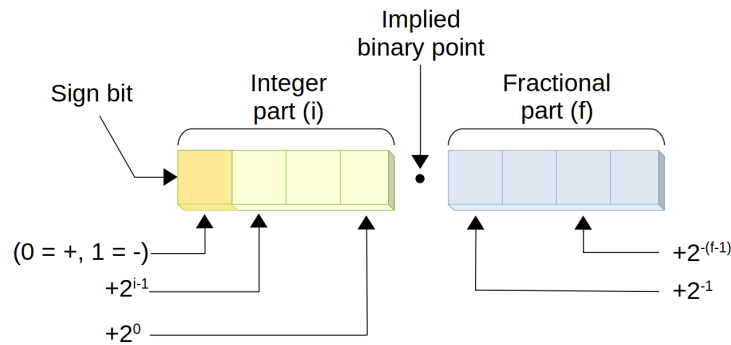


Figure 3.7: Fixed-point binary number representation

The fixed-point form is given by a collection of bits that define the integer and the fractional part of the number, in the example identified with i and f , separated by a radix point. A 32-bit format is used in the decoder and is characterised by the first four bits, including the sign bit; two's complement uses this binary digit with the greatest place value as the sign to indicate whether the binary number is positive or negative; the following three bits form the rest of the integer part, which are allocated for the integer part and the remaining 28 bits for the fractional part. Ranges are written as hexadecimal numbers for a succinct exemplification.

This structure represents integer values from -8 to +7, a total of 16 different integer values. Values are handled using two-complement notation. This approach simplifies arithmetic operations, especially subtraction, by eliminating the need for separate logic for handling negative numbers. The effective range is between the highest value (0x7FFFFFFF), representing the maximum positive number just below +8.0, while the lowest value (0x80000000) represents -8.0.

A minimum non-zero value that can be represented is 0x00000001 (i.e. about 3.725e-9), indicating that the system can differentiate minimal changes in value. With 28 bits dedicated to the fractional part, this fixed-point format achieves around 8.6 digits of decimal accuracy. Fixed-point numbers can be added or subtracted as normal integers, but multiplication requires a shift operation of the 64-bit result from 56 fractional bits back to 28. The elementary mathematical routines, defined as function-like macros, are described in the following table and can be used in distinct implementations, as mentioned in the previous subsection.

Operation	Description
ML0	Performs a multiplication and assigns the result to the lowest part of the operand
MLA	Adds the result of multiplication to the least significant part of the operand
MLN	Negates the value of a given variable
MLZ	Casts a variable to a specific fixed-point type and discards its highest part
SHIFT	Bitwise shift to move the bit values of a binary object

Table 3.1: Arithmetic and logic operations used in the decoding & scaling algorithms

3.2.4 Procedure in the context of the IOb-MPEG-Decoder

This topic presents the characterisation of the inclusion of the decoder software into the firmware program that is stored in the SRAM memory fragment or external DDR memory, depending on the chosen CPU architecture, combined with the expected driver functions to handle the communication peripherals according to the response of the decoder software unit.

During the implementation of the audio codec decoder component along with the embedded system software, several distinct alternatives were considered relative to the input and output communication with the active elements of the decoder.

The decoder application's submodule directory must be appended to incorporate the source files and the included references. The program is embodied with a specific software section, the firmware, on which there will be the declaration of the required functions and structures as well as the necessary device drivers to create interaction with extrinsic hardware peripherals.

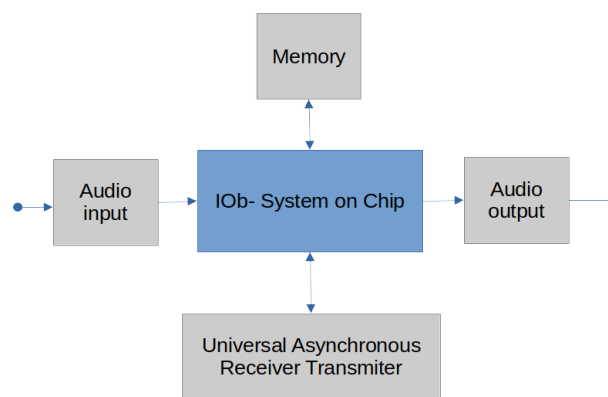


Figure 3.8: High-level SoC diagram with communication modules

Once compiled, the program is loaded into a designated memory section carefully addressed to align with the SoC's operational flow. This precise positioning in memory is crucial for optimal performance, as it allows for efficient access and execution of the decoding program. Compiling and placing the application within the SoC's memory ensures seamless integration and operation.

Audio bitstream decoding is a core function of this system, involving converting compressed audio data into a format suitable for playback or further processing. This process requires careful handling of the audio bitstream, adhering to MPEG standards to accurately decode the data. The decoder must efficiently process various aspects of the audio bitstream, such as decoding frames, handling error conditions, and maintaining synchronisation with other audio elements. Precision in decoding is paramount to sustain audio fidelity and ensure a high-quality output.

Integrating peripheral interfaces with the MPEG Audio Decoder (MAD) high-level Application Programming Interface (API) is essential for robust data transport, scaling, and error signal. This integration facilitates seamless communication between the SoC's core processing unit and its peripheral devices. Through the MAD API, data transport is streamlined, ensuring that audio data is efficiently transferred through the system. Scaling capabilities allow for dynamic adjustment of audio parameters, while robust error information mechanisms ensure that any issues are promptly identified and addressed, maintaining system integrity.

The system is designed with several key requirements in consideration:

- **Minimum latency** Achieving the lowest possible latency is crucial for real-time audio processing, involving optimising the data pathways and ensuring that the decoding process is as efficient as possible to minimise delays between input and output. A minimum of 1 audio frame is expected, since the time between the first frame being loaded to the stream structure being decoded and consequent decoded frame products being transferred at the output.
- **Support for MPEG audio standards** Full compliance with MPEG audio standards is mandatory to ensure compatibility with a wide range of audio formats and to guarantee high-quality audio output.
- **Configurable input and output channels** Flexibility in configuring input and output channels allows the system to adapt to various audio setups and requirements, including synchronising with other audio sources to ensure cohesive operation in complex audio environments.
- **Real-time operation** The system is optimised for real-time operation, ensuring that audio decoding and processing occur instantaneously to facilitate live applications without perceivable delays.
- **Testing and validation** Testing routines are implemented to evaluate every aspect of the system, from individual components to the entire SoC, including functional testing of the decoding process, performance testing under various conditions, and validation against industry standards. Validation ensures the system meets the design specifications and performs reliably in real-world scenarios.
- **Extensibility and future-proofing** A modular design approach allows individual system components to be updated or replaced as needed without affecting overall functionality. Embracing open-source technologies and standards promotes a collaborative development model and reduces dependency on proprietary solutions. This approach fosters innovation and ensures the system remains adaptable and scalable to future technologies and standards.

a) Integration strategies

In the interim of the project development, several options were addressed to gradually implement and test the audio processing application within the bare-metal system environment. Initially, the firmware was built with a simple readout of the MPEG test files from a specific memory region, using UART to load the audio data to a particular region in memory reserved for that purpose and subsequent storing of the resulting PCM audio samples, written as raw audio data content in a contiguous memory region, accomplished using dynamic allocation of memory size, taking into account previous knowledge of the input file length and by estimating the size of the decompressed audio file from the sampling frequency and bit-rate. However, this approximation based purely on its bit-rate can be inaccurate because audio compression can vary the amount of data used to represent a segment of audio regarding the different MPEG layers files might have a VBR encoding configuration, making simple bit-rate calculations not always reliable for determining the exact length of an audio track. The padding bit must be accounted for since this flag might bring additional content to the audio frame, increasing its size.

The problem with this approach is the misuse of memory space, which could be critical. Relying on the size of both input and output files leads to a system state dependent on these variables. It reflects a lack of scalability since other components might need the referred memory blocks. Because of this, a polling mode was implemented with an external communication interface connected to a tester module responsible for loading and storing the referred audio files and comparing the results with decoded examples to verify data integrity throughout the process. Ensuing tests were performed with iterated SoC versions using UART and AXI Stream interfaces.

The first step was developing a simple application programming interface to achieve a series of preliminary assessments before being ported to the IOB-SoC-MPEG-Decoder. Secondly, the decoder program is integrated with the system-on-chip per se and within the scope of the design deployed in FPGA hardware, running remotely on a server. Finally, the evaluation tests, written to estimate the implemented algorithm's computing performance using different audio files, are supported by an embedded tester platform provided and previously validated by IObundle.

The data transport mechanism is vital in the compressed audio decoding system's structure. Peripheral drivers are essential to any system that links with external hardware devices. They act as a connection between software commands and physical hardware operations. A device driver is a specialised interface allowing higher-level computer programs to interact with a hardware device. This interaction enables the software to perform actions such as initiating data transfers or controlling memory accesses. In the audio decoder's case, these drivers allow the software to articulate with the memory transferences, input and output interfaces, and central processing units with distinctive instruction set extensions. This communication is crucial for tasks such as fetching audio files from memory or a device on the network and buffering decoded audio data.

Once the software simulation is verified, testing the system with the full-scale SoC implemented on the FPGA is fundamental. It is necessary to port the software application to the embedded systems' physical domain, scoping the constituent components to be remodelled according to the differences in architecture types and peripherals to handle incoming and outgoing data from the decoder. This simulation is used to verify the system's functionality in a controlled environment, study the conditions, and determine if the system meets the desired benchmarks. It helps identify and rectify any potential problems in system design or interaction between various components and ensures the system's boot management.

b) Firmware customisation

This section provides a more specific view of software adaptation related to the MPEG decoder and its interoperability with hardware peripherals in the proposed embedded system.

The author created the firmware sourced from the template provided by IOBundle and the *minimad* program example in the LibMAD repository. In this software segment, the necessary structures and functions that constitute the MPEG audio decoder API are instantiated, including the communication components' respective driver functions.

The firmware is converted into machine code instructions, and the embedded system processor's binary codes can directly interpret and execute.

$$\log_2(\textit{Firmware Binary Size}) = \text{SRAM_ADDR_W}$$

This indicates that the width defines the number of addressable units required in the memory to store the entire binary, reflecting the firmware's memory footprint.

After the bootloader's initial setup, the firmware is loaded into SRAM memory. This process involves copying the firmware memory configuration using a UART16550 serial communication interface, making it accessible for execution. At this designated memory address, the CPU subsequently executes the program.

Handling audio files in the MPEG audio decoder involves several key steps. Firstly, audio files are polled from the connected digital audio interface and stored locally in an input buffer or directly fetched from the system's memory, which involves reading a portion of the audio file from memory to be processed by the decoder. Once decoded, the audio data is returned to the memory or the output interface.

Integrating API functions and attributes with firmware components is a primary aspect of embedded software development paradigms. Using high-level APIs in such integrations regarding streamlining development processes ensures efficient utility development, taking advantage of software as building blocks.

In this context, the abstraction provided by the high-level API refers to the way the MPEG audio decoder is implemented, separating the details from the underlying software, allowing the firmware to use the decoding logic without managing library-specific information, which is the main reason for choosing the low-level API, although coming at the expense of less detail and control over the decoding process, where each step must be handled explicitly so the procedure becomes more straightforward. The following methods characterise the programming interface that is instantiated utilising the `mad_decoder` structure, which is constituted by function pointers for each callback function:

- **Input** The input reference is used to re-fill the stream with unprocessed data during the decoding process, reading audio content from the *AXI4 Stream In* peripheral, which can be done using a semi-parallel approach since the input buffer is packed with MPEG frames fetched from the interface FIFO while the underlying process is occurring or from previously allocated memory that is done synchronously: data is carried into the stream until the frame is complete, then decoded and sent to the respective output.
- **Header** During the decoding sequence, the MPEG audio bitstream is interpreted according to the contents organised at the beginning of each frame. The digitised sound signal might have different properties related to variable bit rate and frame size, which should be considered when synthesising frame subband samples. This callback function allows for the evaluation of general information about the audio file segments.
- **Scale** With the scaling utility callback, the decoded audio passes through post-processing of rounding, clipping, and scaling samples down to 16 bits. Dithering or noise shaping is not performed, which would be recommended to obtain exceptional audio quality, such as the 24-bit PCM output. This routine is, therefore, conditional.
- **Output** The output function is invoked after each frame of MPEG audio data has been completely decoded. This method transfers the decoded PCM audio using the *AXI4 Stream Out* peripheral or stored to a predefined memory region that can be sent to a file using the UART interface after the whole process is complete.
- **Error** Whenever a decoding error occurs, this callback function is called. Examples might be a header lacking data integrity, a frame with corrupt data, insufficient memory space or an incompatible input buffer pointer. The hazard is displayed by the reference of the stream structure, indicating the `mad_error` code enumeration; the list of possible error types can be found in the `mad.h` header file about the decoder source files.

The firmware is structured around managing input and output data streams using several buffer sizes and manipulating audio data at various decoding stages. Several parameters are defined to manage these data streams, including `INPUTBUFFERSIZE` and `NBR_BYTES`. The `INPUTBUFFERSIZE`, set at 4096 bytes, specifies the buffer size used to store incoming encoded audio data before processing. Meanwhile, `NBR_BYTES`, set at 1024, determines the standard chunk size of data to be processed in each operation cycle. The figure 3.9 describes a model of the firmware sequential operations.

The handling of consumed and unprocessed data is critical during the decoding process. These operations utilise the `input_buffer`, a buffer array where incoming data is stored and gradually shifted as it is processed. The firmware keeps track of how much of this buffer is unprocessed versus how

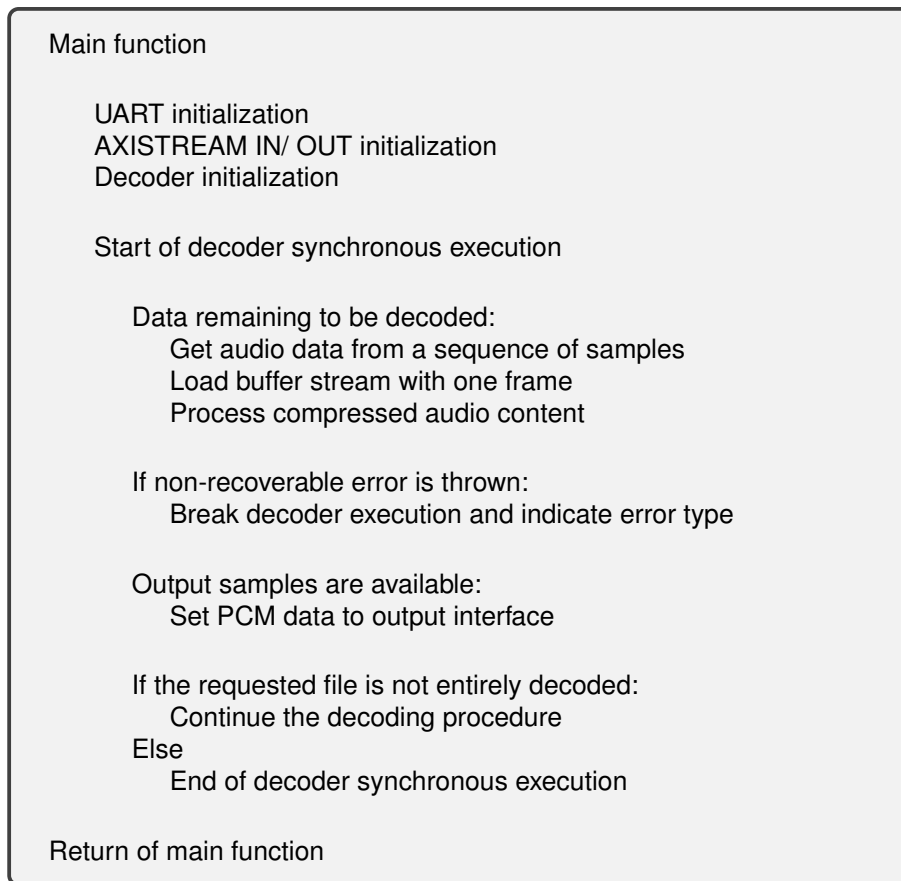


Figure 3.9: Decoder firmware pseudocode

much has been consumed (i.e., already decoded), allowing for efficient data flow management through the decoder, accomplished by manipulating the `input_buffer` based on the amount of data processed and shifting unprocessed data to the beginning of the buffer for subsequent decoding cycles.

The firmware also employs input and output offset counters to track the progression of decoding. These counters help manage the position within the input and output buffers, ensuring data integrity as chunks are processed and outputted. A critical aspect of this system is the setting of `MAD_BUFFER_GUARD` zero bytes at the end of the input buffer. This operation is crucial to prevent buffer overrun errors by marking the end of valid data, which is particularly important in real-time decoding scenarios where data boundaries must be clearly defined.

Regarding the audio data, the firmware is designed to decompress audio encoded in the MP3 format and convert it into a PCM (Pulse Code Modulation) format. This conversion process handles 24-bit PCM samples, although the internal processing within the LibMAD library operates at a higher resolution, which is later scaled down. The output buffering considers that each PCM word is 8 bits, structured to accommodate stereo audio by processing two channels simultaneously.

The output routine ensures that decoded data is appropriately formatted into stereo or mono PCM samples and pushed into the output stream, typically an AXI4 Stream Out peripheral in an embedded environment. This systematic processing of audio samples, frame by frame, contributes to the overall efficiency and effectiveness of the decoding process, ensuring that the audio output maintains high fidelity relative to the original encoded format.

When using LibMAD, not all bytes from the input buffer may be consumed. If `mad_frame_decode()` detects an incomplete frame, the start of this incomplete frame is indicated by a non-null pointer in

the **stream** structure. This is a common issue and can be managed by shifting unused bytes to the beginning of the buffer for the next refill. The buffer size should be large enough to accommodate the largest possible frame at the highest bit-rate of 320 kbps and sampling frequency at 44,1 KHz, with 1152 samples per frame, equivalent to approximately 1044 bytes.

Decoding the last frame of an audio file requires an additional MAD_BUFFER_GUARD of zero bytes following it, ensuring the frame is fully decoded. This guard is necessary because, particularly with Layer III decoding, there's a potential for Huffman decoding routines to read slightly past the buffer end with certain kinds of invalid input. The presence of these extra bytes, which do not cause buffer overflow due to careful allocation in buffer management, prevents issues and maintains the accessibility of data needed for beginning the next frame.

Recoverable errors due to corrupted data streams allow for continued decoding attempts by skipping the corrupted parts and synchronising with the next frame, thus managing minor data integrity issues without halting the decoding process. In case of a non-recoverable error, all decoding should stop, except when the error is MAD_ERROR_BUFLEN, which indicates that additional input is required. The buffer should then be refilled, and decoding should resume, ensuring any remaining bytes at the buffer's end are moved to the front. This approach effectively handles incomplete frames.

As presented in section e), the primary parameters for the SoC configuration used in the IOB-MPEG-Decoder are defined to optimise performance and memory usage. The parameters are summarised in table 3.2.

Parameter	Value
INIT_MEM	1
USE_EXTMEM	1
USE_MUL_DIV	1
USE_COMPRESSED	1
BOOTROM_ADDR_W	12
SRAM_ADDR_W	19
MEM_ADDR_W	24

Table 3.2: SoC configuration parameters for the IOB-MPEG-Decoder

The SoC operating mode and the components from configuration options play a structural role in system functionality and flexibility. The INIT_MEM = 1 setting allows the system to store a program received via UART and initiate execution from this program or from the pre-loaded booting setup, firmware, and data binaries into a specific memory type such as SRAM or DDR. This configuration enhances the adaptability of the startup process.

Furthermore, the USE_EXTMEM = 1 option activates the utilisation of DDR memory, providing alternatives for memory layout and specifying the address location for the IOB-MPEG-D software. This choice determines whether the program runs from internal or external memory within the IOB-SoC, thus influencing the overall performance and scalability of the system.

Enabling both USE_MUL_DIV = 1 and USE_COMPRESSED = 1 configures the SoC to support extended instruction sets for multiply and divide operations on integer registers and to utilise compressed RISC-V instructions. This setup optimises processing efficiency and reduces the amount of memory the application uses.

In the software development environment, defining constants via C language preprocessor directives is crucial for configurable compilation. Macros, integral in the make command for targeting software compilation, are defined as IOB_SOC_MPEGD_FLAGS, presented in table 3.3. These flags include MAD_DECODER_MODE_SYNC for synchronous thread execution, FPM_DEFAULT for selecting the default fixed-point math option, OPT_SSO for optimising fast subband synthesis, which is mandatory

Configuration option	Description
MAD_DECODER_MODE_SYNC	Synchronous thread execution
FPM_DEFAULT	Fixed-point math option default
OPT_SSO	Fast subband synthesis approximation optimisation
NDEBUG	Define to disable debugging assertions
SCALING	Enable/ disable scaling option for high-quality audio output
AUDIO_IN	Input audio file name
AUDIO_OUT	Output audio file name

Table 3.3: Description of configuration options for the compilation flags

because of the expected loss of speed and accuracy when using the default implementation, and NDEBUG to disable debugging assertions during production runs. Additionally, the constants AUDIO_IN and AUDIO_OUT are used to specify the names of the input and output audio files, respectively, facilitating the definition and identification of the testing samples.

c) Profiling

Profiling in the context of an MPEG audio decoder running on a SoC with a RISC-V CPU entails performance analysis, mainly focusing on processing speed, memory usage, and power consumption. This process is critical for identifying bottlenecks and inefficiencies and gives an insight into their applications' internal workings. By understanding where the application spends most of its time or consumes the most resources, it is possible to optimise code and improve performance, ensuring it operates within the desired parameters and constraints.

The audio decoding procedure's functions are the essential portions of software code that should be given context for the particular use of profiling and diagnostic tools. Monitoring the execution of these functions provides insights into how each contributes to the overall workload. By profiling these functions, areas for optimisation can be identified, such as speeding up data processing routines or reducing memory usage, enhancing the efficiency of the MPEG audio decoder.

- **Debugging** Trace and debug capabilities are vital in bare-metal embedded systems. These tools provide visibility into the system's operation, allowing for code monitoring, inspecting memory and register states and tracking the data flow, essential for identifying and resolving issues at the system's hardware and software levels, ensuring the audio decoder functions correctly in all scenarios. For instance, valuable information about the state of the decoder process and memory acquisition operations, audio file specifications and frame metadata, stream buffer loading, and consumed and unprocessed data verification.
- **Temporal tracing** Profiling within this SoC can be observed from two primary scopes: input/output performance and functional execution. Input/output profiling focuses on how efficiently the system handles data transfers between various components, such as reading audio files from memory and writing decoded output back. This profiling helps optimise data handling routines for speed and efficiency. On the other hand, profiling the functions used in the decoding process involves analysing the time and resources consumed by different functions. Table 3.4 provides an overview of the configuration parameters and their descriptions, which are essential for profiling different layers and interfaces of the decoding system. These parameters, such as API.PROFILE, PROFILE.I, and PROFILE.II, enable targeted analysis of specific components within the system.

The mentioned feature involved creating two files, `profiling.c` and `profiling.h`, which are written to encapsulate a set of functions and structures to track the elapsed clock cycles for individual

Parameter	Description
API_PROFILE	Application programmable interface profiling
PROFILE_I	Layer I & II profiling instances
PROFILE_II	Layer III profiling instances

Table 3.4: Configuration parameters and descriptions for profiling

functions. This profiling system includes three methods that collectively manage the profiling data and ensure performance metrics after the initialisation for the peripheral driver that is accomplished in the firmware, with *timer_init* function relative to the timer base address.

Profiling methods are defined such as *initProfiling*, which resets the five profiling array instances, *promptInfo*, to display the profile intervals, and *resetArray*, takes as arguments a pointer to a profiling group to pass by reference along with the size, so the contents of the array that represent the lapsed time for each function instance accessed during the execution to be set to zero, and sets the timer counter to zero, with *timer_reset*.

For that effect, a union of fixed-size arrays declared as external variables of the type `uint64_t` within both the profiling source and header files is included by the firmware source and the `global.h` pertaining to the LibMAD repository, which is included by its constituting files, defines conditional debugging and features used by the decoder. Each array corresponds to a specific section of the software stack. This organisation reflects the common functionalities across different layers, as observed in the files `bit.c`, `frame.c`, and `synth.c`, and also includes unique functionalities found in `layer12.c` and `layer3.c`.

Operationally, the profiling approach involves iterating over the union structure, which holds a pointer to an array associated with each profile group. This array is instantiated with a pre-defined fixed size to account for each file's total functions. Code coverage involves indexing each array element to a function currently being monitored, and the measurement of execution time is recorded from the moment a function is called to when it returns, using *timer_get_count* to get the timer count for the elapsed clock cycles of the system running at a specific frequency and compute elapsed time. The time is measured cumulatively in functions invoked recurrently in frequency inversion and synthesis to account for the total elapsed processing time per frame. Furthermore, the profiling approach is non-intrusive; the profiling data is printed only after each frame's decoding is completed. This timing ensures that the printing process interferes minimally with the ongoing data streaming or the subsequent frame processing.

In addition, an execution profile specifically for API-related functions has been established. This profile offers a distinct view into the software's internal workings, providing information without impacting the lower-level procedure. By doing this, it is possible to define and constrain the data delivery between the firmware and communication peripherals in a speed-independent way, considering the worst-case execution time between consecutive frames as the time measurement. With this examination, it is possible to infer the correct functioning of the decoder concerning the coupling of both software and hardware domains.

Chapter 4

System Verification

From the onset of the requirements research, a plan is devised that considers the expected performance metrics and hardware limitations of the system related to the application prototype. The validation phase is an essential stage of development workflow before deployment. This testing phase ensures the specifications accurately align with the final product. Moreover, it plays a pivotal role in improving the system's overall quality and identifying and mitigating potential errors across all aspects of the architecture, contributing to reliability and efficiency in real-world scenarios.

The intense product competition in system features and capabilities caused by the increasingly demanding markets drives the continuous development of embedded systems technology. To be competitive, new designs must exhibit upgrades in functionality, performance, and reliability and declines in features like power consumption and size, hence the necessity of a robust infrastructure to support thorough system verification [42].

Verifying the system design is accomplished by iterating from software emulation based in an operating system environment to the next simulation step in the context of the FPGA synthesisable model. Focusing on an accurate hardware environment is valuable because of timing constraints and relative interoperability amidst software/ hardware peripherals and analysing resource utilisation. With this approach, it is possible to predict the behaviour of the software application more precisely and how to work efficiently with the unique architectural features, including specific input/ output components and optimisation alternatives. Furthermore, this step is essential for resolving data bottlenecks, synchronisation problems, and interface mismatches, often not apparent in a purely software-based emulation.

Throughout these steps, the high-level API of the audio decoder is incorporated with the system's firmware and compiled according to the set of definitions that allow the application to run on the FPGA, which in turn is programmed with the description of the SoC based on a *RISC-V* central processing unit. After the simulation, the built-in application firmware is therefore executed on the target machine to confirm the correct functioning of the MPEG audio decoder integrated with the built-system design.

The decoder application must handle various content complexities, dynamic ranges, and detail levels in different audio segments with accuracy as close as possible compared to the pure-software option. This aspect is particularly challenging given that the speed features present a significant contrast between a pure software implementation and the proposed system. While regular computers with general-purpose CPU cores, in a broader sense, are designed for a wide range of tasks with higher clock speeds, computing systems that are built with application-specific purposes offer efficiency but with less flexibility to perform other tasks, typically with inferior clock speeds. Therefore, optimising the decoder to leverage the strengths of this architecture, such as its customizability and efficiency, while compensating for its lower raw processing power is essential for maintaining high performance and accuracy in diverse audio processing scenarios.

4.1 Testing platform

The IOB-SoC-Tester [43] component standardizes the testing environment, whose purpose is to run tests on hardware prototypes. In the verification stage of the development project, this tester provides a base system that developers can adapt for their specific application, removing the dependency for verification under simulation via HDL test benches, which can be time-consuming.

Implementing Continuous Integration/ Continuous Delivery (CI/ CD) in embedded systems presents unique challenges due to the direct interaction with hardware layers and the need for specialized testing frameworks to simulate hardware behaviour accurately [44].

The system addresses these challenges by providing a standardized platform to integrate into CI/ CD pipelines, improving the existing development process. Implementing such services aims to enhance development workflows by automating testing and deployment tasks.

The system manages the task queue in a CI/ CD context by handling requests and processing them in an ordered manner, as facilitated by the Ethernet LAN connection and the multiple ports indicating various interaction points or tests. This setup underscores the relationship with the "host machine," the central server coordinating multiple tasks, including software builds, test execution, and deployment operations. The tester is based on IOB-SoC, which has a similar build environment and uses compatible peripherals. The Tester includes the Unit Under Test and controls its I/ O, as illustrated in Figure 4.1.

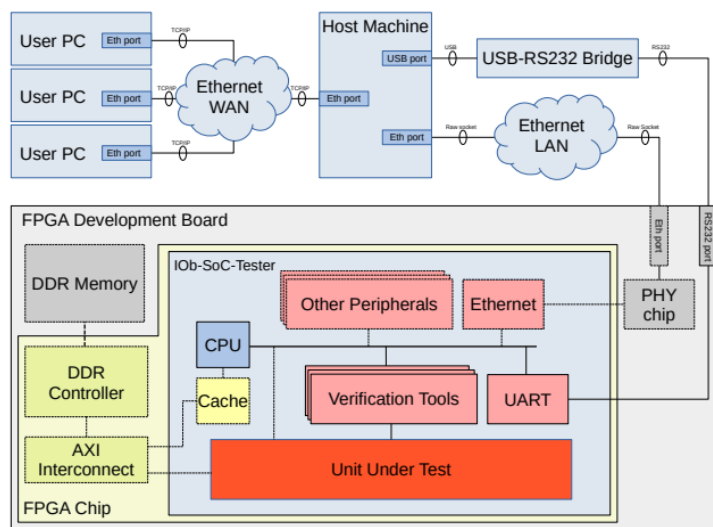


Figure 4.1: Test environment infrastructure diagram
Source: IOB-SoC-based Tester System [43]

From the development user's standpoint, communications are established via Secure Shell (SSH) with the available processing resource, and the results are promptly displayed in the user's command-line interface (CLI). The diagram illustrates the use of network interfaces to establish robust communication with the processing resources, ensuring seamless data flow and efficient management of development processes.

The depicted FPGA development board within the IOB-SoC-Tester framework aligns with these concepts, where integration and functional testing are conducted. The interaction between the FPGA chip, DDR memory, and other peripherals is scrutinised for cooperative functionality as part of the integration testing, highlighting the system's ability to simulate complex hardware interactions.

When considering the test environment, the notions of a hardware/ software testbench and sandbox emerge. A testbench is often a virtual environment used to verify the logic of digital circuits, including SoCs, at the hardware level. This environment can include simulation models that replicate the execution

of hardware components, enabling a thorough examination before physical prototypes are created. On the other hand, a sandbox can be considered a safe, isolated environment where untested code or experimental features are executed without affecting the primary system, commonly used in software development to test user interface changes and new functionalities.

The Unit Under Test (UUT) is a digital system under development that requires verification. It may be a core, an SoC, a peripheral or any other digital system. IOB-SoC-based systems generally include most of these requirements except for the Tester-related files.

In this project, the IOB-MPEG-Decoder serves as the UUT, integrating the MPEG audio decoder into the testing framework provided by the IOB-SoC-Tester and integrating the IOB-MPEG-Decoder as the UUT involves embedding the decoder into the existing IOB-SoC-Tester infrastructure. This process requires aligning the decoder's interfaces with the tester's input/output protocols to ensure seamless communication between the tester and the UUT.

In software testing, various approaches are categorised by the level of access to the system's internal workings:

- **White-box:** Also known as clear or transparent testing, it involves thorough knowledge of the internal logic and structure of the code.
- **Black-box:** Assesses the functionality without any insight into the internal code structure, focusing solely on inputs and outputs.
- **Grey-box:** Is a blend of both, where some internal knowledge is utilised, but the focus remains mainly on external functionality.

The Tester can only control the UUT's input/ output. Because it lacks access to the internal signals, it implements the black and grey verification methods that are incompatible with the white-box verification methodology.

Integration and functional testing are then applied to individual components to validate their adherence to specifications without delving into their internal architectures, hinting at a black-box testing approach. In this context, it involves the exchange of enquiry and control signals, with the system being tested and its feedback analysed. For the IOB-SoC-Tester, these signals are communicated via the UART interface, and commands are defined to control the action of the low-level component. The diagram also indicates a USB-RS232 bridge, suggesting that legacy serial communication stanith pre-defined conditions and monitor its behaviour and performance.

To support the IOB-MPEG-Decoder as the UUT, firmware testing anansport mecd data trhanisms are implemented to facilitate the transfer of test data and contdards are accommodated within the testing framework, ensuring compatibility and comprehensive testing across different communication protocols. Each of these interfaces serves as a conduit for test signals and responses, allowing testers to methodically probe the system under test wrol commands to the decoder system. This involves developing firmware that can interface with the UUT, handle data buffering, and manage communication protocols necessary for the decoder's operation. The testing framework ensures that the decoder receives the appropriate input data streams and that its output can be captured and analyzed for correctness.

In summary, the testing platform leverages the IOB-SoC-Tester to provide a robust environment for software-on-hardware testing. The platform verifies the system's functionality through its external interfaces using black-box and grey-box testing methodologies.

4.1.1 IOb-SoC-SUT

The IOb-SoC-SUT [31] platform is based on the IOb-SoC template to expedite testing on integrated systems, using an example system as a peripheral or another primary structure and software applications on hardware prototyping.

This system showcases the Tester's communication abilities and usage with an SoC UUT as its peripherals. Therefore, it features an optional ETHERNET peripheral, which can be used as a high-speed data transmission interface, a UART serial interface, and external DDR memory.

It encloses an integrated SoC in development into the tester wrapper as a system under test and manages the flow of information between both modules. This way, it is possible to introduce an impulse and collect the system under test response and status.

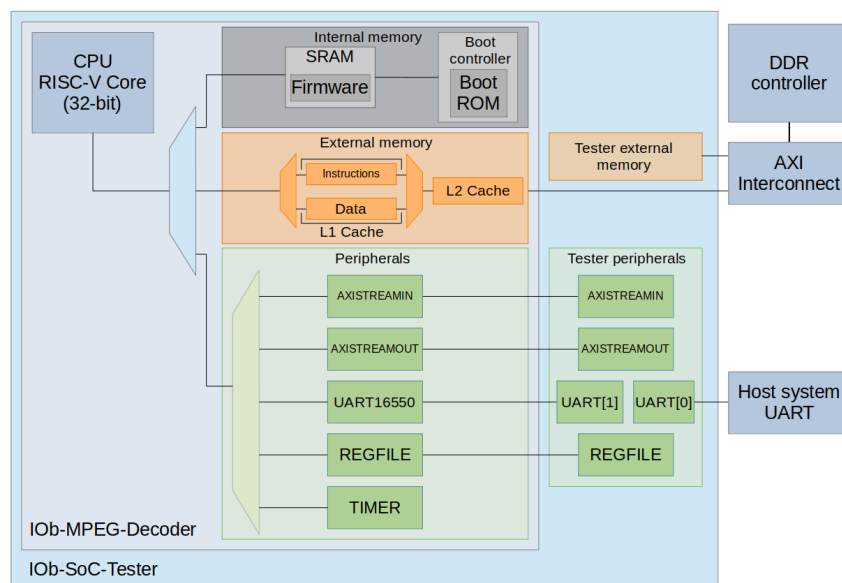


Figure 4.2: IOb-SoC Tester & MPEG Audio Decoder as unit under test
Source: IOb-SoC-based Tester System [43]

The SoC Tester integrates the MPEG-Decoder SoC and utilizes AXI Stream In/ Out to send and receive data from the verified system and the UART for configuration management and communication with the implemented SUT.

The modular firmware architecture supports the tester and the SoC's distinct hardware and software directories. The system leverages DDR memory shared between the SUT and the Tester to enhance data handling capabilities. Each system incorporates one AXI Interconnect for internal communications and an additional one for optional external memory interfaces. The tester includes two UART components; one connects to an external interface, and the other establishes communication with the SUT, facilitating audio channel management and control.

The UART linkage between the Tester and the SUT is crucial for sending control commands and receiving status updates, enabling synchronized operation between the two systems. The shared DDR memory between the SUT and the Tester is a shared storage space for audio data and configuration files, streamlining data exchange and reducing latency. The following figure 4.3 shows a pseudocode format of the tester firmware.

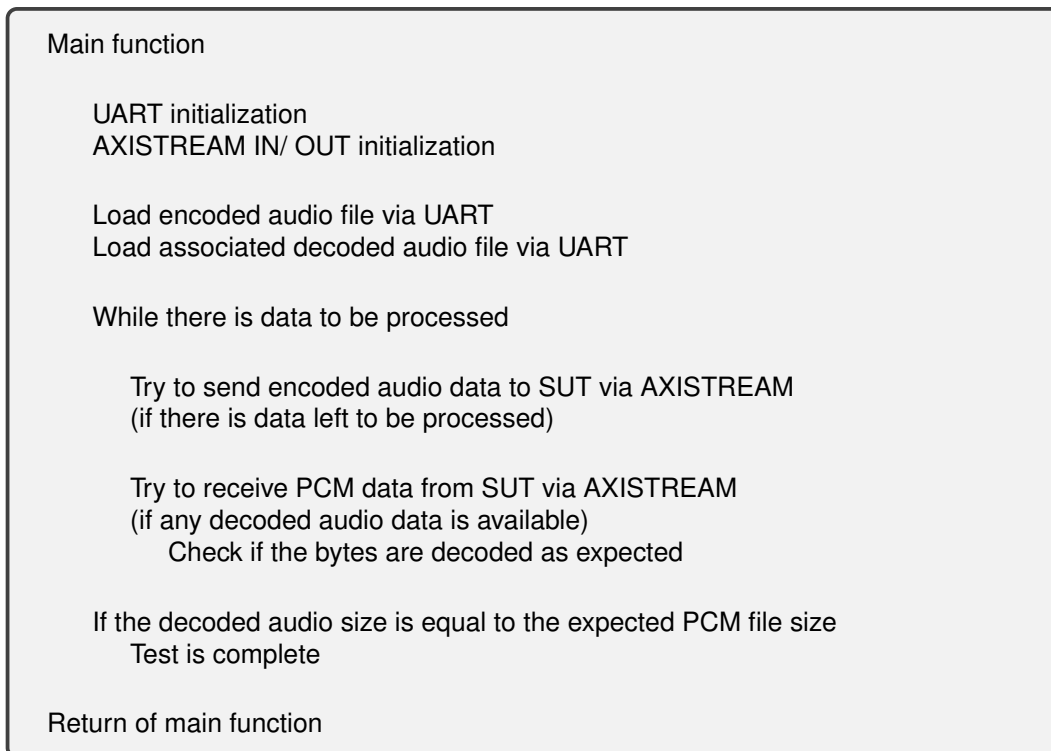


Figure 4.3: Tester firmware pseudocode

In the context of the IOb-MPEG-Decoder being used as a unit under test, two interfaces exist: UART0, which is attached to the referred module, and UART1, which is instantiated by the tester system, running in a bare-metal environment without an operating system. Therefore, SUT initialisation and enquiry/ control are led through the tester firmware, which is implemented to manage interactions between both systems. During operation, the UART interfaces listen for specific commands or send commands to the SUT or external systems. For instance, upon receiving the ENQ command from the SUT, the tester will respond with an ACK, signalling readiness to proceed with data transmission. There is also a possibility of initialising memory from the tester by uploading the configuration file as the bootloader does. The communication is established intermittently since the tester waits for SUT messages to be transferred until the *test.log* file is transferred.

The UART communication follows specific protocols, including control signals like FTX (File Transmission), FRX (File Reception), and EOT (End of Transmission), to manage data transfers effectively. In polling mode, the decoder is always ready to receive more data, continuously checking for incoming encoded audio. The AXI Stream I/O interfaces send and receive the audio files in partitions, allowing large files to be processed in manageable segments. This partitioning facilitates smoother data flow and reduces the chance of buffer overflows. To manage data transmission between the Tester and the SUT interface, the driver functions are used:

- **axistream.in_pop** Pop or retrieve data from the input buffer. It extracts a 4-byte value from the input stream FIFO and stores this value in a byte array provided as an argument. The function also determines and returns the number of valid bytes within this array. The significance of the valid bytes depends on the data width and whether the current value represents the last word of the current frame. A secondary return value indicates if the popped word is the last word of the frame, affecting the count of valid bytes.
- **axistream.in_empty** Signal when no more data values are present in the input buffer. It acts as a status checker for the input buffer's emptiness. When invoked, it returns a Boolean value indicating whether the buffer is empty (true if empty, false otherwise).

- **axistream_out_push** Pushing data into the output buffer and managing how data is transferred into the stream FIFO. It accepts three parameters: a byte array containing the data to be pushed, the number of valid bytes in the array, and a boolean flag indicating whether the current push operation includes the last word of the current frame. If the pushed data marks the end of a frame and there are remaining valid bytes in the buffer, the function adjusts the write strobe (WSTRB) according to the data width and the number of valid bytes, ensuring that only the valid parts of the buffer are transmitted.
- **axistream_out_full** Check the status of the output buffer to determine if it is full and unable to accept more data values. When invoked, it returns a Boolean value that indicates whether the output buffer is filled (true if full, false otherwise). This function is essential for managing the data flow to prevent buffer overflow and ensure efficient data transmission.

4.1.2 Audio samples evaluation set

Using different audio features in the testing phase ensures reliable operation across variable inputs, which is crucial for proper operation in distinct appliances. The decoder should be able to manage corrupted or incomplete data streams and maintain steady operation under various conditions. The table 4.1 contains the information about the test file attributes used in the project.

	Audio file example			
	Noise	Speech	Classical	Jazz
Bitrate [bps]	192000	80000	320000	320000
Samplerate [Hz]	44100	16000	44100	44100
Channel mode	Single	Single	Stereo	Stereo
Number of frames	383	556	9825	16634
Frame playing time [μs]	26122	72000	26122	26122

Table 4.1: Audio file examples and respective specifications

The software tool Audacity [45] was used to cut samples and verify audio in waveform and frequency spectral plots. Audacity provided a convenient way to prepare the audio samples and visually inspect them to ensure they contained the desired characteristics for testing.

Short versions of the test files were used in the simulation environment to reduce the space needed in memory for encoded and decoded audio data. This approach allows for efficient testing without the overhead of handling large files, which is particularly important in resource-constrained environments.

For example, pure sine wave tones are less complex to decompress than random noise due to their simplicity, lower entropy and variability, and higher predictability, which leads to better compression efficiency. In contrast, random noise, which has a high entropy and is less predictable, is more challenging to compress and decompress efficiently. Higher complexity in these areas increases decoding duration due to the need for more information processing per unit of time. This also applies to bit-rate allocation, where increasing values reflect on more data used to represent the audio signal in the same interval. This results in better audio quality since there are more bits to represent subtle details in the audio signal accurately, but it increases the cost of the decoding process. When maintaining a high signal-to-noise ratio, compression algorithms might allocate more bits to accurately represent subtle details in the audio signal, increasing the data size and processing requirements, and with the compression efficiency, it reduces the amount of data needed without significantly compromising audio quality. This means that there is a trade-off between speed and accuracy, affecting overall performance.

Chapter 5

Results

In order to verify the decoder system, various tests were conducted to measure its performance and validate its functionality. The time measurement results presented in this document are based on FPGA run times with the SoC deployed and modified firmware. This setup assesses the decoder's performance in a hardware environment similar to its intended application.

This chapter presents the experimental results of the IOB-MPEG-Decoder system. It also presents the LibMAD software time profiling and performance metrics, which focus on assessing the real-time processing capabilities and efficiency of different processors used in the project. This task also ensures the decoder performs effectively and complies with audio quality and accuracy. The obtained results are compared with the decoding requirements.

5.1 Profiling

The following table displays the execution times for various functions related to the MP2 decoding process, measured in microseconds (μs), relative to the test sample group used for acknowledging different audio file types and features, defined in subsection 4.1.2.

Function	Audio file example			
	Noise	Speech	Jazz	Classical
decode_header	62	64	60	60
mad_header_decode	83	88	79	80
mad_layer_II	18537	18183	27012	25509
II_samples	30	31	31	30
mad_bit_read	3	3	3	3
mad_synth_frame	89279	88930	177243	149710
dct32	7878	7878	7878	7878

Table 5.1: Execution time per function related to the MP2 decoding procedure (frame average) [μs]

The functions in Table 5.1 include data fetching, decoding algorithms, error checking, and output generation routines. Notably, the functions *mad_layer_II* and *mad_synth_frame* consume the most processing time during the MP2 decoding process. The *mad_layer_II* function handles the decoding of Layer II audio data, which involves complex mathematical operations such as subband synthesis and frequency inversion. The significant execution time for this function is due to the computational demand of processing and reconstructing the audio signal from its compressed form.

The *mad_synth_frame* function synthesises the decoded audio frames into PCM samples. This process requires handling a large amount of data and performing calculations to convert frequency-domain

data back into the time domain. The high execution time reflects the extensive computation generating high-quality audio output, similarly with the *dct32* function, a nested call invoked from the frame synthesis method, which is a fraction of the execution time of the complete synthesis procedure. It is computed a 32-point Discrete Cosine Transform on the encoded signal. Although its execution time is less than the previously mentioned functions, it still contributes significantly to the overall decoding time due to the complexity of the transform operations.

Table 5.2 displays the execution times for various functions related to the MP3 decoding process, measured in microseconds (μs):

Function	Audio file example			
	Noise	Speech	Jazz	Classical
III_sideinfo	160	165	297	294
III_scalefactors	140	166	455	372
III_exponents	25	27	50	47
III_requantize	3	3	3	3
III_huffdecode	9342	9631	11413	10158
III_aliasreduce	5214	6514	6517	6517
III_overlap	712	712	980	1076
III_overlap_z	37	63	432	370
III_freqinver	144	151	288	298
fastsdct	50	51	63	63
sdctII	8029	8030	12218	13926
dctIV	11680	11680	17571	20256
imdct36	14547	12176	18527	21317
mad_bit_read	1845	1858	1964	1964
mad_synth_frame	79956	86348	167901	180189
dct32	7878	7878	15720	15725

Table 5.2: Execution time per function related to the MP3 decoding procedure (frame average) [μs]

The group with the prefix "III" constitutes the bitstream decoding aspect; hence, the function's name represents the individual part of the procedure, which is notable for its significant execution times. The function with the longest time interval, *III_huffdecode*, performs Huffman decoding, which is essential for decompressing the high-quality encoded audio data. Following this step, *III_aliasreduce* takes the most CPU time. It consists of aliasing noise reduction, introduced during the encoding process. This function involves calculations that adjust sample values to minimize distortion, adding to the processing load. *III_overlap*, which handles the overlap-add process during the reconstruction of the time-domain signal due to using long transform blocks in MP3 encoding, adjacent blocks of data overlap. It combines the current block with the previous one by adding overlapping samples.

The frequency inversion method, implemented in the *III_freqinver* function, corrects the phase of specific frequency components. In the MP3 decoding process, frequency inversion is applied to the odd-numbered subbands to maintain the correct spectral characteristics of the audio signal, which can lead to audible noise if the phase distortions of subband signals are not corrected.

Discrete time-to-frequency transformation is achieved with the *imdct36* function, which computes the Inverse Modified Discrete Cosine Transform (IMDCT) for 36-point blocks. The IMDCT is used to convert frequency-domain coefficients back into time-domain samples. Functions such as *dctIV* and *sdctII* compute variants of the Discrete Cosine Transform (DCT), which are essential for transforming between time and frequency domains, make part of the inverse modified transform method. The DCT-IV (Type IV DCT) is used within the IMDCT computation, while the Scaled Discrete Cosine Transform II

(*sdctll*) is used in synthesis filter bank operations. These transforms are optimised versions that reduce computational complexity compared to the standard DCT but still require considerable processing power.

The *mad_synth_frame* function, similar to its role in MP2 decoding, consumes a large portion of the decoding time. It is responsible for synthesizing PCM audio samples from the decoded frequency-domain data. This involves passing the data through the synthesis filter bank, which reconstructs the time-domain signal.

In the case of Jazz and Classical recordings, the *dct32* function shows increased execution times. The increased execution time for these genres is attributed to the high bit-rate recordings, channel mode, complex harmonic content and dynamic range, requiring more intricate processing during the transform stages. In summary, most of the execution time is spent on functions related to mathematical transforms, decoding, and synthesis of audio frames due to the complexity of the operations and the amount of processed data.

5.2 Performance metrics

Real-time audio decoding is crucial for applications such as streaming media, where delays between data receipt and audio output must be minimal. The decoder must process each audio frame within the frame's duration to avoid playback interruptions or quality degradation, so the processing time requirements should be addressed. The following metrics have been chosen to evaluate the decoder's performance:

- **Halting Difference:** a cumulative measure of the differences between the duration of each frame and the time taken to decode it, providing a measure of the system's responsiveness. A positive value indicates that the decoder is operating within the required time limits, while a negative value would suggest a potential for real-time processing issues. It is calculated using the following formula (5.1):

$$\text{Halting Difference} = (\text{Frame Duration}(i) - \text{Decoding Time}(i)) \quad (5.1)$$

A positive value indicates that the decoder operates within the required time limits, while a negative value suggests potential real-time processing issues.

- **Elapsed execution time comparison:** This metric compares the elapsed decoding time across the central processing units models to identify which delivers the fastest processing relative to the audio frame duration.

The efficiency of instruction sets and the number of operations dispatched per clock cycle are critical in minimising the decoding time. Many predominant factors, such as the processor's operating frequency, affect how quickly instructions are processed, impacting the total running time overhead. Higher frequencies generally improve performance but can also increase power consumption and heat generation, which might not be ideal for all systems.

Ensuring the decoder meets real-time requirements involves the exploitation of instruction-level parallelism that varies according to the computing architecture and software optimisation. The basis for comparison relies on different processing unit options, as described in section a), according to elapsed decoding time and selecting the option that can handle the required operations efficiently within the time constraints of audio playback. This study evaluates the impact of these factors on the decoding process by analyzing how the different architectures and RISC-V ISA extensions perform under the same workload at a constant frequency of 100 MHz.

5.3 FPGA system resources utilization

This section analyses the FPGA system resources utilized by the MPEG audio decoder implemented in the System-on-Chip (SoC).

- **LUT:** Lookup Tables are configurable logic blocks that implement truth tables for digital logic. They are fundamental in defining logic functions in FPGA fabric. LUTs can be used for simple logic operations or combined to implement complex combinational functions.
- **LUTRAM:** LUTRAM utilizes LUTs as memory rather than logic, allowing storage of temporary data near where it is processed. This local storage can be faster than accessing centralized memory blocks, reducing latency and improving performance for specific data-intensive tasks.
- **Flip-flops:** Flip-flops are used within FPGAs to store binary data (1 or 0). They are essential for creating digital circuit registers, counters, and other state-holding elements. Their usage directly relates to the state and timing management within the system.
- **DSP:** Digital Signal Processors in FPGAs are specialized blocks designed to perform high-speed arithmetic operations, especially multiplication, which are crucial for signal-processing tasks.
- **Block RAM (BRAM):** BRAMs are dedicated memory blocks for storing large data arrays. They provide higher bandwidth and lower latency access than synthesized RAM structures made from LUTs.
- **CLB registers:** registers within Configurable Logic Blocks (CLBs) are used to store intermediate values and support sequential logic operations.
- **Unique control sets:** dedicated sets of control signals, such as clocks, enable and resets required to manage different portions of the FPGA fabric independently.

The following table 5.3 outlines the system component utilization of the MPEG decoder implemented on the FPGA:

Component type	Resources	Utilization (%)	Available
LUT as logic	16568	6.83	242400
LUTRAMs	7711	6.84	112800
DSP	7	0.36	1920
RAMB36	90	15.00	600
RAMB18	1	0.08	1200
CLB registers	23042	4.75	484800
Unique control sets	1805	2.98	60600

Table 5.3: FPGA resource usage for the IOB-MPEG-Decoder

5.4 Evaluation on different types of audio content

This section evaluates the MPEG audio decoder's performance over different types of audio content. The system operates at a frequency of 100MHz and uses the PicoRV32 as the baseline for code profiling. Key aspects of the evaluation include:

- **File comparison:** verifying the decoder's accuracy by comparing the expected versus actual decoded data ensures the implementation produces correct audio output. This is accomplished by

validating the file contents, byte-by-byte, against the generated output made with the original implementation of LibMAD. All tested files yielded compliant results compared with the pure software implementation.

- **Influence of audio features:** analysing decoder performance across different file specifications, such as bitrate, frequency, sample rate and channel format, evaluates the algorithm’s versatility to handle various audio content effectively.

A detailed comparison of different RISC-V processing units according to the chosen metrics provides insights into their suitability for audio decoding. To assess the performance improvements offered by various processor architectures, we compared the execution times of the PicoRV, DarkRV, and VexRV across different types of audio content. The results are summarized in table 5.4.

CPU	Audio file example							
	Noise		Speech		Jazz		Classical	
PICORV	115872	139762	115177	145453	212306	261403	183270	272575
VEXRV	5905	8074	6056	7083	7047	8241	6879	8027
DARKRV	3845	5727	4043	5315	5778	6465	4677	6341

Table 5.4: Execution time on average per frame for MP2 and MP3 files [μs]

PicoRV32 has significantly higher execution times than the VexRV and DarkRV, which is much more significant than the duration of one audio frame, making it impractical for reliable real-time use. In contrast, the DarkRV and VexRV architectures achieve decoding times within the frame duration.

5.5 Real-time requirements

To quantify the system’s responsiveness, the total halting difference is calculated for each CPU and audio file, as shown in Table 5.5

CPU	Audio file example							
	Noise		Speech		Jazz		Classical	
PICORV	-89750	-113640	-89055	-119331	-186120	-235281	-157148	-246453
VEXRV	+20217	+18048	+20066	+19039	+19075	+17881	+19243	+18095
DARKRV	+22277	+20395	+22079	+20807	+20344	+19657	+21445	+19781

Table 5.5: Halting difference for decoding time per frame of MP2 and MP3 files [μs]

A positive halting difference value indicates that the each frame is decoded faster than the frame duration, leaving sufficient time for seamless audio playback. The DarkRV and VexRV CPUs consistently show positive values, demonstrating their ability to meet processing requirements. In contrast, the PicoRV32 shows negative values, confirming that it cannot process frames quickly enough for real-time decoding, in MP2 and MP3 file formats.

Notably, the smallest discrimination thresholds for interaural time differences in human hearing are near $10 \mu s$. Therefore, the positive halting differences achieved by the DarkRV and VexRV (ranging from approximately $+17,881 \mu s$ to $+22,277 \mu s$) are well above this threshold, ensuring that the decoded audio will not introduce perceivable delays or synchronization issues.

$$Speedup\ required = \frac{Worst - case\ execution\ time}{Audio\ frame\ playback\ duration} \quad (5.2)$$

The required and attained speedups for the audio decoding process values are summarized in the table 5.6:

	Audio file example							
Speedup	Noise		Speech		Jazz		Classical	
Required	4,44	5,35	1,60	4,04	8,13	10,01	7,02	10,43
Attained	30,14	24,40	29,96	27,37	36,74	40,43	39,19	42,99

Table 5.6: Attained speedup and required values for MP2 and MP3 files

The required speedup represents the minimum factor by which the decoding process must be accelerated to achieve real-time performance compared to the baseline PicoRV32. The attained speedup shows the improvement achieved with the DarkRV32 and VexRV32. The results indicate that the attained speedups significantly exceed the required values. To further analyze the performance, it is considered the Cycles Per Instruction (CPI) aspect:

$$CPI = \frac{Execution\ Time * Clock\ rate}{Instruction\ Count} \quad (5.3)$$

An improvement in CPI indicates that fewer clock cycles are needed per instruction, leading to faster execution times. Differences in architecture and instruction set extensions, such as including multiply-accumulate operations or exploitation instruction parallelism using pipelined architectures, substantially impact the decoding time. While increasing the clock frequency can reduce execution time, it may come at the cost of higher power consumption, thermal output, and timing issues regarding signal synchronization in digital circuitry.

The differences between the required and attained speedups for MP2 and MP3 files highlight the efficiency of the different architectures. DarkRV and VexRV units consistently provide sufficient headroom over the required values, especially in the worst-case scenario for real-time decoding. This margin in processing bandwidth means these architectures meet and comfortably exceed the minimum performance thresholds for different audio formats, ensuring stability and reliability.

In conclusion, the verification demonstrates that the DarkRV and VexRV architectures meet the real-time processing requirements for MPEG audio decoding across various types of audio content. They offer the best performance options, making them suitable for implementing the IOb-MPEG-Decoder system.

Chapter 6

Conclusions

This chapter summarises the main achievements of the thesis, which focused on implementing an MPEG audio decoder on a System-on-Chip (SoC) with a RISC-V ISA central processing unit. The project explored various facets of architecture composition, from software development and verification to the physical deployment, emphasising flexibility and open-source code integration.

6.1 Achievements

The decoder's implementation demonstrated high flexibility in handling different audio formats with compliance by exploiting processing parallelism and significantly improving decoding efficiency. The approach to achieving the desired performance requisites was taken to the level of the central processing unit. With this approach, a working audio decoder was successfully implemented and tested.

Digital audio data decompression is successfully achieved with the PicoRV unit and efficient real-time decoding with the DarkRV and VexRV models. It also met stringent sound quality compared to pure software implementation, with reduced component usage in the FPGA fabric. The project highlighted the challenges and solutions in moving from a simulated environment to actual physical implementation. Bridging the gap between software development, testing, and deployment on physical targets is fundamental in embedded systems and hardware design. Several key deliverables were produced:

- **Audio decoding application porting and profiling:** the open-source *LibMAD* library was ported to the MPEG decoder system, and comprehensive execution profiling was conducted to analyze CPU cycle count during the decoding procedure to validate the real-time processing requirement.
- **Input/ output modules:** integrated I/O modules to transport encoded and decoded audio data, fundamental for incorporating the decoder into existing infrastructures, providing seamless data flow for real-time audio decoding.
- **Streaming capabilities:** this feature allows the decoder to process live audio streams, an essential feature for modern multimedia applications.
- **Tester platform:** connection with a tester platform, allowing comprehensive testing and validation under simulated real-world conditions. This setup was essential for verifying the performance and compliance of the decoder.
- **Improve performance without compromising accuracy:** achieved enhancements at an operational frequency of 100 MHz without compromising audio decoding accuracy, compared with the pure software implementation of the decoder.

6.2 Future work

While the project met its primary objectives, the following areas are suggested for future development:

- **Support for additional audio formats:** expanding the decoder to support additional audio formats would increase its applicability, for example, AAC or FLAC.
- **Code optimisation:** further optimisations in software code could be explored to enhance the decoder's suitability for low-power operated devices.
- **Scalability testing:** extensive testing could ensure the design performs well as functional demands and complexity increase.
- **OS environment:** integration with an operating system to exploit asynchronous digital audio decoding.
- **Multi-channel support:** burst-mode option and multiple channels support.
- **Fully integrated application:** non-isolated implementation on a fully working environment with an encoder source and decoder endpoint on a DAB stream.
- **Implement with video:** expansion to include a video decoding feature in parallel with the developed work in audio data decompression.

This study contributes to the ongoing efforts to develop more efficient, scalable, and adaptable solutions for audio decoding in MPEG-1/2 formats, which can conceive and enhance audiovisual applications across broadcasting and multimedia, extending to future digital communication frameworks.

References

- [1] International Organization for Standardization. ISO 11172:1993 - Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s, 1993. URL <https://www.iso.org/standard/22412.html>. Accessed May 2024.
- [2] International Organization for Standardization. ISO 13818:1998 - Information technology — Generic coding of moving pictures and associated audio information, 1998. URL <https://www.iso.org/standard/26797.html>. Accessed May 2024.
- [3] P. L. Yingbiao Yao, Qingdong Yao and Z. Xiao. Embedded software optimization for MP3 decoder implemented on RISC core. *IEEE Transactions on Consumer Electronics*, 50, 2004.
- [4] IObundle, Lda, 2018. URL <https://www.iobundle.com>. Accessed May 2024.
- [5] I. Underbit Technologies. LibMAD - MPEG audio decoder library, 2004. URL <https://github.com/markjee/libmad>. Accessed December 2023.
- [6] R. G. David Katz. *Embedded Media Processing*, chapter 5, page 65. Elsevier Science & Technology, 2005.
- [7] S. W. Smith. *Guide to digital signal processing*, chapter 8, 9, pages 141–177. California Technical Publishing, 1999.
- [8] S. D. Jürgen Herre. Psychoacoustic Models for Perceptual Audio Coding. *International Audio Laboratories Erlangen*, July 2019.
- [9] D. Pan. A tutorial on MPEG audio compression. *IEEE MultiMedia*, 2(2):60–74, 1995.
- [10] N. Humfrey. Twolame - optimised mpeg audio layer 2 encoder, 2005. URL <https://www.twolame.org>. Accessed December 2023.
- [11] T. A. da Silva. MPEG1/ 2 layers I/ II encoder using a RISC-V processor and hardware accelerators. MsC thesis, Instituto Superior Técnico, Portugal, 2023.
- [12] A. Sugiyama and M. Iwadare. The origin of digital information devices: the silicon audio and its family. *APSIPA Transactions on Signal and Information Processing*, 7, 2018.
- [13] K. Lagerström. Design and implementation of an MPEG-1 Layer III audio decoder. MsC thesis, Chalmers University of Technology, Sweden, 2001.
- [14] W. Su and P. Razaghi. Design of flexible audio processing platforms using the system-on-chip environment. Technical report, Department of Electrical & Computer Engineering, The University of Texas at Austin, 2012.
- [15] S. M. Resve Saleh, Steve Wilton. System-on-Chip: Reuse and Integration. *Proceedings of the IEEE*, 94(6):1050–1069, June 2006.

- [16] IObundle. IOb-SoC, 2020. URL <https://github.com/IObundle/iob-soc>. Accessed May 2024.
- [17] F. H. Dirk Koch and D. Ziener. *FPGA for Software Programmers*, chapter 15, pages 261–284. Springer International, 2016.
- [18] S. Williams. Icarus Verilog, 2020. URL <https://github.com/steveicarus/iverilog>. Accessed October 2023.
- [19] W. Snyder. Verilator, 2001. URL <https://www.veripool.org/verilator/>. Accessed October 2023.
- [20] IObundle. IOb-LIB, 2023. URL <https://github.com/IObundle/iob-lib>. Accessed May 2024.
- [21] Verilog-AXI, 2018. URL <https://github.com/alexforencich/verilog-axi>. Accessed December 2023.
- [22] IObundle. IOb-PicoRV32, 2022. URL <https://github.com/IObundle/iob-picorv32>. Accessed October 2023.
- [23] IObundle. IOb-Cache, 2020. URL <https://github.com/IObundle/iob-cache>. Accessed December 2023.
- [24] IObundle. IOb-UART, 2020. URL <https://github.com/IObundle/iob-uart>. Accessed May 2024.
- [25] R. McGrath. GNU Make, 2023. URL <https://www.gnu.org/software/make>. Accessed October 2023.
- [26] L. Torvalds. Linux Kernel, 1991. URL <https://www.linux.org>. Accessed October 2023.
- [27] N. Foundation. nix-shell - virtual build environment, 2020. URL <https://nixos.org/manual/nix/stable/command-ref/nix-shell>. Accessed November 2023.
- [28] K. A. Andrew Waterman. The RISC-V instruction set manual - volume I: User level ISA. Technical report, CS Division, EECS Department, University of California, Berkeley, 2017.
- [29] D. P. Andrew Waterman, Yunsup Lee and K. Asanović. RISC-V Compressed ISA V1.7. Technical report, CS Division, EECS Department, University of California, Berkeley, 2015.
- [30] R. Stallman. GNU Operating System, 1983. URL <https://www.gnu.org/home.en.html>. Accessed October 2023.
- [31] IObundle. IOb-SoC-SUT, 2022. URL <https://github.com/IObundle/iob-soc-sut>. Accessed May 2024.
- [32] R. Patel and A. Rajawat. Embedded software profiling methodologies. *International Journal of Embedded Systems and Applications*, 1(2), 2011.
- [33] J. Fenlason. GNU Profiler, 1988. URL https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html. Accessed October 2023.
- [34] D. S. Andreas Gerstlauer, Daniel Gajski. Design of a MP3 decoder using the System-on-chip Environment. Technical report, Center for Embedded Computer Systems, University of California, Irvine, 2007.
- [35] C. Wolf. A Size-Optimized RISC-V CPU - PicoRISCV, 2015. URL <https://github.com/YosysHQ/picorv32>. Accessed October 2023.

- [36] SpinalHDL. FPGA implementation of 32-bit RISCv CPU - VexRISCv, 2024. URL <https://github.com/SpinalHDL/VexRiscv>. Accessed October 2023.
- [37] IObundle. IOb-VexRV32, 2024. URL <https://github.com/IObundle/iob-vexriscv>. Accessed October 2023.
- [38] M. S. et. al. Opensource CPU - DarkRISCv, 2020. URL <https://github.com/darklife/darkriscv>. Accessed October 2023.
- [39] IObundle. IOb-DarkRV32, 2022. URL <https://github.com/IObundle/iob-darkrv32>. Accessed October 2023.
- [40] IObundle. IOb-AXI4-Stream, 2022. URL <https://github.com/IObundle/iob-axis>. Accessed December 2023.
- [41] IObundle. IOb-Timer, 2020. URL <https://github.com/IObundle/iob-timer>. Accessed December 2023.
- [42] C. Villarraga. Formal verification of firmware-based system-on-chip modules. MsC thesis, Technische Universität Kaiserslautern, Deutschland, 2007.
- [43] A. A. B. Nóbrega. IOb-SoC-based Tester System. MsC thesis, Instituto Superior Técnico, Portugal, 2022.
- [44] F. Segatz. Continuous integration for embedded software with modular firmware architecture. MsC thesis, KTH Royal Institute of Technology, Sweeden, 2023.
- [45] D. Mazzoni and R. Dannenberg. Audacity, 1999. URL <https://www.audacityteam.org/>. Accessed October 2023.

