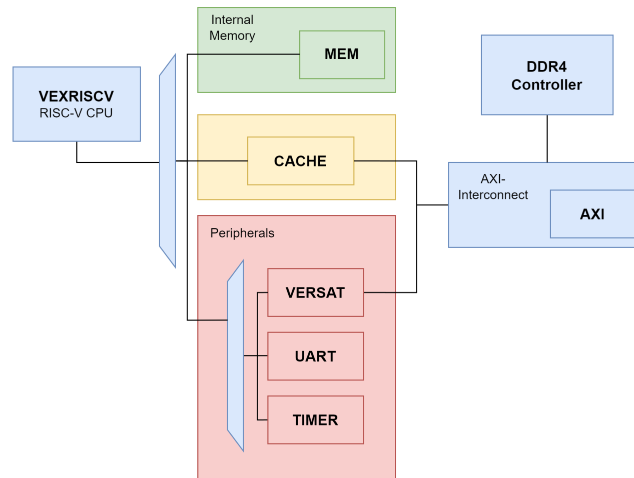

MPEG1/2 layers I/II encoder using a RISC-V processor and hardware accelerators

Tiago Alves da Silva

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. Doutora Teresa Maria Canavarro Menéres Mendes de Almeida
Supervisor: Prof. Doutor José João Henriques Teixeira de Sousa
Member of the Committee: Prof. Doutor Marcelino Bicho dos Santos

November 2023

Declaration

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

Acknowledgements

I want to thank Professor José de Sousa for for his invaluable guidance and support during my master's thesis. As my supervisor, he played a significant role in helping me understand the research process, providing valuable insights on what needed to be done and how to approach it. His professionalism and high standards were instrumental in this work.

I want to express my gratitude to Rúben Teixeira for his assistance in enhancing my understanding of the hardware accelerator matter. His guidance on the practical aspects of this work, including optimization techniques, was immensely helpful, especially during the latter stages.

I am also deeply thankful to my family for their unwavering support and encouragement. Their constant belief in me and encouragement to give my best has been a motivating force throughout my academic journey.

Resumo

Este trabalho apresenta um codificador de áudio MPEG-1/2 Layer II para uma arquitetura embarcada RISC-V, que inclui um acelerador de hardware reconfigurável. Embora tais sistemas sejam comuns em processadores embarcados comerciais como ARM, este trabalho é o primeiro a apresentar uma implementação em RISC-V. A vantagem é que a arquitetura RISC-V é uma especificação aberta com alguns designs de hardware de código aberto disponíveis. Um acelerador de hardware permite que o sistema funcione em ambientes de baixa frequência, como um dispositivo FPGA. Neste trabalho, o software do sistema utiliza a biblioteca de código aberto do codificador TwoLAME. O hardware do sistema é baseado no IOb-SoC, uma plataforma RISC-V SoC de código aberto escrita em Verilog. O processador VexRiscv foi escolhido, e o acelerador de hardware foi implementado usando a ferramenta de design de acelerador reconfigurável de código aberto Versat. O trabalho apresenta otimizações de software e dois aceleradores de hardware para acelerar o cálculo do modelo psicoacústico do algoritmo. O desempenho base é 6,2 vezes mais lento que o tempo real para um sistema rodando a 100MHz, o que indica que uma implementação para 620MHz atenderia ao objetivo. Com aceleração de hardware, o desempenho alcançado é 2,4 vezes mais lento que o tempo real para um sistema rodando a 100MHz, o que indica que uma implementação para 240MHz atenderia ao objetivo.

Palavras-chave: Codificador de Áudio, *TwoLAME*, Sistema num Chip, Aceleração de Hardware, Matriz de Portas Programável em Campo.

Abstract

This work introduces an MPEG-1/2 Layer II Audio encoder for a RISC-V embedded architecture featuring a reconfigurable hardware accelerator. Although such systems are standard on commercial embedded processors such as ARM, this work is the first to present a RISC-V implementation. The advantage is that the RISC-V architecture is an open specification with a few open-source hardware designs available. A hardware accelerator allows the system to run on low-frequency environments like an FPGA device. In this work, the system software uses the TwoLAME encoder open-source library. The system hardware is based on IOb-SoC, an open-source RISC-V SoC platform written in Verilog. The VexRiscv CPU has been chosen, and the hardware accelerator has been implemented using the Versat open-source reconfigurable accelerator design tool. The work features software optimizations and two hardware accelerators to accelerate the computation of the psychoacoustic model of the algorithm. The base performance is 6.2x slower than real-time for a system running at 100MHz, which indicates that an implementation for 620MHz would meet the goal. With hardware acceleration, the achieved performance is 2.4x slower than real-time for a system running at 100MHz, which indicates that an implementation for 240MHz would meet the goal.

Keywords: Audio encoder, *TwoLAME*, System-on-Chip, Hardware acceleration, Field-Programmable Gate Array.

Contents

1 Introduction **1**

1.1 About MPEG-1 Layer II 1

1.2 Motivation 1

1.3 Objectives 3

1.4 Outline 4

2 Background **5**

2.1 ISO/IEC 11172 standard 5

2.2 Moving Picture Experts Group 8

2.3 MPEG-1/2 Layers I/II IP Cores 10

2.3.1 CWda74 11

2.3.2 IPB-MPEG-SE 11

2.4 MPEG-1/2 Layers I/II Chips 12

2.4.1 CX23415 Codec 12

2.4.2 Futura II ASI+IP™ 13

2.5 MPEG-1/2 Layers I/II Systems 13

2.6 MPEG-1/2 Layers I/II Software 14

2.6.1 TooLAME/LAME 14

2.6.2 *TwoLAME* 14

2.7 System-on-Chip 17

2.7.1 IOb-SoC components 18

2.7.2 IOb-SoC deliverables and resources 19
2.7.3 IOb-SoC repository 19
2.8 *Versat* 20
2.8.1 Functional units 21
2.8.2 Operators 22
2.8.3 Syntax 23

3 Hardware architecture 27

3.1 VexRiscv 27
3.2 Versat 27
3.3 Timer 29
3.4 *spectrum_search* accelerator 30
3.4.1 Control and Data paths 30
3.4.2 Functional units 32
3.4.3 Modules 33
3.4.4 Control 36
3.5 *masking_threshold* accelerator 36
3.5.1 Control and Data paths 36
3.5.2 Modules 37
3.5.3 Control 38
3.6 Overview 38

4 Software architecture 41

4.1 Audio test files 41

4.2 Firmware 42

4.3 Optimization 44

4.4 Profiling 47

5 Results **49**

5.1 Profiling 49

5.2 FPGA implementation 51

5.3 Execution time 53

5.4 Real-time requirements 54

6 Conclusion **59**

6.1 Achievements 59

6.2 Future work 60

Bibliography **61**

List of Tables

1 *CWda74* features. 11

2 *TwoLAME* features. 14

3 Implementation resources for *Xilinx Kintex Ultrascale* Devices. 19

4 Implementation resources for Intel Cyclone V Devices. 19

5 *Versat* functional units. 22

6 *Versat* operators. 23

7 New *Versat* functional units. 32

8 Execution time for all input files (first phase of *profiling*) [ms]. 49

9 Execution time for all input files (second phase of *profiling*) [ms]. 50

10 Execution time for all input files (third phase of *profiling*) [ms]. 51

11 FPGA implementation results of IOb-MP2-E with and without *Versat*. 52

12 Execution time of IOb-MP2-E with and without *Versat* for all input files [ms]. 53

13 Real time for all input files encoding [ms]. 55

14 Speedup achieved, possible, and required for all input files. 56

List of Figures

1 Audio encoder basic structure from ISO/IEC 11172 [11]. 6
2 Encoder block diagram from ISO/IEC 11172 [11]. 7
3 Layer II encoder flow chart from ISO/IEC 11172 [11]. 8
4 First part of *firmware.c* pseudo code. 15
5 Second part of *firmware.c* pseudo code. 16
6 Third part of *firmware.c* pseudo code. 17
7 Base IOb-SoC high-level block diagram from IObundle [35]. 18
8 *versatspec.txt* example. 24
9 High-level block diagram of the IOb-MP2-E in use. 29
10 Control and data paths of the *spectrum_search* hardware accelerator. 31
11 Control and data paths of the *masking_threshold* hardware accelerator. 37
12 Hardware setup overview. 39
13 New *firmware.c* pseudo code. 43

List of Acronyms

AAC Advanced Audio Coding

AC-3 Audio Codec 3

AES-EBU Audio Engineering Society-European Broadcasting Union

AES3 Audio Engineering Society 3

AMBA Advanced Microcontroller Bus Architecture

APB Advanced Peripheral Bus

ASIC Application-Specific Integrated Circuit

AXI Advanced eXtensible Interface

BRAM Block RAM

CBR Constant Bit Rate

CPU Central Processing Unit

DAB Digital Audio Broadcast

DSP Digital Signal Processor

DMA Direct Memory Access

FPGA Field-Programmable Gate Array

FPU Floating-Point Unit

FU Functional Unit

H.264 Advanced Video Coding

HE-AAC High-Efficiency Advanced Audio Coding

HDMI High-Definition Multimedia Interface

Hz Hertz

I2S Inter-IC Sound

IEC International Electrotechnical Commission

IP Intellectual Property

ISO International Organization for Standardization

LUTs Look-Up Tables

Mbps Megabits per second

MS Milliseconds

MSB Most Significant Bit

MPEG Moving Picture Experts Group

MP2 MPEG-2 Layer II

RTL Register-Transfer Level

RV32IM RISC-V 32-bit Integer Multiplication

SDI Serial Digital Interface

SPI Serial Peripheral Interface

SPDIF Sony/Philips Digital Interface

TDM Time Division Multiplexed

URAM UltraRAM

UART Universal Asynchronous Receiver-Transmitter

VHDL VHSIC Hardware Description Language

VBR Variable Bitrate

1 Introduction

1.1 About MPEG-1 Layer II

Moving Picture Experts Group (MPEG)-1 Layer II, often referred to simply as MP2, is an audio compression format developed as part of the MPEG-1 standard. MPEG-1 is a set of standards created by the MPEG for video and audio compression. MPEG-1 Layer II specifically addresses audio compression.

MP2 is less well-known and widely used than its successor, MP3 (MPEG-1 Layer III), which offers higher compression and better audio quality for consumer applications. However, MP2 has been used in various professional applications, including:

- **Digital Broadcasting:** MP2 was widely used in Digital Audio Broadcasting (DAB) systems, particularly in Europe. It provided a good balance between audio quality and compression efficiency for radio and television broadcasting.
- **Digital Audio Recording:** Some early digital audio recorders and workstations used MP2 for audio compression. It was considered a high-quality audio format for professional applications.
- **Video DVDs:** MP2 audio compression was used in some early video DVD standards, particularly in European and Japanese releases.
- **Video Conferencing:** MP2 was used in video conferencing and telecommunication applications for its relatively good audio quality and low latency.
- **Archival Audio:** MP2 was used for archiving audio in some cases, as it provided a good compromise between file size and audio quality.

1.2 Motivation

There are three options for integrating an MPEG-1/2 Layer II encoder in a system: buy an encoder chip, use a software encoder in the user system, or buy an IP core.

Encoder chips, like the *CX23415 MPEG-2 Codec* [1], the *MPEG-2 Encoder CW-4888* [2], or the *Futura II ASI+IP* [3] can be purchased off the shelf. However, this means an additional chip on the board, increasing its area/volume, weight, and power consumption.

A software encoder, which relies on a Central Processing Unit (CPU), may be used instead. This CPU can be either an existing one available in the system or an additional processor chip or Intellectual

Property (IP) core [4]. Such software encoders are commonplace in commercial embedded processors, with ARM processors being a notable example. However, it is essential to consider the cost and effort associated with porting the software to the CPU. Depending on the specific application, this factor may significantly influence both the overall viability and efficiency of the system.

The last option is to license a commercial MPEG-1/2 Layer II encoder IP Core. It reduces area/volume, weight, and power consumption, allowing users to develop a top-notch system and beat the competition. To the best of the author's knowledge, there is only one IP core designed explicitly for MPEG-1/2 Layer I/II Audio in the market, namely the *CWda74* [5], later re-branded *IPB-MPEG-SE* [6], which uses fixed-point calculations.

This work considers the design of a RISC-V IP core. Designing an embedded MP2 encoder using a RISC-V processor can have several motivations and advantages, depending on the specific use case and requirements. Here are some reasons for such a design:

- **Open-Source and Customizable Architecture:** RISC-V is an open-source instruction set architecture (ISA). This openness allows developers to customize and optimize the processor for their specific application, making it a flexible choice for embedded systems. You can tailor the processor to handle the tasks required for MP2 encoding efficiently.
- **Low Power Consumption:** RISC-V processors can be designed to be power-efficient. It is crucial for embedded systems, especially in applications where power constraints are significant, such as portable devices and battery-powered equipment.
- **Real-Time Processing:** MP2 encoding often requires real-time processing, which RISC-V processors can be optimized for. Real-time capabilities are essential in applications like live audio streaming, broadcasting, or communication systems.
- **Integration and System-on-Chip (SoC):** RISC-V processors can be integrated into a larger system-on-chip (SoC), including various components, such as memory, peripherals, and custom accelerators. This integration can result in a compact and efficient solution for MP2 encoding, making it well-suited for embedded applications.
- **Custom Instructions and Accelerators:** RISC-V allows for creating custom instructions and accelerators tailored to specific tasks. It can significantly accelerate the MP2 encoding process by implementing instructions that directly assist in the encoding algorithm.
- **Licensing and Cost:** Using RISC-V is often cost-effective because it is open-source, which means no licensing fees are required. It can be particularly advantageous for embedded system manufacturers looking to minimize costs.

- **Scalability:** RISC-V processors come in various configurations and can be scaled to match the performance requirements of the target application. This scalability is beneficial for accommodating different processing needs in various embedded devices.
- **Longevity and Future-Proofing:** RISC-V is designed to be a stable and long-lasting architecture. By using RISC-V, you can ensure that your embedded MP2 encoder will remain compatible and supported for years.
- **Research and Education:** Using RISC-V for an MP2 encoder project can also serve educational and research purposes. It allows researchers and students to experiment with processor design, optimization techniques, and multimedia encoding algorithms.

Overall, designing an embedded MP2 encoder with a RISC-V processor offers flexibility, optimization, and efficiency for specific applications. The open RISC-V architecture, combined with hardware accelerators, improves energy efficiency, encoding speed, and resource utilization.

1.3 Objectives

This work introduces an MPEG-1/2 Layer II Audio encoder for a RISC-V embedded architecture featuring a reconfigurable hardware accelerator. The encoder will be developed using *IObundle, Lda*'s [7] IOb-SoC [8], a System-on-Chip template comprising an open-source RISC-V [9] processor. The TwoLAME [10] repository, an open-source MPEG Audio Layer II encoding software based on the ISO/IEC 11172 [11], will provide the algorithm. The objectives of this work are the following:

- Port the *TwoLAME* [12] audio encoder open-source library to the system, using floating-point precision.
- Perform software optimizations.
- Profile the encoding algorithm while running in FPGA [13].
- Use *Versat* [14] open-source reconfigurable accelerator design tool to accelerate the encoding algorithm.
- Check the requirements for real-time encoding.
- Compare with the fixed-point *CWda74* [5], the only competitor.

1.4 Outline

This document is composed of 5 more chapters. In the second chapter, the state-of-the-art and the tools used in this work are presented. In the third chapter, the hardware architecture is fully described. In the fourth chapter, the software architecture is fully described. In the fifth chapter, experimental results are presented. In the sixth and final chapter, the achievements are pointed out, and directions for future work are outlined.

2 Background

This chapter addresses the state-of-the-art. It starts by presenting the ISO/IEC 11172-3 [11] standard. Then, it describes the MPEG-1/2 Layers I/II encoders available in the market. In the end, it presents the tools used in this work, IOb-Soc and *Versat*.

2.1 ISO/IEC 11172 standard

The ISO/IEC 11172 is an international standard under the title *Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s*. This standard is divided into four parts (Systems, Video, Audio, and Compliance testing), with the Audio part being the relevant one for this work.

Focusing on the audio, the ISO/IEC 11172-3 specifies the coded representation of high-quality audio for storage media, and also the method for decoding high-quality audio signals. Therefore, this part is intended for application to digital storage media. It provides a total continuous transfer rate of around 1.5Mbits/sec for both audio and video bitstreams, with sampling rates of 32kHz, 44.1kHz, and 48kHz.

An audio encoder is responsible for processing the digital audio signal and producing the compressed bitstream for storage. The encoder algorithm is not standardized and may use various means of encoding, such as estimation of the auditory masking threshold, quantization, and scaling. However, the encoder output must be such that a decoder, conforming to the specifications of the coded audio bitstream, will produce audio suitable for the intended application.

Figure 1 illustrates the basic structure of an audio encoder.

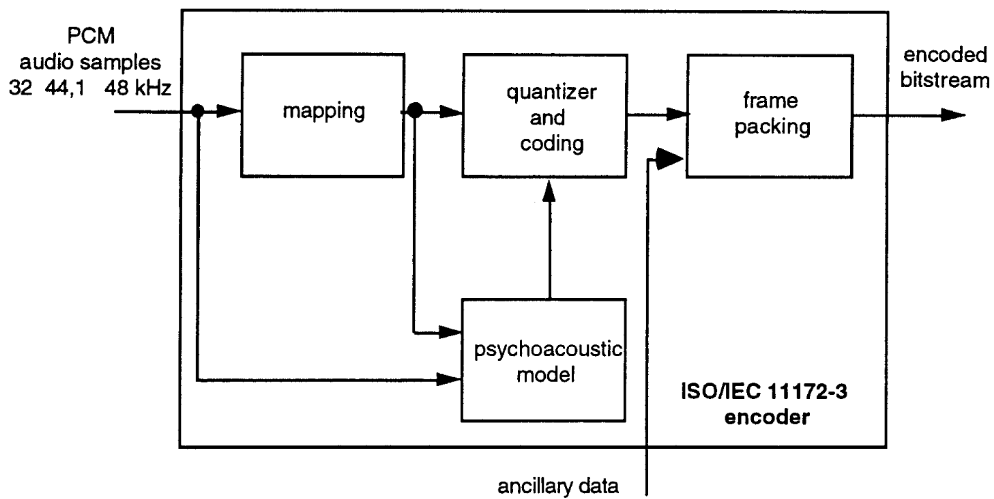

Figure 1: Audio encoder basic structure from ISO/IEC 11172 [11].

The encoding process contains four main blocks: *mapping*, *psychoacoustic model*, *quantizer and coding*, and *frame packing*.

First, the **mapping** block creates a filtered and subsampled representation of the input audio stream, usually called subband samples (for Layer II). At the same stage, the **psychoacoustic model** block creates a set of data to control the next block (*quantizer and coding*). These control data depend on the coder implementation, with one possibility being the use of a masking threshold estimation. Then, the **quantizer and coding** block creates a set of coding symbols from the mapped input samples, depending on the encoding system once again. Finally, the **frame packing** block assembles the actual bitstream from the output data of the previous blocks, adding information if necessary. The encoding process supports four different modes: single channel, dual channel (two independent audio signals coded within one bitstream), stereo (left and right signals of a stereo pair coded within one bitstream), and Joint Stereo (with the stereo irrelevancy and redundancy exploited).

In particular, the ISO/IEC 11172-3 (MPEG-Audio) psychoacoustic algorithm contains four primary parts, as shown in figure 2: *Filter Bank*, *Psychoacoustic Model*, *Bit or Noise Allocation* and *Bitstream Formatting* [11].

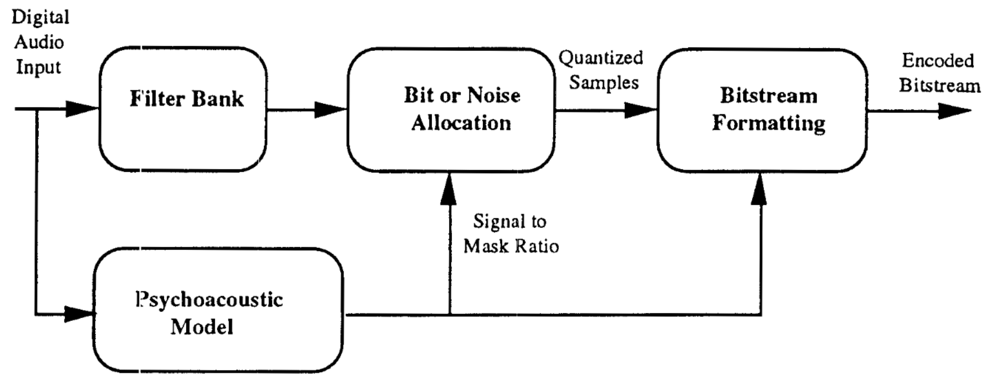

Figure 2: Encoder block diagram from ISO/IEC 11172 [11].

The **Filter Bank** does a time-to-frequency mapping, being one of two types. It can be a polyphase filter bank or a hybrid polyphase/Modified discrete cosine transform (MDCT) [15] filter bank, with each delivering a specific mapping in time and frequency. These filterbanks are critically sampled, having the same number of samples in both analyzed and time domains, and provide the primary frequency separation for the encoder, with quantized output samples. In layer II, a filter bank with 32 subbands is used. In each subband, 12 or 36 samples are grouped for processing. The **Psychoacoustic Model** calculates a just noticeable noise level for each band in the filter bank. This noise level is used in the *Bit or Noise Allocation* part to determine the actual quantizers and quantizer levels. The final output of the model is a signal-to-mask ratio (SMR) for each band (Layer II). The **Bit or Noise Allocation** takes both the output samples from the *Filter Bank* and the SMR from the *Psychoacoustic Model* and adjusts the bit allocation, to meet the bitrate and masking requirements. At low bitrates, these methods attempt to spend bits in a way that is inoffensive when they cannot meet the psychoacoustic demand at the required bitrate. In Layer II, this method is a bit allocation process, where several bits are assigned to each sample (or group of samples) in each subband. The **Bitstream Formatting** takes the quantized filterbank outputs, together with the bit allocation and other required side information, and encodes and efficiently formats all that information. In Layer II, a fixed Pulse code modulation (PCM) code is used for each subband sample, with the exception that quantized samples may be grouped.

Figure 3 shows a more detailed Layer II Encoder flow chart.

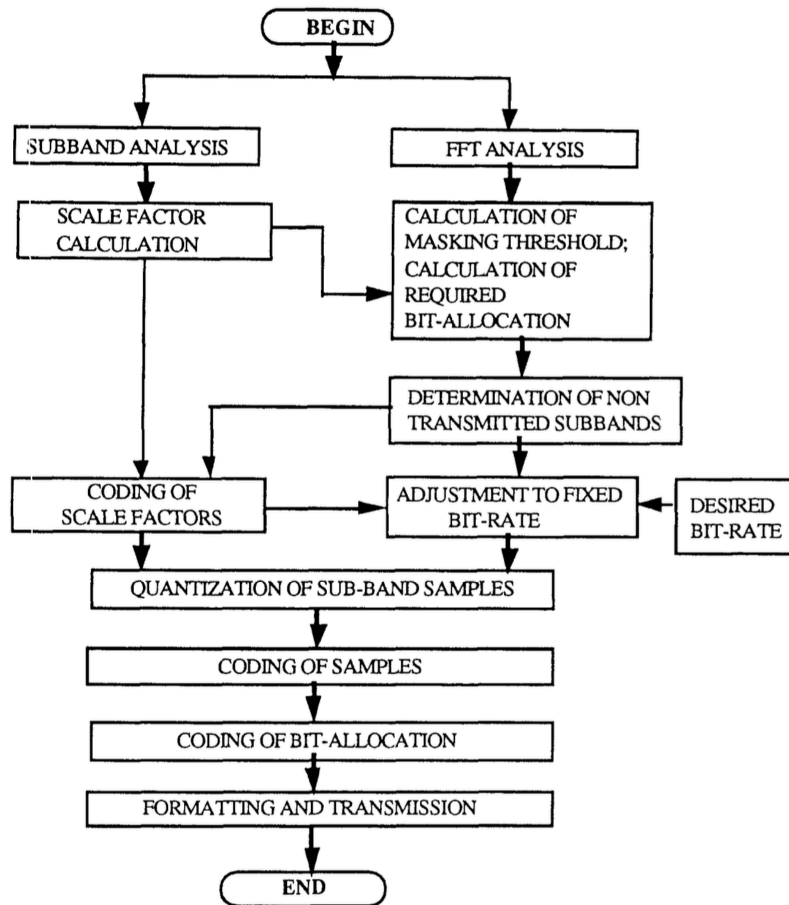

Figure 3: Layer II encoder flow chart from ISO/IEC 11172 [11].

2.2 Moving Picture Experts Group

The MPEG is a working group that sets standards for media coding, established by the International Organization for Standardization (ISO) [16] and the International Electrotechnical Commission (IEC) [17]. Thus, the MPEG standard is a set of specifications for audio and video compression, containing two variations (among others), MPEG-1 and MPEG-2. Both variations cover audio and video, but since this work is focused on audio, the relevant standards are MPEG-1 Audio and MPEG-2 Audio.

The MPEG-1 Audio, defined by ISO/IEC 11172-3 [11], standardizes the information that an audio encoder must produce to write a bitstream conformant with the standard requirements. Moreover, it standardizes how an audio decoder has to parse, decompress, and resynthesize the information to reconstruct the original audio stream.

This standard performs perceptual audio coding, which does not attempt to retain the input signal exactly after encoding and decoding. Instead, it ensures that the output signal sounds the same to a hu-

man listener. More precisely, an MPEG-1 audio encoder transforms the sound signal into the frequency domain, eliminates the frequency components that are masked by stronger frequency components, and packages the analyzed signal into a compressed audio bitstream.

Focusing on the encoding process, the primary psychoacoustic effect is called ”auditory masking”, where parts of a signal are not audible due to the function of the human auditory system. For example, if there is a sound that consists mainly of one frequency, all other sounds that consist of a close frequency but are much quieter will not be heard. Considering this, the parts of the signal that are masked are commonly called irrelevant, as opposed to the redundant parts that are removed. To eliminate this irrelevancy, the encoder contains a **psychoacoustic model** which analyzes the input signals within consecutive time blocks and determines, for each block, the spectral components of the input audio signal (by applying a **frequency transform**). Then, it models the masking properties of the human auditory system, and estimates the noise level for each frequency band, usually called ”threshold of masking”. In parallel, the input signal is fed through a time-to-frequency mapping, resulting in spectrum components for subsequent coding. Finally, in the **quantization and coding** stage, the encoder tries to allocate the available number of data bits, meeting both the bitrate and masking requirements (”threshold of masking”). The information on how the bits are distributed over the spectrum is contained in the bitstream as side information.

The MPEG-1 standardizes three different coding schemes, namely Layer I, Layer II, and Layer III, with the first two layers being the relevant ones in this work. Layer I has the lowest complexity and is specifically suitable for applications where the encoder complexity plays an important role. Layer II requires a more complex encoder and decoder, being directed towards one-to-many applications, i.e. one encoder serves many decoders.

Compared to Layer I, Layer II can remove more of the signal redundancy and apply the psychoacoustic threshold more efficiently. In Layer II, the digitized audio signal is divided into blocks of 1152 samples, with each block being encoded within one MPEG-1 audio frame. Therefore, an MPEG-1 audio stream consists of consecutive audio frames, each one containing a header and the encoded data. The header contains general information, such as MPEG Layer, sampling frequency, number of channels, etc. Although most of this information may be the same for all frames, MPEG decided to give each audio frame a header to simplify synchronization and bitstream editing.

All the previous information describes one variation of MPEG, MPEG-1 Audio. This variation represents the first phase of dealing with mono and two-channel stereo sound coding, at sampling frequencies commonly used for high-quality audio (48, 44.1, and 32 kHz). In addition, there is a second variation, MPEG-2 Audio, which includes three main points. The first point is the extension of MPEG-1 to lower sampling frequencies (16 kHz, 22.05 kHz, and 24 kHz), providing better sound quality at very low bit rates. The second point is the backward-compatible (BC) extension of MPEG-1 to multichannel sound,

supporting up to 5 full bandwidth channels plus one low-frequency enhancement channel. The MPEG-2 BC stream adheres to the structure of an MPEG-1 bitstream, meaning that an MPEG-2 BC stream can be read and interpreted by an MPEG-1 audio decoder. The third and last point is a new coding scheme called Advanced Audio Coding (AAC), which is more efficient and presents higher quality.

Today, the MPEG-1 Audio standard is the most widely compatible lossy audio format in the world, thanks to technical merits and excellent audio quality performance. Within the professional and consumer market, four fields of applications can be identified, namely broadcasting, storage, multimedia, and telecommunication.

2.3 MPEG-1/2 Layers I/II IP Cores

Knowing the wide range of customers that need digital audio, belonging to all industries, this work proposes developing an IP core to encode MPEG-1/2 Layer II, using a RISC-V processor and hardware accelerators.

An intellectual property core (IP core) consists of a block of logic or data that is used in a semiconductor chip when making a field-programmable gate array (FPGA) or application-specific integrated circuit (ASIC) [13]. Therefore, IP cores are usually the property of a particular person or company, being created throughout the design process and eventually turned into components for reuse. Third-party IPs can also be purchased and implemented into designs.

Ideally, an IP core should be entirely portable, meaning it should be possible to insert it into any vendor technology or design methodology. However, this is not always the case, existing two main categories of IP cores, soft IP core, and hard IP core. A soft IP core is generally offered as a synthesizable RTL model. It is developed in a hardware description language like SystemVerilog [18] or VHSIC Hardware Description Language (VHDL) [19], or can occasionally be provided synthesized with a gate-level netlist. One advantage of this IP is the possibility to customize during the physical design phase and map to any process technology. A hard IP core has logic and physical implementation, meaning that its physical layout is finished and fixed in a particular process technology. One advantage of this core is the better predictability of chip timing performance and area for its technology.

A company that purchases an IP core license usually receives everything that is required to design, test, and implement the core in its product. It may also receive logic and test patterns, signal specifications, design notes, and a list of known bugs or limitations.

Two IP Cores, two Chips, and one Software that perform MPEG-1/2 Layer I/II audio encoding are presented in the following subsections.

2.3.1 CWda74

The *CWda74* [5] is an audio IP core capable of encoding one audio stream in real-time, provided by *Coreworks, S.A.* [20].

This IP core contains the MPEG-1/2 Layer I/II encoder software and the *Coreworks* processor-based hardware audio engine platform (*CWda1011*). Initially, the software is compiled into a binary file, which can be automatically boot-loaded through one of the control interfaces, parallel Advanced Microcontroller Bus Architecture (AMBA) [21] Advanced Peripheral Bus (APB) or serial Serial Peripheral Interface (SPI). Once the software is loaded, the program runs on the audio engine platform. The system can be configured, controlled, and monitored through a configuration, control, and status register file, accessed by the control interfaces. The Audio Input and Output Interfaces use a native parallel interface. Other standard audio interfaces, such as Inter-IC Sound (I2S)/Time Division Multiplexed (TDM) and Sony/Philips Digital Interface (SPDIF), are also available. The Memory Interface can be AMBA Advanced eXtensible Interface (AXI) (for ASIC or Xilinx FPGA), Avalon (for Altera [22] FPGA), or Memory Interface Generator (MIG) (for Xilinx FPGA).

The *CWda74* IP core delivers Program binary, Software manual, Netlist or RTL, Implementation constraints, and Hardware datasheet. As attributes, it presents low operation frequency and low power consumption, with the possibility of being optimized to fulfill different design specifications. Table 1 describes the main features.

Features
ISO/IEC 11172-3 and 13818-3 standards
Fraunhofer IIS high-quality software
Mono, dual mono, stereo, and joint stereo channel modes
16, 22.05, 24, 32, 44.1, and 48 kHz sampling rates
16 to 24-bit input audio resolution
300 kB external memory requirement
Configurable output latency
1 frame minimum latency
Control, configuration, and monitoring protocol
Real-time operation @75 MHz

Table 1: *CWda74* features.

2.3.2 IPB-MPEG-SE

The *IPB-MPEG-SE* [6] is an audio IP core capable of encoding up to two stereo audio streams in real-time, provided by *IPbloq* [23].

This IP core is designed to run on the *IPbloq* audio engine platform *IPB-PLAT*, which supports the encoding and decoding of multiple streams in multiple formats, on a single device. More precisely, the *IPB-MPEG-SE* software requires an instance of the *IPB-PLAT* audio engine platform with only one processor.

Initially, the program is uploaded using a hardware interface. Then, the system is configured, run, and monitored through a configuration, control, and status register file, accessed by the same Control Interface. The Audio Input and Output Interfaces include a native parallel interface. Other interfaces, such as I2S/TDM and SPDIF/Audio Engineering Society 3 (AES3), are also available.

The *IPB-MPEG-SE* IP core delivers Program binary, Software manual, RTL of FPGA netlist, Implementation constraints, and Hardware datasheet. As attributes, it presents low operation frequency, low power consumption, and compact hardware implementation, fitting economically in FPGAs and ASICs. In terms of features, this IP core is very similar to the *CWda74*, extending the Real-time operation for two audio streams @150 MHz.

2.4 MPEG-1/2 Layers I/II Chips

2.4.1 CX23415 Codec

The *CX23415* [1] is a low-cost, full-duplex MPEG-2 codec that integrates the functionality of several Integrated Circuits (ICs) in a single device, provided by *Conexant Systems, Inc* [24].

This chip was the first device to deliver MPEG-2 audio/video encoding and decoding, transport stream (TS) generation, and on-screen display control in a single chip. The ability to incorporate up to five different chip functionalities allowed for reducing the cost of designing and manufacturing digital audio and video products.

For audio encoding and decoding, the *CX23415* integrates MPEG-1 Layer II, with sampling rates of 32 kHz, 44.1 kHz, and 48 kHz, and compressed bit rates up to 448 kbit/sec. The encoder supports 16-bit samples, while the decoder supports 16-, 18-, or 20-bit outputs.

As audio input and output, this chip supports Stereo Sony I2S. As MPEG input and output, it supports Peripheral Component Interconnect Direct memory access (PCI DMA) master or PCI slave, 8-bit parallel program data, 8-bit parallel SPI transport data, and 1-bit serial transport data.

The *CX23415* most relevant features are high-quality real-time encoding and MPEG-1 and MPEG-2 support.

2.4.2 Futura II ASI+IP™

The *Futura II* [3] is a broadcast-oriented MPEG-2/H.264 encoder that supports all standard broadcast formats, including North American standards.

This device, developed by *Magnum Semiconductor Inc.*, is capable of encoding MPEG-1 Layer II at 192, 224, 256, 320, and 384 Kbps, with sampling rates of 32, 44.1, and 48 kHz. It can also encode Dolby Digital-3 (AC-3), MPEG-4 Advanced Audio Codec – Low Complexity (AAC-LC), and High-Efficiency Advanced Audio Coding (HE-AAC).

As analog audio input, it supports one Stereo (two channels) with a frequency range from 20Hz to 20kHz. As digital audio input, it supports the Audio Engineering Society-European Broadcasting Union (AES-EBU) and Serial digital interface/High-Definition Multimedia Interface (SDI/HDMI) (H.264 only). As output, both ASI and IP ports deliver MPEG-2 with a bit rate from 3.4 to 19.39 Mbps.

The *Futura II*'s most relevant features are 800 milliseconds latency, user-selectable resolution and bit rate, and two encoding modes, Constant Bit Rate (CBR) and Variable Bitrate (VBR).

2.5 MPEG-1/2 Layers I/II Systems

The **MPEG-2 Encoder CW-4888** [2] is an MPEG-2 encoder that feeds video signals of analog program sources, like cameras and broadcasters, to digital broadcast networks.

Initially, this device receives the standard composite video and the associated sound signals. Then, it digitizes and compresses the input according to the MPEG-2 standard, outputting the result as Asynchronous Serial Interface (ASI) [25] or Internet Protocol (IP) [26] streams.

For audio, this device supports mono, dual, stereo, and joint stereo sound modes. The audio input signal is converted by a dual-channel encoder, which performs MPEG-1 layer I/II compression based on ISO/IEC 11172-3 [11]. The bit rate can be set between 32 and 448 kbit/s, with sampling frequencies of 33 kHz, 44.1 kHz, and 48 kHz.

The *CW-4888*'s most relevant features are FPGA circuitry and the option for two or four independent encoder units in one frame. As attributes, it presents extremely low power consumption, high reliability, and a long lifespan.

2.6 MPEG-1/2 Layers I/II Software

2.6.1 TooLAME/LAME

The *LAME* [27] is a high-quality MPEG Layer III audio encoder, licensed under the Lesser General Public License (LGPL) [28] and considered the best MP3 encoding software at mid-high and variable bitrates. As attributes, this software delivers better quality compared to all other encoders at most bitrates, better quality and speed compared to ISO reference software, and three different encoding modes (CBR, VBR, and Average Bit Rate (ABR)). This encoder is also free format and compilable as a shared library (on Linux/Unix) and Dynamic link library (DLL) [29] (on Windows).

Based on portions of *LAME* and ISO dist10 code, the *TooLAME* [30] was developed as a free software MPEG-1 Layer II audio encoder, written primarily by Mike Cheng. *TooLAME* became well-known and widely used for its particularly high audio quality, despite the existence of many MP2 encoders.

2.6.2 *TwoLAME*

Despite being unmaintained since 2003, the *TooLAME* software was directly succeeded by the *TwoLAME* [12] code fork, which is the focus of this work. Thus, *TwoLAME* is an optimized MPEG Layer 2 audio encoder, based on *TooLAME*, with its latest version (0.4) released in 2019. Table 2 describes the additional features not provided in the original *TooLAME*.

Features
Static and shared library (*libtwolame*)
Fully thread-safe
API similar to *LAME*'s (easy porting)
Front-end supports a wider range of input files
automake/*libtool*/*pkgconfig* based build system
Written in Standard C (ISO C99 compliant)

Table 2: *TwoLAME* features.

The *TwoLAME* repository includes a *simplefrontend* directory that contains a basic implementation of the software, written in C. It has a *simplefrontend.c* file, containing the *main()* function, and two other files, *audio_wave.c* and *audio_wave.h*. Figure 4 shows the pseudo code for the first part of *simplefrontend.c*, mainly consisting of initialization.

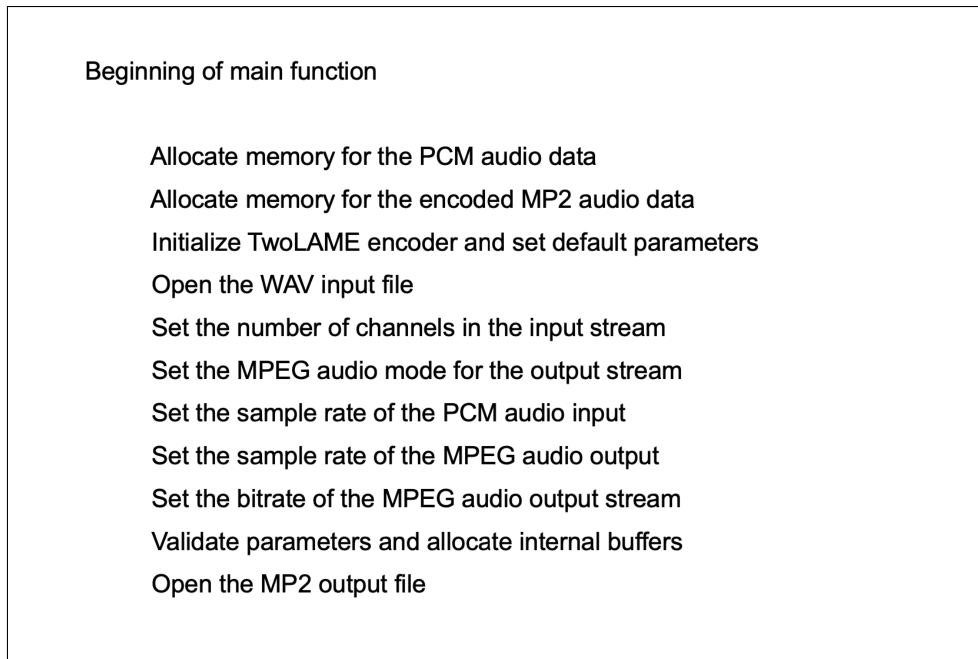

Figure 4: First part of *firmware.c* pseudo code.

The program starts by allocating memory for two different buffers to allow the encoding process. One of them is *pcmaudio*, which receives part of the original input file. This PCM buffer has a size of AUDIOBUFSIZE multiplied by two (corresponds to the number of bytes of short integer type), being allocated and initialized with zero by *calloc*. The other buffer is *mp2buffer*, which receives part of the encoded file that is later written in the output file. This MP2 buffer has a size of MP2BUFSIZE (corresponds to the number of bytes of unsigned char type), being allocated and initialized with zero by *calloc* as well.

Then comes the *TwoLAME*-related stuff. The first function is *twolame_init*, which initializes the encoding software by setting defaults for all parameters and returning a pointer necessary to all future *TwoLAME* calls [31]. The second function is *wave_init*, which parses the wave header. This function belongs to *audio_wave.c* and is responsible for processing the wave header (the first four bytes). Apart from identifying the file as WAVE, the header gives relevant information like sample rate and audio mode, which is collected in a *wave_info_t* struct. The remaining initialization functions specify encoding options. There is the *twolame_set_num_channels*, which sets the number of channels in the input stream. There is also the *twolame_set_mode*, which sets the MPEG Audio Mode (like mono or stereo) for the output stream. In addition, the *twolame_set_in_samplerate* sets the sample rate of the PCM audio input, while the *twolame_set_out_samplerate* sets the sample rate of the MPEG audio output. The *twolame_set_bitrate* sets the bitrate of the MPEG audio output stream.

After defining the main encoding options, *twolame_init_params* function is called. It prepares *TwoLAME* to start encoding by checking all the parameters, making sure they are valid as well as allocating buffers and initializing internally used variables. Then, *fopen* opens the output file where the encoded MP2 data is later written.

Moving to the *TwoLAME* execution, figure 5 shows the pseudo code for the second part of *simple-frontend.c*, mainly consisting of the encoding process.

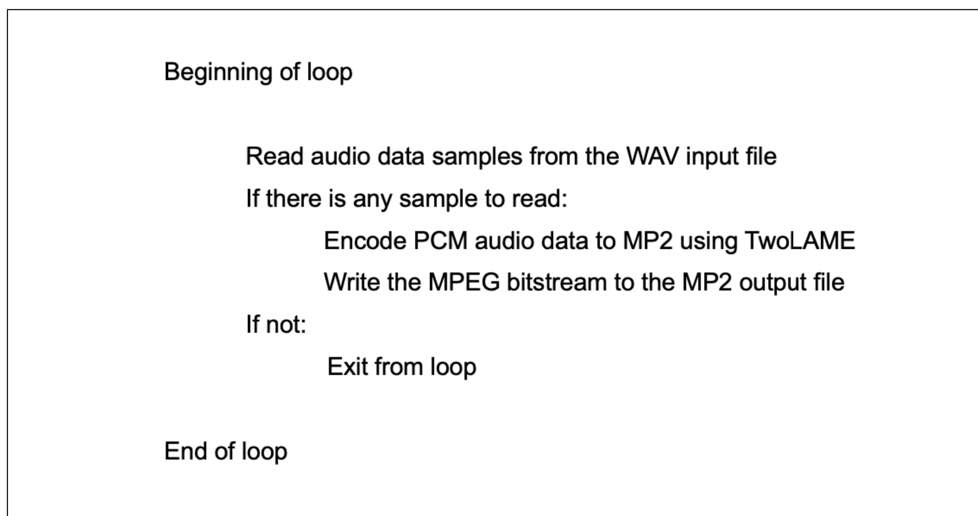

Figure 5: Second part of *firmware.c* pseudo code.

This process is based on a *while* loop, as the main idea is to encode one part of the input audio file at a time, repeating the loop as many times as required. Therefore, each loop iteration starts by reading a chunk of audio data from the input WAV file through *wave_get_samples*. This function also belongs to *audio_wave.c* and is responsible for reading AUDIOBUFSIZE samples to the PCM buffer (*pcmaudio*). With the buffer already loaded, *twolame_encode_buffer_interleaved* is responsible for encoding the audio data using TwoLAME [31]. This function has a high level of complexity, as expected. It makes use of the *libtwolame* library, which englobes many processes. In particular, the function is composed by *twolame_buffer_init*, which sets the *bit_stream* struct, and *encode_frame*, which encodes one audio frame at a time. After being encoded, the chunk of data is written in the output file through *fwrite*. The *frames* variable, which counts the number of frames encoded, is also updated.

Lastly, figure 6 shows the pseudo code for the third part of *simplefrontend.c*.

```
Encode any remaining PCM audio data to MP2
Write the MPEG bitstream to the MP2 output file
Shut down the TwoLAME encoder and free all allocated memory

End of main function
```

Figure 6: Third part of *firmware.c* pseudo code.

In this part, the *TwoLAME* software finishes execution and so does the IOb-SoC system. The first function is *twolame_encode_flush*, which encodes any remaining buffered PCM audio, i.e., any remaining audio samples in the PCM buffer [31]. This function is simpler than *twolame_encode_buffer_interleaved* and returns at most a single frame. The *fwrite* is used once again to write the remaining encoded data in the output file. Lastly, the encoding software is closed through *twolame_close* function. It shuts down the *TwoLAME* encoder and frees all memory that was previously allocated, including PCM and MP2 buffers.

2.7 System-on-Chip

A System-on-Chip (SoC) is an integrated circuit that combines components of an electronic system. These components usually include a Central Processing Unit (CPU), memory interfaces, on-chip input/output devices, input/output interfaces, and secondary storage interfaces. Putting many elements of a computer system on a single piece of silicon has some advantages, such as low power requirements, reduced cost, increased performance, and reduced physical size.

The IOb-SoC is a System-on-Chip template, comprising an open-source RISC-V [9] processor, which users can modify, simulate, and implement in ASICs [32] and FPGAs [13]. This SoC, provided by *IObundle, Lda*, supports stand-alone and boot-loading modes. It also allows an internal RAM [33] or an external Double Data Rate (DDR) [34] controller via an L1/L2 cache system.

Figure 7 shows a base IOb-SoC high-level block diagram.

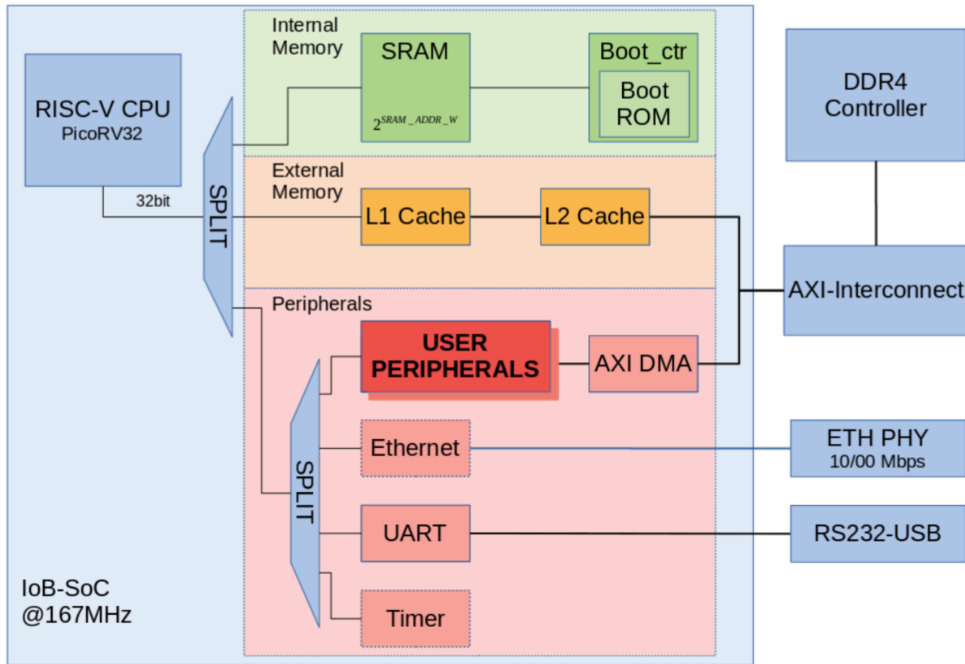

Figure 7: Base IOb-SoC high-level block diagram from IObundle [35].

2.7.1 IOb-SoC components

Starting with the **CPU**, the IOb-SoC uses a PicoRV32 CPU core, an open-source 32-bit processor that implements the RV32IMC instruction set, with an operating frequency of 167MHz.

As **Memory** subsystem, this SoC includes three main components. The Boot Read-Only Memory **Boot ROM** is a ROM used for booting the system. The Static Random Access Memory **SRAM** [36] is an internal memory that allows the system to run the program or the bootloader without the need for external memory. The **Cache** is an optional component that stores data from the external Double Data Rate (DDR) memory.

The communication between the CPU and the system's components (master and slaves) occurs through two buses. The **Memory bus** allows the CPU to communicate with the memory subsystem, while the **Peripheral bus** allows the CPU to communicate with system peripherals.

In addition, the **IOb-Interconnect** component is a bus switch responsible for the valid-ready handshake protocol between the CPU and all the peripherals. Based on the peripheral prefix from the address bus, this component selects which peripheral-specific bus connects to the CPU bus. The **IOb-Interconnect** multiplexes or demultiplexes, depending on the direction, the CPU bus signals (*valid*, *ready* and *rdata*). By doing so, the component creates enough signals to ensure that there is one of

each signal for each of the peripherals. Optionally, two peripherals connected to the peripheral bus can communicate directly without CPU intervention, allowing for faster data transfers between peripherals.

There is also a **Native to AXI adapter**, which allows communication between peripherals and memory controllers that use the AXI4 [21] protocol. Since the CPU only contains the native bus, the IOb-SoC uses this component to convert the signals from one bus to the other, with each peripheral that uses the AXI4-Lite port containing one **Native to AXI adapter**.

Finally, The Universal Asynchronous Receiver-Transmitter (**UART**) peripheral allows the SoC to communicate with external systems, through the RS-232 serial communication protocol.

All these components are integrated into IOb-SoC as GitHub submodules. They belong to repositories tracked by GitHub and forked from *IObundle, Lda*, but arranged in a different way and with specific names.

2.7.2 IOb-SoC deliverables and resources

As deliverables, the IOb-SoC includes Hardware Description Language (HDL) source code, software C source code, simulation testbench, implementation constraints for map, place, and route, demo files, and user documentation for system integration.

Tables 3 and 4 show the implementation resources used for *Xilinx Kintex Ultrascale Devices* and *Intel Cyclone V Devices*, respectively, according to the product brief.

Resource	Usage
LUTs	1869
Registers	1029
DSPs	4
BRAM	5
PIN	6

Table 3: Implementation resources for *Xilinx Kintex Ultrascale* Devices.

Resource	Usage
ALM	1335
FF	1177
DSP	3
BRAM blocks	22
BRAM bits	165,888
PIN	6

Table 4: Implementation resources for Intel Cyclone V Devices.

2.7.3 IOb-SoC repository

The IOb-SoC *GitHub* repository is composed of specific directories and segments (Tool Command Language (TCL), Verilog, Makefile, etc), as briefly described in the list below.

document/ This directory supports all the documentation, from LaTeX code and scripts to generated PDFs, like the guide and product brief.

hardware/ This directory supports the hardware layer, containing multiple subdirectories.

– **hardware.mk** This makefile contains targets, dependencies, and rules related to hardware.

– **fpga**/ This directory contains scripts to synthesize and run the system in FPGAs.

fpga.mk This makefile contains targets, dependencies, and rules related to FPGAs.

– **simulation**/ This directory contains scripts to run RTL simulation in simulators.

simulation.mk This makefile contains targets, dependencies, and rules related to simulators.

– **src**/ This directory contains Verilog scripts related to the system's components.

software/ This directory supports the software layer, containing multiple subdirectories.

– **software.mk** This makefile contains targets, dependencies, and rules related to software.

– **firmware**/ This directory contains the software to run in the system, after initialization.

– **bootloader**/ This directory contains the bootable software for the system.

– **pc-emul**/ This directory contains scripts to run the firmware on the computer.

– **console**/ This directory contains the software that allows interaction between computer and FPGA, via UART.

submodules/ This directory contains all submodules and peripherals used in the system. While the peripherals are added to the peripheral bus, the submodules only represent components that are integrated into the system.

config.mk This makefile contains targets, dependencies, and rules related to the SoC configuration, like the list of peripherals.

2.8 *Versat*

Versat is a Coarse-Grained Accelerator designed for embedded systems. It addresses the challenge of accelerating compute-intensive inner loops in code while efficiently managing the transitions between

code suitable for the Coarse-Grained Accelerator and code that needs to run on the host processor [37]. A list of the key features is presented below:

- **Partial Reconfiguration**: *Versat* supports partial reconfiguration by using a configuration register file and a configuration memory. This approach allows for random access to configuration fields and offers flexibility in configuration management.
- **Integration with Embedded Systems**: *Versat* cores serve as co-processors within embedded systems, working alongside application processors. They optimize performance and energy efficiency for compute-intensive tasks. Application programmers can access *Versat*'s capabilities through a dedicated API library.
- **Compiler**: *Versat* acts like a compiler, facilitating the efficient acceleration of tasks. The programming language syntax is a subset of C/C++, with hardware data descriptions, enabling programmers to leverage *Versat*'s power while keeping software tools separate from application processor tools.

2.8.1 Functional units

In practical terms, *Versat* contains several functional units (FUs) in its source, all written in Verilog. Nonetheless, the user has the possibility of creating and adding new ones to the source. A brief description of each functional unit is provided in the table below.

FU	Description
Const	This module outputs a 32-bit value configurable by the CPU.
FloatAdd	This module receives two 32-bit floating-point inputs, adds them together, and outputs a 32-bit floating-point value.
FloatSub	This module receives two 32-bit floating-point inputs, subtracts one from the other, and outputs a 32-bit floating-point value.
FloatMul	This module receives two 32-bit floating-point inputs, multiplies them together, and outputs a 32-bit floating-point value.
FloatNot	This module receives a 32-bit floating-point input, negates the most significant bit (MSB), and outputs a 32-bit floating-point value.
Float2Int	This module receives a 32-bit floating-point input and converts it to a 32-bit integer output. It performs a conversion operation that truncates the fractional part of the floating-point input, effectively extracting the integer component. The resulting 32-bit integer output represents the integer part of the input floating-point number. In cases where the input is negative, the output will represent the floor of the absolute value of the input.
Mux2	This module receives two 32-bit inputs and one 1-bit control input. It operates as a 2-to-1 multiplexer, selecting one of the 32-bit inputs based on the 1-bit control input. If the control input is zero, the output will be the first 32-bit input; otherwise, the output will be the second 32-bit input.
Mem	This module contains an internal memory with two input and two output ports (True dual-port synchronous RAM). It offers a memory-mapped interface that allows the CPU to store and read data from the memory while the *Versat* accelerator is not running. Internally, this module contains two Address Generator Units (AGU) that generate the addresses used to access the memory, one for each port. The AGUs can be configured to output or store data (only one type per port, the same port cannot be configured to output and store data at the same time in one run).
LookupTable	This module contains an internal memory with two input and two output ports (Dual port synchronous RAM). It offers a memory-mapped interface that allows the CPU to store and read data from the memory while the *Versat* accelerator is not running. Internally, this module acts like a lookup table since the output is the value stored in the address given by the input.

Table 5: *Versat* functional units.

2.8.2 Operators

Versat also contains a set of operators defined in its specification source. The operators execute either over the right operand or between the left and right operands. In practice, this terminology just abstracts the implementation of the operators in *Versat*, through functional units.

A brief description of some operators is provided in the table below.

Operator	Description
$-$	This operator negates the right operand.
$\sim$	This operator performns binary one's complement of the right operand.
$\&$	This operator performs a logical AND between the left and right operands.
$\ll$	This operator shifts the bits of the left operand to the left by the number of positions defined by the right operand.
$\vert$	This operator performs a logical OR between the left and right operands.
$\wedge$	This operator performs a logical XOR between the left and right operands.

Table 6: *Versat* operators.

2.8.3 Syntax

Knowing the FUs and operators *Versat* provides, it is convenient to understand how the hardware design can be developed using such resources. Figure 8 shows a coding example of *versatSpec.txt*, the file where the *Versat* accelerator should be described.

```
module Example (op1, op2) {
    FloatMul mul;
#
    op1 -> mul:0;
    op2 -> mul:1;
    mul -> out;
}

module top () {
    Const A;
    Const B;
    Mem mem;
    Example ex;
#
    add = A + B;
    add -> mem:0;

    sub = A – B;
    sub -> mem:1;

    mem:0 -> ex:0;
    mem:1 -> ex:1;
    ex -> out;
}
```

Figure 8: *versatspec.txt* example.

Focusing on the syntax, the code is divided into two modules, *Example* and *top*, each representing a part of the hardware design.

The *Example* module is defined with two input ports, *op1* and *op2*, and a *FloatMul* FU, *mul*. The code after the # symbol specifies how data flows through the module. In this case, *op1* and *op2* are connected to input ports 0 and 1 of the *mul* FU, respectively. Then, the output of the *mul* FU is connected to the output port of the *Example* module.

The *top* module is the starting module, i.e. it represents the overall design. This module is defined with two *Const* FUs, *A* and *B*, and a *Mem* FU, *mem*. In addition, it also contains an instance of the *Example* module called *ex*. Once again, the code specifies how data flows through the modules. In this case, *A* and *B* are added together, and the result is stored in the *add* signal. Then, *add* is connected to input port 0 of the *mem* FU. *A* and *B* are also subtracted, and the result is stored in the *sub* signal.

Then, *sub* is connected to input port 1 of the *mem* FU. Afterward, the output ports 0 and 1 of *mem* are connected to the input ports 0 and 1 of *ex* instance, respectively. The *ex* instance performs what was described previously for the *Example* module, and its output is connected to the output port of the top module.

In terms of functionality, this example starts by performing arithmetic operations on constants *A* and *B*. Then, it sends the result of those operations to a memory unit, which works as an index for each input port (of the data array). Based on these indices, the memory unit outputs one value in each output port, which is then used in the following module to perform a floating-point multiplication operation. The final result is accessible through the output port of the top module.

3 Hardware architecture

This chapter addresses the hardware architecture. It starts by setting up *VexRiscv*, *Versat* and *Timer* in the IOb-MP2-E. Then, it describes two hardware accelerators that were developed to accelerate the *psycho_3_threshold* function. In the end, a system overview is done.

In the initial stage of this work, the IOb-MP2-E system comprised eight main components: **AXI**, an AXI interconnect protocol; **CACHE**, a high-performance Verilog cache; **LIB**, a set of *Verilog* macros; **MEM**, a set of memory Verilog descriptions; **PICORV32**, a RISC-V processor; **TWOLAME**, an optimized MP2 encoding software; **UART**, a UART core; **DDR4 Controller**, a controller for DDR memory (figure 9). As previously stated, the ultimate goal of this work is to develop a system that beats the only MPEG1/2 Layer II encoding IP core on the market, the *CWda74* [5]. This system uses fixed-point precision, and so the most logical approach to the problem turns out to be implementing the *TwoLAME* algorithm using floating-point precision. Looking at the **PICORV32** CPU available, porting the software in these conditions would be undoable, due to its fixed-point arithmetic limitation.

3.1 VexRiscv

For this reason, the current CPU has to be substituted, and the **VEXRISCV** becomes the premier pick, as it includes a floating-point unit (FPU) [38]. In this process, the *PICORV32* was removed from the IOb-MP2-E and the *VEXRISCV* was added as the only CPU (figure 9). This was straightforward, as *VEXRISCV* was previously implemented and tested by *IObundle, Lda*. Apart from adding the submodule to IOb-MP2-E's repository, some paths were changed to allow the correct compilation of the CPU. Interrupt signals were also added to the system data bus, such as *timerInterrupt*, *softwareInterrupt*, and *externalInterrupt*.

3.2 Versat

Given *Versat* being the optimal choice for accelerating *TwoLAME* algorithm, integration into IOb-MP2-E becomes imperative. Therefore, the first step in this process was adding **VERSAT** to the IOb-MP2-E (figure 9). This was simple as *VERSAT* was previously implemented and tested by *IObundle, Lda*. Apart from adding it as a submodule to IOb-MP2-E's repository, some paths were changed to allow correct compilation. The second step was more complex, consisting of adding an AXI interconnection between *Versat* and IOB-MP2-E's external memory (DDR4 SDRAM). The system usually contains a single AXI [39] interconnect instance, used by external memory and CPU. Therefore, a pragmatic solution

was to double the wire size of the existing AXI interconnection, providing communication from/to *Versat* while keeping the same AXI instance.

Focusing on *Versat* functionality and interfaces, *versatSpec.txt* functions like a code script, while *Versat* operates akin to a compiler. In other words, *Versat* takes the code, described in a Domain Specific Language (DSL), and generates not only the accelerator (in Verilog) but also the necessary headers and C source code. The generated code empowers the firmware to govern the accelerator effectively, i.e. it ensures correct correspondence between CPU actions and the accelerator.

Versat generates hardware accelerators according to the dataflow paradigm. The algorithm is implemented by instantiating functional units (specified in *versatSpec.txt*) and connecting them to execute the intended part of the algorithm. *Versat* handles data validity transparently from the user, inserting buffer units between connections to ensure data from convergent paths arrive at the same time when needed. Regarding the accelerator's internal registers, some of these registers are constant and shared by all accelerators, such as Direct Memory Access (DMA) and control registers. On the opposite, others depend on the specific functional units present in the accelerator. The FUs need to implement certain interfaces recognizable by *Versat* in order to perform useful work. Most FUs implement input and output ports in order to connect with other FUs. *Versat* gives one the possibility to create and add new Functional units to the source. Furthermore, FUs can implement a variety of interfaces that allows them to receive or send data to outside the accelerator:

- **Configuration:** Inputs that allow the CPU to modify and control the functionality of individual units. A configuration register connects to the configuration entries of each unit, conditioning its behavior.
- **State:** Outputs that allow the CPU to read data from the units. Useful for small amounts of data transfer.
- **Memory-Mapped:** Allocation of specific address space for units, allowing access to them through read and write operations. This interface is primarily used for memory units, enabling the CPU to read and write data as required.

The accelerator generated by *Versat* groups all these interfaces into a single memory-mapped interface, accessed by the CPU. When the CPU accesses an address inside the accelerator memory space, the CPU can be accessing a control register of the accelerator, the configuration, or the state registers (which are connected to the configuration and state interfaces of their specific units) or it can be accessing the FUs directly through the memory mapped interface. Moreover, a DMA mechanism is used for efficient data transfers between the accelerator and external memory, specifically between the accelerator's memory-mapped interfaces and DDR. This DMA is configured by the CPU. In practice,

the accelerator loads memories such as *LTtm*, *LTnm*, *freq_subt*, and $mem \rightarrow dbtable$ using the DMA, enabling efficient and high-speed data movement.

3.3 Timer

To allow profiling the *TwoLAME* algorithm, another component was added to IOb-MP2-E, the **TIMER** [40]. This peripheral is a 64-bit hardware timer, equipped with reset, enable, and reading functions. It includes a software driver and an example C application, which helps understand how it works. Similarly to the other components, the TIMER is tracked by a *GitHub* repository (forked from *IObundle, Lda*), integrating the IOb-MP2-E as a submodule. This component was previously implemented and tested by *IObundle, Lda* as well. In addition, some paths were added to allow correct compilation. Figure 9 shows a high-level block diagram of the IOb-MP2-E system at this point.

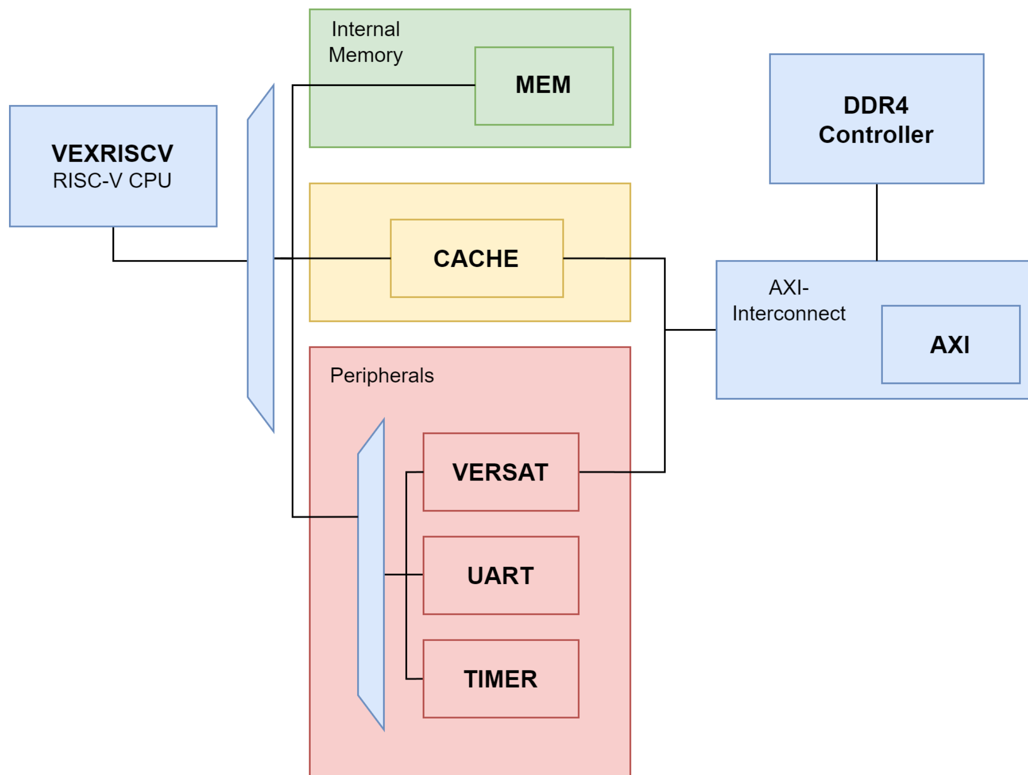

Figure 9: High-level block diagram of the IOb-MP2-E in use.

Based on the results of *TwoLAME*'s profiling, shown in *Results* section, the subsequent step involves developing the hardware accelerators for the *psycho_3_threshold* function, using *Versat*. The *psycho_3_threshold* function contains three for loops, with the first one being a simple initialization process, setting each element of arrays *LTtm* and *LTnm* to a constant value (*DBMIN*) within the range of 0 to 135 ($SUBSIZE - 1$). Performing this initialization in hardware incurs unnecessary overhead, as the

result remains constant for all iterations. Instead, in a hardware implementation, we can directly use the constant value DBMIN as an input for the subsequent hardware operations (the next for loops, in this case), optimizing efficiency and reducing the need for a dedicated initialization loop. Therefore, only two accelerators should be developed to execute the second and third for loops of *psycho_3_threshold*. The third for loop is simple as it just includes the *psycho_3_add_db* function. On the opposite, the second for loop is not only complex but it is also more interesting. The main for loop (outer loop) contains two *if* conditions and, inside each condition, there is another for loop (inner loop). Moreover, the inner loop is the same in both conditions, differing only on part of the input data. This means that it is only necessary to develop hardware that executes both inner loops, with the if conditions and the outer loop being handled by the CPU.

Before developing the accelerators, it is crucial to determine the control and data paths of the loops that are intended to be accelerated. In other words, the paths refer to the fundamental components that dictate how the software interacts with the hardware to achieve acceleration. The control path defines the flow and sequencing of operations within the software that need to be accelerated. It includes decisions, branching, loops, and other control structures that determine the program's behavior [41]. Determining the control path is vital for optimizing the hardware design to efficiently execute these operations. The data path refers to the route through which data flows within the software during its execution. It involves operations and transformations applied to the input data to produce the desired output. Understanding the data path is crucial for designing hardware components that can process and manipulate the data effectively to accelerate the software's performance.

3.4 ***spectrum_search*** accelerator

3.4.1 Control and Data paths

This section shows both control and data paths of the first accelerator, *spectrum_search*, corresponding to the second for loop in the original *psycho_3_threshold* function.

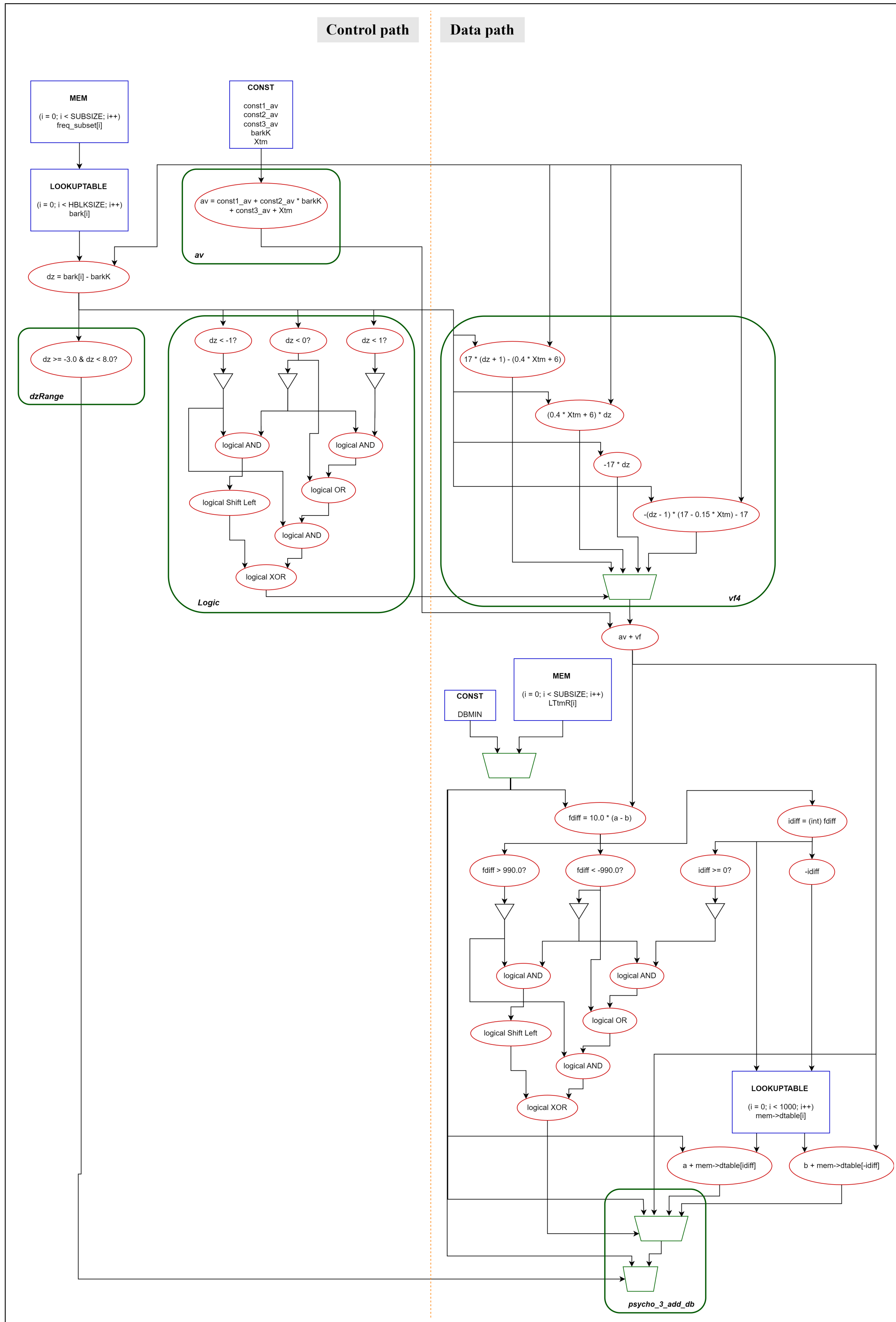

Figure 10: Control and data paths of the *spectrum_search* hardware accelerator.

3.4.2 Functional units

Based on the previous Control and Data path, five FUs were designed to allow developing the *spectrum_search* accelerator (and also *masking_threshold*), such as *FloatLess*, *FloatGreater*, *FloatGreaterEqual*, *Mux4* and *Conditional1*.

A brief description of each functional unit is provided in the table below.

FU	Description
FloatLess	This module receives two 32-bit floating-point inputs and outputs a 32-bit value (repeated 1-bit result 32 times) indicating whether the first input is less than the second. It performs a comparison operation to determine if the first input is less than the second. If the first input is less than the second, the output will be a 32-bit value with all bits set to 1; otherwise, the output will be a 32-bit value with all bits set to 0.
FloatGreater	This module receives two 32-bit floating-point inputs and outputs a 32-bit value (repeated 1-bit result 32 times) indicating whether the first input is greater than the second. It performs a comparison operation to determine if the first input is greater than the second. If the first input is greater than the second, the output will be a 32-bit value with all bits set to 1; otherwise, the output will be a 32-bit value with all bits set to 0.
FloatGreaterEqual	This module receives two 32-bit floating-point inputs and outputs a 32-bit value (repeated 1-bit result 32 times) indicating whether the first input is greater than or equal to the second. It performs a comparison operation to determine if the first input is greater than or equal to the second. If the first input is greater than or equal to the second, the output will be a 32-bit value with all bits set to 1; otherwise, the output will be a 32-bit value with all bits set to 0.
Mux4	This module receives four 32-bit inputs and one 32-bit control input. It operates as a 4-to-1 multiplexer, selecting one of the 32-bit inputs based on the two least significant bits (LSB) of the control input. The two LSB of the control input can be '00', '01', '10', or '11', selecting the first, the second, the third, or the fourth input, respectively. The output will be the 32-bit input that was selected.
Conditional1	This module receives two 32-bit inputs and one 32-bit control input. It operates as a 2-to-1 multiplexer, selecting one of the 32-bit inputs based on the 32-bit control input. If the LSB of the control input is zero, the output will be the second 32-bit input; otherwise, the output will be the first 32-bit input.

Table 7: New *Versat* functional units.

3.4.3 Modules

After analyzing the control and data paths, the hardware accelerator is developed in *versatSpec.txt*. This file specifies the whole data process, which can be divided into several modules each representing a certain functionality or output. The starting module is called *start* and invokes all the other modules.

A brief description of each module is provided in the list below.

av

1. Float multiplication (*mul1*):
    - Takes *const2* and *bark* as inputs.
    - Multiplies *const2* and *bark*.
2. Float addition (*add1*):
    - Takes the output of *mul1* and *const1* as inputs.
    - Adds *mul1* and *const1*.
3. Float addition (*add2*):
    - Takes the output of *add1* and *Xtm* as inputs.
    - Adds *add1* and *Xtm*.
4. Float addition (*add3*):
    - Takes the output of *add2* and *const3* as inputs.
    - Adds *add2* and *const3*.
5. Output (*out*):
    - Takes the output of *add3* as the final output of this module.

dzRange

1. Greater or equal comparison (*ge1_dzRange*):
    - Compares the input *dz* with *const1_dzRange* for greater than or equal condition.
2. Less than comparison (*lt1_dzRange*):

- Compares the input *dz* with *const2_dzRange* for less than condition.

3. Logical AND operation (*and*):
    - Performs a logical AND operation on the outputs of *ge1_dzRange* and *lt1_dzRange*.
4. Output (*out*):
    - Takes the output of *and* as the final output of this module.

Logic

1. Less than comparison (*lt1_Logic*, *lt2_Logic*, *lt3_Logic*):
    - Compares the input *dz* with *const1_Logic*, *const2_Logic* and *const3_Logic* for less than conditions.
2. Bitwise NOT, AND and Shift Left operations:
    - Performs bitwise NOT, AND, and Shift Left operations with the previous outputs to calculate *one*.
3. Bitwise AND, OR, and Shift Left operations:
    - Performs bitwise AND, OR, and Shift Left operations with the previous outputs to calculate *zero*.
4. Bitwise XOR operation (*sel*):
    - Combines the previous outputs using bitwise XOR to obtain the final selection.
5. Output (*out*):
    - Takes the output of *sel* as the final output of this module.

vf4

1. Calculation of $0.4 * Xtm + 6$:
    - Multiplies *Xtm* by *const3_vf4*.
    - Adds *const4_vf4* to the result.
2. Calculation of $17 * (dz + 1) - (0.4 * Xtm + 6)$:

- Adds 1 to the input *dz*.
- Multiplies the result by *const1_vf4*.
- Subtracts the output of step 1 from the previous result.

3. Calculation of $(0.4 * Xtm + 6) * dz$:

   - Multiplies the output of step 1 by the input *dz*.

4. Calculation of $-17 * dz$:

   - Multiplies the input *dz* by *const5_vf4*.

5. Calculation of $-(dz - 1) * (17 - 0.15 * Xtm) - 17$:

   - Subtracts 1 from the input *dz*.
   - Multiplies the previous result by *const6_vf4*.
   - Multiplies *Xtm* by *const1_vf4*.
   - Subtracts the output of step 5 from the previous result.

6. Conditional selection based on *logic1*:

   - Uses a multiplexer *mux4_vf4* to select one of the previous outputs based on the input *logic1*.

7. Output (*out*):

   - Takes the output of *mux4_vf4* as the final output of this module.

psycho_3_add_db

1. Input multiplexing (*mux4_psycho_3*):

   - Uses a multiplexer to select one of the inputs (*a*, *b*, *add1*, *add2*) based on the selection signal *sel*.

2. Conditional selection using if condition (*conditional_psycho_3*):

   - Uses a conditional block to select *a* or the output of *mux4_psycho_3* based on the input *if*.

3. Output (*out*):

   - Takes the output of *conditional_psycho_3* as the final output of this module.

start

This is the main module, which is responsible for invoking all the other modules. It interconnects all modules by setting the correct inputs and outputs, and it also executes simple logical operations.

3.4.4 Control

Apart from specifying the data process, setting all the input data for the hardware accelerator is also required. The inputs can either be simple constants or data arrays that are streamed by a memory unit, sequentially. The inputs can also be stored in a memory unit and accessed randomly.

As mentioned in the *psycho_3_threshold* function section, the first accelerator executes both inner loops in the second for loop of *psycho_3_threshold*. Therefore, this accelerator executes several runs, i.e. there is the need to suplly the accelerator with at least two sets of input data. This is done in two separate functions because part of the data is valid for all the runs. The part of the data that doesn't change (between runs) belongs to the *initVersat* function, while the remaining part belongs to *configureVersat* function.

In *initVersat* the constants (that don't change between runs) are set by assigning the intended value to the constant name (in the source files generated by *Versat*). For floating-point data, the *PackInt* function is required, which receives the intended value as argument. There is also a memory unit setting for *freq_subset* array. This sets an internal memory with each position of the array, from 0 to SUBSIZE, read by the CPU through *VersatUnitWrite* function, sequentially. There are also two memory unit settings for *bark* and *mem→dbtable* arrays. These set two lookup tables, with SUBSIZE and HBLKSIZE, respectively.

In *configureVersat* four constants are set as they differ between runs. An input of *mux2* multiplexer is also set based on a flag that is handled by the CPU. At this point, all the input data is set for the hardware accelerator, and so it starts the execution through *RunAccelerator* functions. This function receives as an argument the number of times that the accelerator should run with the same settings, which is 1.

3.5 *masking_threshold* accelerator

3.5.1 Control and Data paths

This section shows both control and data paths of the second accelerator, *masking_threshold*, corresponding to the third for loop in the original *psycho_3_threshold* function.

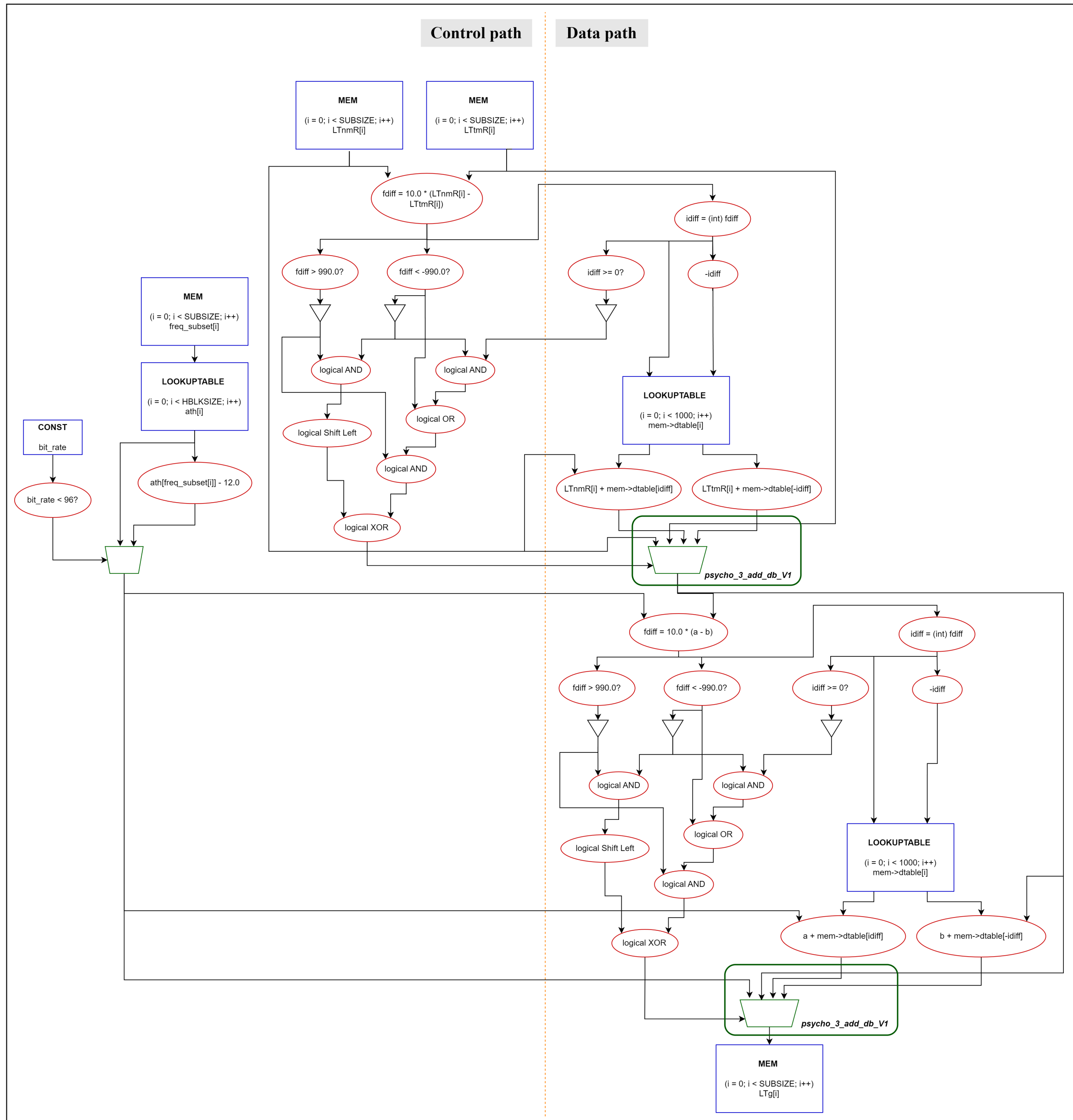

Figure 11: Control and data paths of the *masking_threshold* hardware accelerator.

3.5.2 Modules

psycho_3_add_db_V1

1. Input Multiplexing (*mux4_psycho_3*):

    - Uses a multiplexer *mux4_psycho_3* to select one of the inputs (*a*, *b*, *add1*, *add2*) based on the selection signal *sel*.

2. Output Selection (out):

   - Takes the output of *mux4_psycho_3* as the final output of this module. The selected input is determined by the value of *sel*.

start1

This module has some similarities with the *start* module of the first accelerator since it contains most of the *start* module description but is duplicated. It invokes and interconnects all the other modules, apart from executing simple logical and conditional operations.

3.5.3 Control

As mentioned in the *psycho_3_threshold* function section, the second accelerator executes the third for loop of *psycho_3_threshold*. The control of this accelerator is done inside a single function since it performs only one run. Therefore, all the settings belongs to *configureVersat1* function.

In this function, the process is similar to the previous accelerator, since the *masking_threshold* accelerator does not contain any functional unit that is not present in the first one.

3.6 Overview

The process of creating IOb-MP2-E, an FPGA-based System on Chip, involves synthesizing and implementing the entire system configuration within the FPGA (the chip). For IOb-MP2-E, this configuration includes elements like *VEXRISCV* CPU, *UART*, *TIMER*, etc. The process defines how these hardware components are mapped and interconnected within the FPGA fabric (in Configurable Logic Elements (CLEs), Block RAM (BRAM), Look-Up Tables (LUTs), etc) ensuring that the CPU, memory, and peripherals work together seamlessly. Once the system configuration is established, the firmware becomes a vital component. The firmware is converted into machine instructions specific to the CPU architecture used in the IOB-MP2-E (which is RISC-V). After synthesis and compilation, the firmware is stored in a Block RAM (SRAM) in the FPGA, allowing high-speed access. When the FPGA is powered on or reset, the firmware is loaded from this memory and executed by the CPU. This process makes the IOb-MP2-E a self-contained, programmable system within the FPGA, with all its hardware components integrated into a coherent, functional unit.

An overview of the hardware setup employed in this work is presented below.

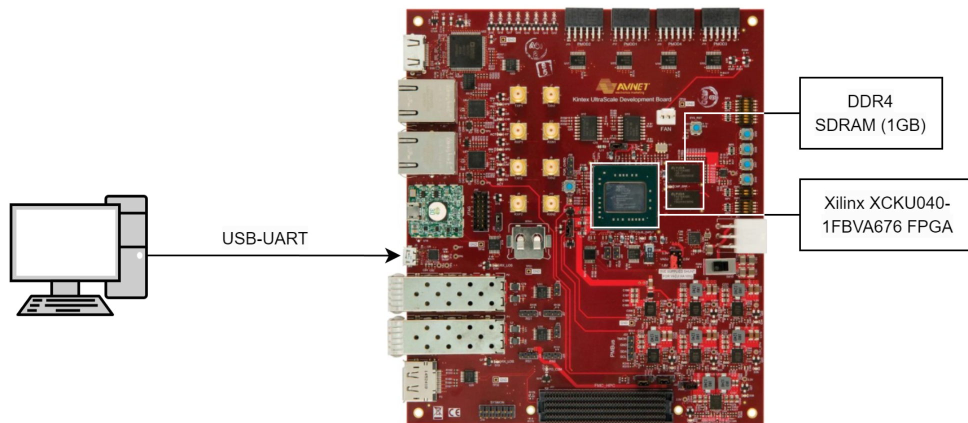

Figure 12: Hardware setup overview.

The relevant components are the Xilinx XCKU040-1FBVA676 FPGA and the DDR4 SDRAM, as explained previously. Nonetheless, a basic description of how the hardware setup works is presented below.

- **Personal Computer (PC):** The PC is responsible for the synthesis, compilation, and programming of IOb-MP2-E into the FPGA.
- **Xilinx Board:** The Xilinx board is an integral part of our hardware setup, housing the FPGA and related resources. This Xilinx board is where IOb-MP2-E is ultimately implemented and executed.
- **USB-UART Connection:** To facilitate communication between the PC and the Xilinx board, a USB-UART connection is employed. This connection is utilized for transferring the synthesized configuration bitstream from the PC to the FPGA.
- **Compilation and Synthesis Software:** On the PC, specialized compilation and synthesis software tools are used to design and compile IOb-MP2-E. These tools generate a configuration bitstream that specifies the functionality and behavior of IOb-MP2-E. Once generated, this bitstream is transferred to the Xilinx board via the USB connection.
- **FPGA Implementation:** The FPGA is responsible for the physical implementation of IOb-MP2-E.
- **IOb-MP2-E Execution:** Once the IOb-MP2-E is successfully implemented in the FPGA, it can be executed within the FPGA's logic fabric. The SoC interacts with other components on the Xilinx board, such as DDR memory, to carry out the intended computational tasks.

4 Software architecture

This chapter addresses the software architecture. It starts by generating audio test files. Then, it configures the firmware to allow execution of *TwoLAME* encoding algorithm. Afterward, basic software optimizations are done in the system. In the end, profiling analysis defines which part of *TwoLAME*'s algorithm requires hardware acceleration.

Before being ported to the IOb-MP2-E, the *TwoLAME* software was tested in a Linux [42] environment, allowing one to verify its functionality independently before integration. This verification had a rudimentary character since the *TwoLAME* software had already been tested and approved by a wide range of users in various systems. Therefore, the verification consisted of listening to both an original audio file and the corresponding encoded file, produced by *TwoLAME*.

4.1 Audio test files

In this context, *Audacity* software [43] was used to generate four audio files. This free open-source software, compatible with multiple operating systems, is capable of multi-track audio editing and recording. The creation of multiple testing files was less about functionality verification and more about preparing for the algorithm's testing in the upcoming phases of this work, motivated by the following ideas:

- **Variation in Audio Characteristics**: Testing files with diverse audio characteristics, including codec variations, encoding settings, and audio durations, allow a comprehensive assessment of *TwoLAME*'s adaptability.
- **Consistency Across Specifications**: Maintaining consistent software performance across various specifications is crucial for a real-time encoding IP core. Using multiple test files ensures that *TwoLAME* handles diverse audio sources, guaranteeing reliable real-time encoding performance regardless of input variations.

The first generated file was *short.wav*, using the *Generate Tone* option in *Audacity*. This mono audio has a sine waveform, 44.1kHz sampling rate, 16 bits per sample, and a size of 27KB (the smallest audio file in the repository). The second generated file was *long.wav*, using the *Generate Rhythm Track* option configured with *Metronome Tick* beat sound in *Audacity*. This mono audio has a 44.1kHz sampling rate, 16 bits per sample, and a size of 683KB. The third generated file was *noise.wav*, using the *Generate Noise* option configured with *White* noise type in *Audacity*. This audio has a 44.1kHz sampling rate, 16 bits per sample, a size of 529KB and, unlike the previous ones, it is stereo. The mono default track was duplicated, with one being distributed 100% to the Left and the other being distributed 100% to the Right

(*panning effect*). The fourth and last generated file was *vivaldi.wav*. This file was not generated from scratch in *Audacity*. Instead, it was simply converted to WAV format. The original file, entitled 'Vivaldi - Spring', was downloaded from the Internet (for free) and opened on *Audacity*. After that, the track was reduced to the initial 4 seconds and exported as WAV, being also stereo. What motivated the use of this file was the recognized quality and complexity of *Vivaldi*'s songs [44]. Testing with both mono and stereo files is crucial for assessing versatility and performance across various scenarios. Mono files are valuable for evaluating bandwidth efficiency and compatibility with single-channel content, while stereo files help assess the encoder's ability to preserve spatial characteristics and deliver an immersive listening experience.

At this point, it was already possible to verify the *TwoLAME* software using the audio files available at the repository. From a standard *automake* process, composed of *./configure*, *make*, and *make install* commands, *TwoLAME* was successfully installed in the environment. Then, by simply executing *./stwolame* followed by the input file name and the output file name (the one desired), the software would run and produce an MP2 version of the original file. In the process, both input and output file names should include the file extension. By executing the *stwolame short.wav short.mp2* command, it was verified that both the original and the encoded audio files sounded the same (for a human listener). The same happened with the second and fourth files through a similar command. The third file is not audible, in the way that it was created just to verify proper correctness in terms of values produced by the system (in upcoming phases). This is because white noise is the worst-case scenario for audio encoding, mainly due to its high requirements.

With the original *TwoLAME* software verified, the next step was porting *libtwolame* (*TwoLAME*'s library) to the IOb-MP2-E, which required a front-end interface.

4.2 Firmware

Based on the *simplefrontend* example present in *TwoLAME*'s repository, the front end was implemented in *firmware.c*. As the name indicates, this file specifies the firmware through some C code. More precisely, it specifies the firmware that runs on the CPU as a software application which, in this work, is intended to be the *TwoLAME* algorithm. To allow that, the firmware is converted into machine instructions that are specific to the RISC-V instruction set [45], being interpreted by *VEXRISCV* (more details in the *Hardware architecture* section). Figure 13 shows the pseudo code for the *main()* function in *firmware.c*.

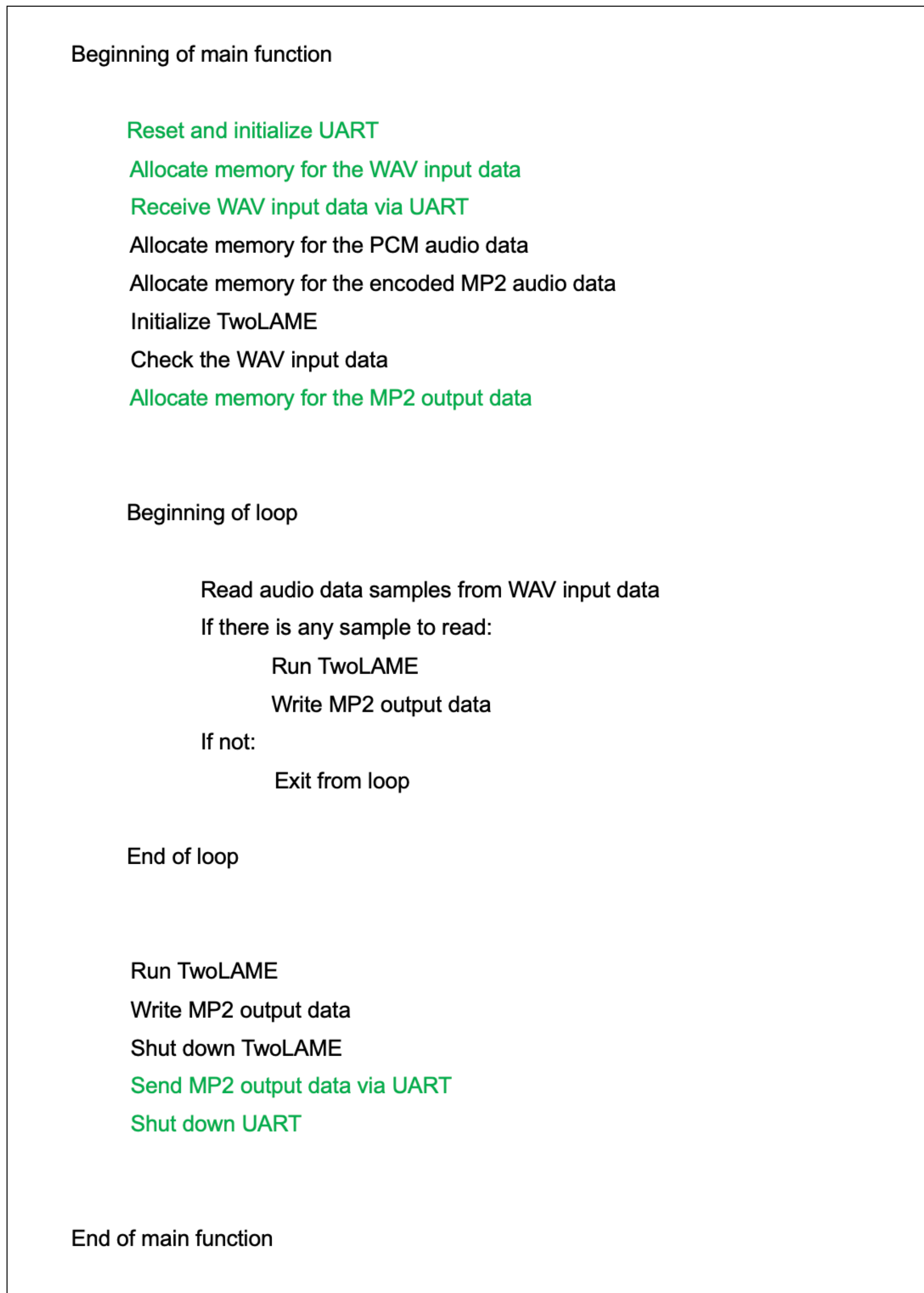

Figure 13: New *firmware.c* pseudo code.

The first difference in *main()* is the usage of *uart_init*, which resets *IObundle, Lda*'s UART peripheral and sets its division factor. This peripheral is needed to transmit data to IOb-MP2-E's external memory via UART. There are also two additional memory allocations for the input and output data. These operations are done through *malloc* using unsigned char type because the UART peripheral transmits one byte at a time. After allocating memory, the input audio data is received via UART in *uart_recvfile*. The function receives the path as an argument, which is defined as *test.wav*. This is a generic name

since the input audio file is copied to the compiling directory as *test.wav*. The previous modifications were done as the IOb-MP2-E does not have an operating system or file system. For this reason, the *fopen* function inside *wave_init* was removed. Furthermore, in each iteration of the *while* loop, a chunk of input data is read from the input data using *fread*. Since this function, present in *wave_get_samples*, also requires a file system, there should be some process to increment the pointer to the input data, so that *fread* can be removed. This is achieved by decrementing a global variable (unread_data) based on the number of samples read in each loop iteration. In addition, since the *pcmaudio* buffer should contain short integers, there is a process inside *wave_get_samples* that converts every two characters into a single short integer. This is done by concatenating two characters, with the first one shifted left eight positions.

Nonetheless, not only the input audio data is received but also the output audio data is sent from IOb-MP2-E's external memory, both via UART. Thus, the *fwrite* functions were also removed from *main*. More precisely, there was one situation inside the *while* loop and another after the loop, with both *memcpy* insertions being responsible for writing the MP2 output data, replacing *fwrite* calls. An important detail about this modification comes with the pointer to the output data. Just like in *wave_get_samples*, there should be a process to increment the pointer to the output data, so that the new data does not overwrite previous data. This process is based on the number of frames encoded in each loop iteration, specifically inside *TwoLAME_encode_buffer_interleaved* function. One of the last modifications in *main* consists of sending the output data via UART in *uart_sendfile*. The function receives the path as an argument, which is defined as '../../encoded.mp2'. This way, the output audio file is copied to the main repository directory as *encoded.mp2*. Afterward, the UART transmission is closed through *uart_finish*, and the program returns.

Apart from *main*, some header files were included in *firmware.c*, related to both the IOb-MP2-E system and *libtwolame*. Two macros were also defined, AUDIOBUFSIZE as 2304 and MP2BUFSIZE as 4096. These variables specify the size of input and output buffers, respectively, which in practice represent the chunk of data encoded each time (the number of frames encoded). Therefore, the number of frames can be altered by changing both variables on the same scale. As it stands, the *TwoLAME* encodes one frame at a time. After this process, the system was emulated on PC, passing through a simulation environment developed by *IObundle, Lda*, which allowed full testing.

4.3 Optimization

Regarding execution in FPGA, some basic software optimizations were performed. According to what was previously stated about the use of *VEXRISCV* CPU, the first change in *libtwolame* was the software precision. In *comon.h*, there is a macro for FLOAT that was initially defined as *double*, but it was changed

to *float*. One reason for this is the fact that *TwoLAME* should be faster rather than more accurate (single-precision is already much better than fixed-point), without compromising the encoded audio quality. Apart from altering the macro, one function in *subband.c* was also changed. The original code contained *modf* [46], a function that belongs to *math.h* [47] and requires double precision. Therefore, it was substituted by *modff* [46], which has the same functionality (returns the fractional part) but uses single precision.

The second change was more strategic, as it entailed a meticulous analysis of the encoding software. It is well-established that trigonometric functions [48] are computationally intensive operations, particularly for audio-processing software like *TwoLAME*. These functions, while mathematically accurate, can be computationally expensive and can slow down the encoding process. Therefore, a common practice in software engineering is to substitute this type of function with precomputed lookup tables, for specific input ranges. This allows the software to access and interpolate results much more quickly than it could by performing complex calculations on the fly. To evaluate this possibility, the *TwoLAME* was emulated on PC using different input test files.

The first case was *psycho_3_powerdensityspectrum*, in *psycho_3.c*. In this function, there was a *log10* operation inside a *i* loop, with *i* ranging from 1 to 512 (HBLKSIZE-1). Since the *log10* operation was performed based on *energy[i]*, the solution was creating a table with 512 positions, so that *energy[i]* determines the index that should be accessed from the table. To be linear access, *energy[i]* has to be multiplied by 1000 and converted to an integer. Since it is guaranteed that in the *else* condition the array value is always positive, a condition was added to upper limit the index of the log10 table (511 is the maximum index). With this, and noticing the multiplication by 10 (of log10) in the original code, the *tablog10_psycho_3_powerdensityspectrum* was created by calculating $LOG10(x/10000)$ in an excel sheet, with *x* ranging from 0 and 511. It was then copied and defined in *psycho_3.c*, with a change in index 0. Since *log10(0)=-inf*, the first index in the table was defined as -40 (the same value as index 1).

The second case was *psycho_3_init_add_db*, also in *psycho_3.c*. In this function, there were both *pow* and *log10* operations inside a *i* loop, with *i* ranging from 0 to 999 (DBTAB-1). However, this case was straightforward to solve, since the result produced in each iteration does not depend on any input data. Considering this, the *tablog10_psycho_3_init_add_db* was created by calculating the whole expression in an excel sheet, with *x* ranging from 0 to 99.9 ($x = i/10.0$).

The third case was *psycho_3_spl*, in *psycho_3.c*. In this function, there was a *log10* operation inside a *i* loop, with *i* ranging from 0 to 31 (SBLIMIT-1). Since the *log10* operation was performed based on *scale[i]* and its value was never higher than 20000 (after being multiplied by 32768), the solution was creating a table with 2000 positions, so that *scale[i]* determines the index that should be accessed from the table. To be linear access, *scale[i]* has to be multiplied by 3276.8 and converted to an integer. Since it is guaranteed that the array value is always positive, a condition was added to limit the index of the

log10 table (1999 is the maximum index). With this, and noticing the multiplication by 20 followed by the subtraction by 10 (of log10) in the original code, the *tablog10_psycho_3_spl* was created by calculating $20 \times log10(x \times 10) - 10$ in an excel sheet, with *x* ranging from 0 and 1999. It was then copied and defined in *psycho_3.c*.

The fourth case was *psycho_3_fft*, in *psycho_3.c* too. In this function, there was a *pow* operation and also a *cos* operation inside a *i* loop, with *i* ranging from 0 to 1023 (BLKSIZE-1). This case was straightforward to solve since the result produced in each iteration does not depend on any input data. Considering this, the *tabcos_psycho_3_fft* was created by calculating both expressions in an Excel sheet, with *x* ranging from 0 to 1023.

The fifth and last case was *create_dct_matrix*, in *subband.c*. In this function, there was a *cos* operation inside a nested loop, with *i* ranging from 0 to 15 and *k* ranging from 0 to 31. This case was also straightforward to solve since the result produced in each iteration does not depend on any input data. Considering this, the *tabcos_create_dct_matrix* was created by calculating the whole expression in an Excel sheet, with *aux* ranging from 0 to 511 (16×32).

The third and last change in *libtwolame* was related to memory. Initially, there was a memory allocation for *bit_stream* struct in *twolame_buffer_init*, called from *twolame_encode_buffer_interleaved* function. Since this function executes in every *while* loop iteration, in *firmware.c*, the memory allocation was also performed many times. Therefore, the *TWOLAME_MALLOC* of *bit_stream* struct was moved to the *main* function, meaning that it executes just once before the encoding process starts. In addition, the *TWOLAME_FREE* in *twolame_buffer_deinit*, called from *twolame_encode_buffer_interleaved*, was removed and, consequently, added to *main*. After this process, the system was emulated on the PC once again to check correctness.

With basic software optimizations already implemented, the next move was measuring the execution time of IOb-MP2-E in FPGA, for different input data. This requires different memory management since allocating memory during program execution on an FPGA can be less straightforward than on a traditional CPU or software-based system. In an FPGA, memory allocation typically needs to be done statically, rather than dynamically during program execution. FPGA-based systems, like IOb-MP2-E, are designed to have a fixed memory structure, and memory is allocated during the configuration of the FPGA (which is static once it is programmed) rather than at runtime. That said, there are some scenarios where dynamic memory allocation is possible on an FPGA, in case they provide memory blocks that can be accessed and reconfigured during runtime, but this is usually limited in capacity and not that flexible.

Considering this, a macro called *PC_EMUL_RUN* is defined to control each memory allocation, through an if directive. In the first case when the program is emulated on PC, *PC_EMUL_RUN* should

be defined. In a second case when the program runs in FPGA, *PC_EMUL_RUN* should not be defined. Focusing on the second case, the memory space available starts at *DATA_BASE_ADDR*. This macro defines the base address based on IOb-MP2-E macros from *config.mk*, the IOb-MP2-E configuration script. In *main()*, the first example is *recvfile_ch* pointer, which represents where the input data is stored in memory. As it is the first, it should be equal to *DATA_BASE_ADDR*. The second example is *pcmaudio* pointer, which represents where the input data buffer is stored. As it is the second, it should be equal to DATA_BASE_ADDR + recv_file_size (the size of the input file). The third example is *mp2buffer* pointer, which represents where the encoded data buffer is stored. It should be equal to $pcmaudio + AUDIObUFSIZE * sizeof(short)$ (the second pointer plus the size of the input data buffer). The fourth example is *mybs* pointer, which represents where the bit_stream struct is stored. It should be equal to $mp2buffer + MP2BUFSIZE * sizeof(unsignedchar)$ (the third pointer plus the size of the encoded data buffer). The fifth and last example is *outfile* pointer, which represents where the encoded MP2 data is stored. It should be equal to $mybs + sizeof(bit_stream)$ (the fourth pointer plus the size of the *bit_stream* struct). In addition, the *free()* functions were also inserted in a *#ifdef PC_EMUL_RUN* directive, since the function should only be called when the memory is dynamically allocated.

4.4 Profiling

The last implementation in the software architecture was **profiling** [49]. This process is a form of dynamic program analysis that can measure different variables, like memory space or time complexity. In this case, it was used to measure the duration of each function. The process of profiling *TwoLAME* consisted of three phases, with the first phase being a more high-level approach. This phase was simple since only the function calls directly made from *main()* were considered. The timer was first initialized through *timer_init(TIMER_BASE)* function. Then, a *elapsed_time* integer array of size 50 was declared and set with zero. In the profiling itself, some basic operations were used for each function call. Immediately before a function call, *timer_time_ms()* is invoked, returning the current time in ms which is then stored in *start_elapse_time* variable as an unsigned integer. Immediately after a function call, *timer_time_ms()* is invoked again, returning the current time in ms which is then stored in *end_elapse_time* variable as an unsigned integer as well. Then, the difference between both variables is calculated and stored in a certain index of *elapsed_time* array (one index for each function call). By doing this, 13 functions were included in the first phase of *TwoLAME* profiling. At the end of *main()*, every position of the array is printed, showing how many ms each function call took. The printing does not influence the profiling, as it is done after the last function measurement. Another interesting detail is that both *uart_recvfile* and *uart_sendfile* functions are excluded from the profiling, which is correct since they perform data transfers that are not part of the *TwoLAME*. Table 8 shows the first phase of the profiling for all input files, presented in *Results* section.

By analyzing *twolame_encode_buffer_interleaved*, it is noticeable that the relevant operation inside the function is *encode_frame()*, as all the other are basic. Considering this, the second phase of profiling includes all function calls inside *encode_frame*. This phase was similar to the previous one in terms of methodology. The only differences were the usage of the *elapsed_time_twolame* array, which was declared and set to 0 in *main()* and passed as an argument to *twolame_encode_buffer_interleaved* and *encode_frame()*, consecutively. The variables that store the time values are also different, being *start_elapse_time_twolame* and *end_elapse_time_twolame*, declared in *twolame.c* as global unsigned integers. By doing this, 23 functions were included in the second phase of *TwoLAME* profiling. It was not 23 functions but 23 blocks of code from *encode_frame()*, because apart from the *libtwolame* functions, there is additional code that has to be included for correct measurements, like if conditions and *for* loops. At the end of *main()*, every position of the array is printed, showing how many ms each part of *encode_frame()* took. Table 9 shows the second phase of the profiling for all input files, , presented in *Results* section.

Looking at *twolame_psycho_3*, there are many other *libtwolame* functions inside it. Considering this, the third phase of profiling includes all function calls inside *twolame_psycho_3*. This phase was also similar to the previous ones in terms of methodology. The only differences were the usage of the *elapsed_time_psycho_3* array, which was declared and set to 0 in *main()* and passed as argument to *twolame_encode_buffer_interleaved*, *encode_frame()* and *twolame_psycho_3*, consecutively. The variables that stored the time values were also different, being *start_elapse_time_psycho_3* and *end_elapse_time_psycho_3*, declared in *twolame.c* as global unsigned integers. By doing this, 13 blocks of code were included in the third phase of *TwoLAME* profiling, with most of them being just functions. At the end of *main()*, every position of the array is printed, showing how many ms each block of *twolame_psycho_3* took. Table 10 shows the third phase of the profiling for all input files, presented in *Results* section.

5 Results

This chapter addresses the experimental results for the IOb-MP2-E running on FPGA, with and without hardware accelerators. It starts by presenting the results of *TwoLAME* profiling. Then, it presents the implementation results in FPGA. Afterward, the execution times are analyzed. In the end, the results obtained are compared with the real-time encoding requirements.

5.1 Profiling

The results of the first stage of *TwoLAME*'s profiling are shown in the table below. The values below 1 ms are represented by a hyphen ('-').

	Input file			
	short.wav	long.wav	noise.wav	vivaldi.wav
twolame_init()	3	2	3	-
wave_init()	-	-	-	-
twolame_set_num_channels()	-	-	-	-
twolame_set_in_samplerate()	-	-	-	-
twolame_set_bitrate()	-	-	-	-
twolame_init_params()	11	11	11	11
wave_get_samples()	4	126	89	135
twolame_encode_buffer_interleaved()	1510	24345	19820	26010
memcpy()	2	59	19	32
twolame_encode_flush()	79	80	185	155
memcpy()	-	-	-	-
twolame_close()	-	-	-	1
***TwoLAME* total**	1614	24719	20210	26467

Table 8: Execution time for all input files (first phase of *profiling*) [ms].

The previous information shows that *twolame_encode_buffer_interleaved* occupies most of the execution time, which is expected as it is responsible for the encoding, calling many other *TwoLAME* functions. Nonetheless, this information is not enough, mainly because developing hardware to accelerate the whole function would be an extremely difficult task. Therefore, the results of the second stage of *TwoLAME*'s profiling are shown in the table below.

	Input file			
	short.wav	*long.wav*	*noise.wav*	*vivaldi.wav*
scale_and_mix_samples	-	9	5	3
twolame_buffer_sstell	3	139	52	78
twolame_available_bits	-	16	7	8
twolame_window_filter_subband	141	3487	2590	3662
twolame_scalefactor_calc	7	172	142	201
twolame_find_sf_max	-	13	3	6
twolame_scalefactor_calc	-	5	6	4
twolame_psycho_3	1377	19062	16222	20829
twolame_sf_transmission_pattern	-	17	8	10
twolame_main_bit_allocation	16	424	343	485
twolame_write_header	2	12	7	7
buffer_putbits	-	4	1	4
twolame_write_bit_alloc	-	11	11	17
twolame_write_scalefactors	1	22	20	27
twolame_subband_quantization	21	503	346	488
twolame_write_samples	12	306	149	210
buffer_put1bit	-	10	4	2
buffer_putbits	-	5	4	2
twolame_dab_crc_calc	-	9	3	3
buffer_put1bit	-	3	-	-
twolame_buffer_sstell	-	1	-	1
twolame_do_energy_levels	-	7	1	3
twolame_crc_writeheader	-	3	1	2
***TwoLAME* total**	1614	24719	20210	26467

Table 9: Execution time for all input files (second phase of *profiling*) [ms].

The previous table shows that the eight block of *encode_frame()* occupies most of the execution time, which includes two nested loops (executed depending on *if* conditions) and a switch case. The nested loops perform simple operations, so they are not relevant. The switch case calls a certain function depending on the switch condition. However, after inspecting *glopts-¿psymodel* condition it is perceptible that case 3 is always selected, calling *twolame_psycho_3*. Nevertheless, this information is still not enough and the results of the third stage of *TwoLAME*'s profiling are shown in the table below.

	Input file			
	short.wav	long.wav	noise.wav	vivaldi.wav
twolame_psycho_3_init	635	642	635	635
mem → fft_buf	5	133	115	155
mem → fft_buf	6	117	77	120
psycho_3_fft	47	1138	886	1253
psycho_3_powerdensityspectrum	40	1026	791	1115
psycho_3_spl	7	170	148	204
psycho_3_tonal_label	7	153	258	173
psycho_3_noise_label	8	220	167	235
psycho_3_dump	-	4	2	4
psycho_3_decimation	3	45	38	67
psycho_3_threshold	611	15168	12975	16684
psycho_3_minimummasking	-	23	14	29
psycho_3_smr	-	9	6	13
***TwoLAME* total**	1614	24719	20210	26467

Table 10: Execution time for all input files (third phase of *profiling*) [ms].

The previous table shows interesting information. First, it is noticeable that several functions have an insignificant execution time, compared to *TwoLAME* total execution time. Second, it is clear that *psycho_3 function 11*, which corresponds to *psycho_3_threshold* function, occupies the biggest part of the program execution. For *short.wav*, *long.wav*, *noise.wav* and *vivaldi.wav*, this function corresponds to 37%, 61%, 64% and 63% of *TwoLAME* execution time, respectively. This motivates the hardware acceleration of *psycho_3_threshold*, described in *Hardware architecture* section.

5.2 FPGA implementation

This work consists of accelerating the *psycho_3_threshold* function in IOb-MP2-E using hardware acceleration. This requires more hardware, which indeed affects the resource consumption in FPGA, compared to the implementation without hardware acceleration. All the work was tested using *KU040 Xilinx Kintex® UltraScale™ Development Kit* [50]. This chip is based on the 28nm *UltraScale+* architecture, containing *Xilinx XCKU040-1FBVA676 FPGA*, 250 MHz LVDS Oscillator (system clock), 1GB DDR4 SDRAM and USB-UART Interface. As internal resources, it allows a total of 242400 LUTs, 484800 FFs, 600 BRAMs, and 1920 DSPs. In this work, the Xilinx board runs at 100MHz, the frequency at which IOb-MP2-E is compiled.

Table 11 shows the implementation results of IOb-MP2-E without *Versat* and with *Versat*, first using *spectrum_search* accelerator, then using *masking_threshold* accelerator.

Metric	without *Versat*	with *Versat*	
		spectrum_search	*masking_threshold*
Total LUTs	24489	38913	34810
Logic LUTs	19818	33359	29267
LUTRAMs	4288	5160	5152
SRLs	383	394	391
Flip-Flops	24888	37653	35186
RAMB36	157	161	176
RAMB18	5	5	6
URAM	0	0	0
DSP Blocks	10	26	14

Table 11: FPGA implementation results of IOb-MP2-E with and without *Versat*.

The previous information offers valuable insights into the influence of *Versat* on various hardware utilization metrics, highlighting the potential advantages and trade-offs of its inclusion in the IOb-MP2-E. A brief description of each metric is presented below.

- **Total LUTs** represent the overall utilization of combinational logic elements within an FPGA. LUTs are fundamental components for implementing digital logic circuits.
- **Logic LUTs** is a subset of LUTs used specifically for implementing logic operations in an FPGA. These LUTs store and compute logical functions and decisions.
- **LUTRAMs** are a specialized type of memory in an FPGA that allows data storage and retrieval within a LUT structure. They combine the functions of both LUTs and small memory units.
- **Shift Register LUTs** (SRLs) are LUTs configured for implementing shift register operations, where data is shifted bit by bit within a sequence. They are often used for tasks involving data serialization and deserialization.
- **Flip-Flops** (FFs) are sequential storage elements within an FPGA. They are used to store and transfer data over time and are essential for implementing memory elements, registers, and clocked logic circuits.
- **RAMB36** represents dedicated memory blocks within the FPGA that can store 36,000 bits of data. These blocks are typically used for larger data storage and retrieval tasks.
- **RAMB18** represents dedicated memory blocks within the FPGA that can store 18,000 bits of data. These are suitable for tasks requiring smaller memory storage.
- **URAM** (UltraRAM) is a type of high-capacity memory resource within some FPGAs. It offers significantly larger memory storage compared to regular RAM blocks and is used for high-performance memory-intensive applications.

- **DSP Blocks** (Digital Signal Processor Blocks) are specialized resources in an FPGA designed for accelerating digital signal processing tasks. They include dedicated hardware for tasks like multiplication, accumulation, and complex arithmetic operations.

Focusing on the measurement without *Versat* and with *Versat* using *spectrum_search* accelerator, the results show a significant increase from 24489 Total LUTs without *Versat* to 38913 Total LUTs with *Versat*. This suggests that *Versat* consumes additional LUT resources within the FPGA, as expected. The inclusion of *Versat* increases from 19818 Logic LUTs to 33359, affirming its impact on logic operations within the system. It also exhibits a change in LUTRAMs from 4288 to 5160, indicating the utilization of dedicated memory resources within the FPGA. The small difference between the two scenarios, from 383 SRLs to 394 SRLs, suggests that *Versat* minimally affects this particular resource. On the opposite, the significant jump in Flip-Flops from 24888 without *Versat* to 37653 with it is notable, indicating a requirement increase for sequential storage elements within the FPGA. There is also a slight increase in RAMB36, and a big one in DSP Blocks, from 10 to 26. Interestingly, there is no difference in the number of RAMB18 and URAM resources, suggesting that *Versat* does not utilize this specific resource.

The measurement results with *Versat* using *masking_threshold* accelerator are similar to the *spectrum_search* accelerator ones. More precisely, the *masking_threshold* accelerator requires a bit less resources than the previous accelerator, except for RAMB36, RAMB18, and DSP Blocks, which are all required in higher quantity.

5.3 Execution time

Table 12 shows the execution time of IOb-MP2-E without *Versat* and with *Versat*, using *spectrum_search* and *masking_threshold* accelerators, for all input files.

		Input files			
		short.wav	*long.wav*	*noise.wav*	*vivaldi.wav*
without *Versat*	*psycho_3_threshold*	536	13365	11381	14694
	TwoLAME total	1509	22672	18520	24325
with ***spectrum_search***	*psycho_3_threshold*	20	350	289	395
	TwoLAME total	974	9391	7209	9729
with ***masking_threshold***	*psycho_3_threshold*	555	13807	11761	15168
	TwoLAME total	1518	22677	18637	24458

Table 12: Execution time of IOb-MP2-E with and without *Versat* for all input files [ms].

The previous information shows that the *spectrum_search* accelerator can induce a substantial reduction in execution time. Specifically, the *psycho_3_threshold* execution times for *short.wav*, *long.wav*, *noise.wav* and *vivaldi.wav* are reduced by approximately 96.26%, 94.6%, 95.34% and 94.98%, respectively. Considering that the *spectrum_search* accelerator executes part of the original function and not the whole function, these results are extremely positive. In this accelerator there is only a a main transference of data at the beginning, followed by the execution of many loop iterations, keeping the data inside the accelerator. Since only a few constants are changed in each iteration, the performance gain is big. Moreover, with a reduction of 94-96% for all input files, the *masking_threshold* accelerator becomes unnecessary. That is because *Versat* does not allow execution of different accelerators in the same run, but rather the execution of a single accelerator, with different configurations (the case of this work) or not. The *masking_threshold* accelerator increased the execution time by a few milliseconds for all input files, i.e. no acceleration was achieved with *masking_threshold* at all. This can be explained by the fact that the overhead of transferring data is greater than any gain in performance using the accelerator. The loop executes only 512 iterations, which is not enough to make the data transfer worthwhile.

5.4 Real-time requirements

After measuring the execution time of the original and the hardware-accelerated implementation in IObMP2-E, it is possible to calculate the speedup achieved for each input file, using the formula presented below.

$$\textbf{\textit{Speedup achieved}} = \frac{\textit{Execution time without Versat}}{\textit{Execution time with Versat}} \tag{1}$$

Additionally, the *Amdahl's Law* [51] can be applied to understand the limitations of the hardware acceleration, for the *psycho_3_threshold* function.

$$\textit{Speedup possible} = \frac{1}{(1-p)+\frac{p}{s}} \tag{2}$$

In the formula above, s is the speedup of the part of the task that benefits from improved system resources, i.e. is the speedup achieved on the parallelizable portion; p is the proportion of execution time that the part benefiting from improved resources originally occupied, i.e. is the proportion of the program that can be parallelized.

In the ideal case where the parallelizable part (p) disappears or, in other words, where the speedup (s) is very high, the value of s approaches infinity, and the formula simplifies accordingly.

$$\textbf{\textit{Speedup possible}} \approx \frac{1}{1-p} \quad (3)$$

A more interesting point is to compare the achieved speedup with the possible speedup based on the real-time requirements, for each input file. This involves a formula to estimate how long it would take to encode the audio file in real time, which is presented below.

$$\textbf{\textit{Real-time}} = \frac{\textit{number of frames} \times \textit{number of samples}}{\textit{sampling frequency}} \quad (4)$$

In this case, the real-time is calculated from the number of frames, the number of samples, and the sampling frequency. The *number of samples* refers to the total number of audio samples in the audio file, which are individual data points that represent the amplitude of the audio signal at a particular point in time [52]. This value is multiplied by the *number of frames* since each frame typically consists of a fixed number of audio samples (the case of this work) to process audio data efficiently. The result of the product is then divided by the *sampling frequency*, a fundamental parameter of digital audio that specifies how many samples are taken per second to represent the analog audio signal.

The *sampling frequency* is $44100Hz$ and the *number of samples* is 1152, for all input files. The remaining variable, *number of frames*, is different for each input file. In concrete, *short.wav*, *long.wav*, *noise.wav* and *vivaldi.wav* contain 11, 296, 114 and 162 frames, respectively. Table 13 shows the real-time for all input files.

	Input file			
	short.wav	*long.wav*	*noise.wav*	*vivaldi.wav*
Real time	287	7732	2978	4232

Table 13: Real time for all input files encoding [ms].

Based on the previous information, the speedup required can be determined by the following formula.

$$\textit{Speedup required} = \frac{\textit{Execution time without Versat}}{\textit{Real-time}} \quad (5)$$

Table 14 presents the outcomes of the previously discussed calculations.

	Input file			
Speedup	*short.wav*	*long.wav*	*noise.wav*	*vivaldi.wav*
achieved with *spectrum_search*	1.549	2.414	2.569	2.5
possible	1.551	2.436	2.594	2.526
required	5.251	2.9321	6.219	5.748

Table 14: Speedup achieved, possible, and required for all input files.

In more detail, the achieved speedup is calculated based on the *TwoLAME total* execution times with *spectrum_search* and without *Versat*, from table 12. The possible speedup is calculated taking into account the proportion of the *TwoLAME* that can be parallelized, which is obtained as the result of the division between *psycho_3_threshold* and *TwoLAME total* execution times without *Versat*, also from table 12. Lastly, the required speedup is calculated based on the *TwoLAME total* execution times without *Versat*, from table 12, and the Real-time, from table13.

Focusing on the results, it is noticeable that the *spectrum_search* accelerator provides a speedup very close to the possible one (the maximum speedup achievable by accelerating *psycho_3_threshold* function). However, the required speedup indicates that the acceleration of *psycho_3_threshold* is not enough to meet the real-time requirements. In particular, the first two input files encoding is approximately at 3.39 and 1.21 speedup from meeting the real-time requirements, respectively. In the first case, the input file has a small number of frames, which doesn't potentiate *Versat* acceleration and, consequently, the achieved speedup stays far from the required one. In the second case, the input file has a significant number of frames but the audio itself is simple, which implies a lower demand of *psycho_3_threshold* function (executed by *spectrum_search* accelerator). For this reason, the required speedup is lower and easier to reach, which makes the achieved speedup very close to the required one. The last two input files' encoding is approximately at 2.42 and 2.30 speedup from meeting the real-time requirements. These results give some guarantees about achieving the goal of this work since the third input file has a considerable number of frames and represents the worst-case scenario for real-time encoding (being a stereo white noise audio file). The fourth input file has a normal audio complexity since it is part of an existing soundtrack, and so the difference between achieved and required speedup is similar to the previous file.

Regarding the hardware architecture, the speedup values give a direct relation to the frequency necessary to meet the real-time requirements. Picking the *noise.wav* input file as an example, the required speed up for IOB-MP2-E without hardware acceleration is 6.2. Since IOb-MP2-E operates at 100MHz, this means that an implementation for 620MHz would meet the goal.

In a case where high clock speeds are a priority, application-specific integrated circuits (ASICs) are often more capable of running at faster frequencies. This integrated circuit can be optimized for

specific tasks and, in some cases, achieve clock speeds as high as 500MHz in advanced semiconductor technologies like 28nm. However, ASICs require a significant upfront investment, and the industry's low volume may make them less cost-effective. In contrast, FPGAs offer flexibility and programmability, allowing for quicker design iterations and so often preferred for low-volume or prototyping applications. Decoders may benefit from ASICs, but for general-purpose or adaptable solutions, CPUs, GPUs, and FPGAs provide versatile off-the-shelf options. Although they may operate at slightly lower clock speeds, this brings the advantages of reduced power consumption, lower heat generation, and potentially longer component lifespan. Considering this, the IOB-MP2-E has the potential to be a top-notch system in the future, as the achieved performance for the worst-case scenario shows that an implementation at 240MHz would meet the goal.

6 Conclusion

In this work, the first system RISC-V-based MP2 audio encoder has been developed to the best of our knowledge. The system has been implemented with the IOb-SoC system equipped with the *VEXRISC* CPU and the *Versat* reconfigurable accelerator.

Compared with the commercial *CWda74* IP Core, the support for floating-point arithmetic allows us to use high-precision and good-quality open-source software such as TwoLame, as we have not found any fixed-point implementations. The CWda74 uses third-party proprietary software and needs to pay royalties for its use. However, due to its fixed-point performance, it uses considerably lower resources and can work at a substantially lower frequency.

6.1 Achievements

The most notable results of this work are summarized below.

- A new IOb-SoC system called IOb-MP2-E has been developed to support the development of the embedded MP2 audio encoder. This system can be emulated on a PC or run on an FPGA.
- The *TwoLAME*'s library was successfully ported to the IOb-MP2-E with floating-point precision.
- Software optimizations were done to reduce unnecessary operations related to mathematical functions and memory allocation.
- Using the FPGA implementation, the software was profiled to determine which functions took most of the execution time. The *psycho_3_threshold* was marked as the function to be accelerated using *Versat*.
- Two hardware accelerators were developed, each corresponding to a different part of the function.
- The FPGA implementation results demonstrate that the addition of the *Versat* accelerator reduces the encoding time considerably, with results close to the theoretical minimum. The obtained speed was not enough to meet the real-time requirements at the implementation frequency of 100MHz. The achieved performance for the worst-case scenario shows that an implementation at 240MHz would meet the goal.
- The FPGA implementation results also demonstrate that the addition of the *Versat* accelerator has a considerable impact on the resources used in a factor of 10x. It shows that the benefits of using these accelerators do not cover the costs. It also suggests that a smaller but frequently reconfigured single accelerator is preferable.

6.2 Future work

The work carried out demonstrates that the main area for improvement is to reduce the resources necessary. Some improvements that have been identified are outlined below.

- Make the accelerator reconfigurable to perform multiple functions while reusing its hardware and implementing various accelerators.
- Explore using custom instructions created to extend the *VEXRISCV*'s ISA, attempting to accelerate *TwoLAME* algorithm horizontally.
- Once the real-time requirements are met, compare the resource consumption with the *CWda74*.

These findings contribute to the ongoing efforts to provide more efficient and competitive solutions for audio encoding in MPEG-1/2 Layer II format, with the potential for real-world applications in broadcasting and multimedia.

Bibliography

[1] I. Conexant Systems. *MPEG-2 Codec CX23415*, 2003. URL https://datasheet.ciiva.com/26931/getdatasheetpartid-486546-26931824.pdf. Accessed Dec 2022.

[2] CableWorld. *MPEG-2 Encoder CW-4888*, 2014. URL http://www.cableworld.hu/downloads/data_sheets/4888p-a.pdf. Accessed Dec 2022.

[3] I. Computer Modules. *Futura II ASI+IP™*, 2017. URL https://cdn-docs.av-iq.com/dataSheet/Futura%20II%E2%84%A2%20SDI%20HDSDI%20HDMI%20-ASI%2BIP.pdf. Accessed Dec 2022.

[4] A. Sengupta. Intellectual property cores: Protection designs for ce products. *IEEE Consumer Electronics Magazine*, 5(1):83–88, 2016. doi: 10.1109/MCE.2015.2484745. Accessed 2022.

[5] MPEG-1/2 - Layer i/ii Audio Encoder, Oct 2017. URL http://coreworks-sa.com/index.php?view=view_ip_core&part_number=CWda74&ipcore_id=57&category=Audio%20Encoders. Accessed Dec 2022.

[6] L. IPbloq. *MPEG-1/2 - LAYER I/II AUDIO ENCODER*, 2019. URL https://ipbloq.files.wordpress.com/2021/09/ipb-mpeg-se-product-brief.pdf. Accessed Dec 2022.

[7] IOBundle. IOBundle Website, 2023. URL https://www.iobundle.com/. Accessed Sep 2023.

[8] IObundle. IObundle/iob-soc: IOB SoC Repository, 2023. URL https://github.com/IObundle/iob-soc. Accessed Apr 2023.

[9] E. A. Waterman and K. Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, December 2019. Accessed 2023.

[10] njh. njh/twolame: TwoLAME is an optimised MPEG Audio Layer 2 (MP2) encoder, 2022. URL https://github.com/njh/twolame. Accessed 2023.

[11] I. O. for Standardization. Iso 22412:2020, 2022. URL https://www.iso.org/standard/22412.html. Accessed Dec 2022.

[12] N. Humfrey. TwoLAME - An Optimized MPEG Audio Layer 2 (MP2) Encoder, 2022. URL https://www.twolame.org/. Accessed Jul 2023.

[13] U. Farooq, Z. Marrakchi, and H. Mehrez. *FPGA Architectures: An Overview*, pages 7–48. Springer New York, New York, NY, 2012. ISBN 978-1-4614-3594-5. doi: 10.1007/978-1-4614-3594-5_2. URL https://doi.org/10.1007/978-1-4614-3594-5_2. Accessed Oct 2023.

[14] IObundle. IObundle/iob-versat: Versat Repository, 2023-03. URL https://github.com/IObundle/iob-versat. Accessed Aug 2023.

[15] Y. Wang and M. Vilermo. Modified discrete cosine transform: Its implications for audio coding and error concealment. *Journal of the Audio Engineering Society*, 51(1/2):52–61, 2003. Accessed 2023.

[16] M. Heires. The international organization for standardization (iso). *New Political Economy*, 13(3): 357–367, 2008. doi: 10.1080/13563460802302693. Accessed 2022.

[17] J. Buck. International electrotechnical commission. In *Handbook of Transnational Economic Governance Regimes*, pages 573–584. n.d. doi: 10.1163/ej.9789004163300.i-1081.494. Accessed 2022.

[18] IEEE. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017*, 2017. Accessed 2023.

[19] Z. Navabi. *VHDL: Analysis and modeling of digital systems*. McGraw-Hill, Inc., 1992. Accessed 2023.

[20] Coreworks SA, 2022. URL http://coreworks-sa.com/index.php?view=home. Accessed Dec 2022.

[21] ARM. *AMBA® AXI™ and ACE™ Protocol Specification*, 2011. URL http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf. Accessed Sep 2023.

[22] Intel Corporation. Intel Company Overview, 2022. URL https://www.intel.com/content/www/us/en/company-overview/company-overview.html. Accessed Sep 2023.

[23] L. IPbloq. IPbloq - About, 2022. URL https://ipbloq.com/about/. Accessed Dec 2022.

[24] TechOnline. Conexant systems, inc., 2022. URL https://www.techonline.com/directory/conexant-systems/. Accessed Dec 2022.

[25] M. Mitescu and I. Susnea. Using the asynchronous serial interface. *Microcontrollers in Practice*, pages 27–48, 2005. Accessed 2022.

[26] A. Shiranzaei and R. Z. Khan. Internet protocol versions — a review. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 397–401, 2015. Accessed 2022.

[27] LAME. About lame, 2022. URL https://lame.sourceforge.io/about.php. Accessed Dec 2022.

[28] I. Free Software Foundation. Gnu lesser general public license. Jun 2007. Accessed Dec 2022.

[29] P. Vankeerberghen, B. Van den Bogaert, and D. Massart. Dynamic link libraries i. introduction. *TrAC Trends in Analytical Chemistry*, 15(6):206–208, 1996. Accessed 2022.

[30] Mike Cheng. tooLAME: Layer2 Encoder, 2007. URL https://web.archive.org/web/20070321155552/http://www.eftel.com/~mikecheng/planckenergy/. Accessed Dec 2022.

[31] TwoLAME. twolame.h file reference, 2022. URL https://www.twolame.org/doc/twolame_8h.html. Accessed Mar 2023.

[32] N. Einspruch. *Application specific integrated circuit (ASIC) technology*, volume 23. Academic Press, 2012. Accessed 2022.

[33] P. Haugen, I. Myers, B. Sadler, and J. Whidden. A basic overview of commonly encountered types of random access memory (ram). n.d. Accessed 2023.

[34] Cadence Design Systems. Double Data Rate (DDR), 2022. URL https://www.cadence.com/en_US/home/explore/double-data-rate-ddr.html. Accessed Sep 2023.

[35] IObundle. IOB SoC GitHub Repository. https://github.com/IObundle/iob-soc. Accessed December 2022.

[36] N. Rathi, A. Kumar, N. Gupta, and S. K. Singh. A review of low-power static random access memory (sram) designs. In *2023 IEEE Devices for Integrated Circuit (DevIC)*, pages 455–459, 2023. doi: 10.1109/DevIC57758.2023.10134887. Accessed 2023.

[37] J. D. Lopes, M. P. Véstias, R. P. Duarte, H. C. Neto, and J. T. de Sousa. Coarse-grained reconfigurable computing with the versat architecture. *Electronics*, 10(6), 2021. ISSN 2079-9292. doi: 10.3390/electronics10060669. URL https://www.mdpi.com/2079-9292/10/6/669. Acessed Oct 2023.

[38] J. Liang, R. Tessier, and O. Mencer. Floating point unit generation and evaluation for fpgas. In *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.*, pages 185–194, 2003. doi: 10.1109/FPGA.2003.1227254. Acessed 2023.

[39] Xilinx. *AXI Reference Guide*, March 2011. URL https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf. Acessed Aug 2023.

[40] IObundle. IObundle/iob-timer: Timer and Clock Generator, August 2022. URL https://github.com/IObundle/iob-timer. Acessed Apr 2023.

[41] A. de Gennaro. Design of control and datapath of scenario-based hardware systems. 2016. Acessed 2023.

[42] GNU Project. Linux and the GNU System, 2023. URL https://www.gnu.org/gnu/linux-and-gnu.en.html. Acessed Sep 2023.

[43] M. G. . contributors. Audacity FAQ, 2023. URL https://www.audacityteam.org/FAQ/. Acessed 2023.

[44] M. T. Raymond L. Knapp, William V. POrter. Antonio Vivaldi, 2023. URL https://www.britannica.com/biography/Antonio-Vivaldi. Acessed Sep 2023.

[45] E. A. Waterman and K. Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. RISC-V Foundation, May 2017. Acessed 2023.

[46] cppreference.com. modf, modff, modfl, 2023. URL https://en.cppreference.com/w/c/numeric/math/modf. Acessed March 2023.

[47] TutorialsPoint. C library - <math.h>, 2023. URL https://www.tutorialspoint.com/c_standard_library/math_h.htm. Acessed Mar 2023.

[48] J. Detrey and F. de Dinechin. Floating-point trigonometric functions for fpgas. In *2007 International Conference on Field Programmable Logic and Applications*, pages 29–34, 2007. doi: 10.1109/FPL.2007.4380621. Acessed 2023.

[49] R. Patel. A survey of embedded software profiling methodologies. *International Journal of Embedded Systems and Applications*, 1(2):19–40, dec 2011. doi: 10.5121/ijesa.2011.1203. URL https://doi.org/10.5121%2Fijesa.2011.1203. Acessed May 2023.

[50] L. FMALL Co. XCKU040-1FBVA676I FPGA Product, 2023. URL https://www.fmall.uk/product/xilinx/xcku040-1fbva676i. Acessed Oct 2023.

[51] G. M. Amdahl. Computer architecture and amdahl's law. *Computer*, 46(12):38–46, dec 2013. ISSN 1558-0814. doi: 10.1109/MC.2013.418. Acessed 2023.

[52] TwoLAME. The libtwolame api. URL https://www.twolame.org/doc/api.html. Acessed Mar 2023.

Appendix A - *versatSpec.txt*

```
module av(const1,const2,const3,bark,Xtm){
    FloatMul mul1;
    FloatAdd add1;
    FloatAdd add2;
    FloatAdd add3;
#
    const2 → mul1:0;
    bark → mul1:1;

    mul1 → add1:0;
    const1 → add1:1;

    add1 → add2:0;
    Xtm → add2:1;

    add2 → add3:0;
    const3 → add3:1;

    add3 → out;
}


module dzRange(dz){
    FloatGreaterEqual ge1_dzRange;
    FloatLess lt1_dzRange;
    Const const1_dzRange;
    Const const2_dzRange;
#
    ////dz[j] >= −3.0
    dz → ge1_dzRange:0;
    const1_dzRange → ge1_dzRange:1;

    ////dz[j] < 8.0
    dz → lt1_dzRange:0;
    const2_dzRange → lt1_dzRange:1;

    ////if (dz[j] >= −3.0 & dz[j] < 8.0)
     and = ge1_dzRange & lt1_dzRange;
    and → out:0;
}


module Logic(dz){
    FloatLess lt1_Logic;
    FloatLess lt2_Logic;
    FloatLess lt3_Logic;
    Const const1_Logic;
    Const const2_Logic;
    Const const3_Logic;
    Const const4_Logic;
#
    ////dz[j] < −1
```

```
    dz → lt1_Logic:0;
    const1_Logic → lt1_Logic:1;

    ////dz[j] < 0
    dz → lt2_Logic:0;
    const2_Logic → lt2_Logic:1;

    ////dz[j] < 1
    dz → lt3_Logic:0;
    const3_Logic → lt3_Logic:1;

    ////sel(1)
    cond1_not = ~lt1_Logic;
    cond2_not = ~lt2_Logic;
    and1 = cond1_not & cond2_not;
    one = and1 ≪ const4_Logic;

    ////sel(0)
    cond3_not = ~lt3_Logic;
    and2 = cond2_not & cond3_not;
    or1 = lt2_Logic | and2;
    and3 = cond1_not & or1;
    zero = and3;

    ////sel
    sel = zero ∧ one;
    sel → out:0;
}


module vf4(dz,logic1,Xtm){
    FloatAdd add1_vf4;
    FloatAdd add2_vf4;
    FloatSub sub1_vf4;
    FloatSub sub2_vf4;
    FloatSub sub3_vf4;
    FloatSub sub4_vf4;
    FloatMul mul1_vf4;
    FloatMul mul2_vf4;
    FloatMul mul3_vf4;
    FloatMul mul4_vf4;
    FloatMul mul5_vf4;
    FloatMul mul6_vf4;
    Const const1_vf4;
    Const const2_vf4;
    Const const3_vf4;
    Const const4_vf4;
    Const const5_vf4;
    Const const6_vf4;
    Mux4 mux4_vf4;
    FloatNot not1_vf4;
#
    ////(0.4 * Xtm[k] + 6)
    const3_vf4 → mul1_vf4:0;
    Xtm → mul1_vf4:1;
    mul1_vf4 → add1_vf4:0;
```

```
    const4_vf4 → add1_vf4:1;

    ////vf = 17 * (dz[j] + 1) − (0.4 * Xtm[k] + 6)
    dz → add2_vf4:0;
    const2_vf4 → add2_vf4:1;
    const1_vf4 → mul2_vf4:0;
    add2_vf4 → mul2_vf4:1;
    mul2_vf4 → sub1_vf4:0;
    add1_vf4 → sub1_vf4:1;
    sub1_vf4 → mux4_vf4:0;

    ////vf = (0.4 * Xtm[k] + 6) * dz[j]
    add1_vf4 → mul3_vf4:0;
    dz → mul3_vf4:1;
    mul3_vf4 → mux4_vf4:3;

    ////vf = (−17 * dz[j])
    const5_vf4 → mul4_vf4:0;
    dz → mul4_vf4:1;
    mul4_vf4 → mux4_vf4:2;

    ////vf = −(dz[j] − 1) * (17 − 0.15 * Xtm[k]) − 17;
    dz → sub2_vf4:0;
    const2_vf4 → sub2_vf4:1;
    sub2_vf4 → not1_vf4:0;
    const6_vf4 → mul5_vf4:0;
    Xtm → mul5_vf4:1;
    const1_vf4 → sub3_vf4:0;
    mul5_vf4 → sub3_vf4:1;
    not1_vf4 → mul6_vf4:0;
    sub3_vf4 → mul6_vf4:1;
    mul6_vf4 → sub4_vf4:0;
    const1_vf4 → sub4_vf4:1;
    sub4_vf4 → mux4_vf4:1;

    ////sel
    logic1 → mux4_vf4:4;

    ////vf
    mux4_vf4 → out:0;
}


module psycho_3_add_db(a,b,add1,add2,sel,if){
    Mux4 mux4_psycho_3;
    Conditional1 conditional_psycho_3;
#

    ////return mux4
    a → mux4_psycho_3:0;
    b → mux4_psycho_3:3;
    add1 → mux4_psycho_3:2;
     add2 → mux4_psycho_3:1;
    sel → mux4_psycho_3:4;

    ////LTtm[j] = psycho_3_add_db()
```

```
    if → conditional_psycho_3:0;
    mux4_psycho_3 → conditional_psycho_3:1;
    a → conditional_psycho_3:2;
    conditional_psycho_3 → out:0;
}


module start() {
    DBMIN;
    Const const1_av;
    Const const2_av;
    Const const3_av;
    Const barkK;
    Const Xtm;
    Const Xnm;
    Mux2 mux2;
    av av1;
    Mem freq_subset;
    LookupTable bark;
    dzRange dzRange1;
    Logic logic1;
    vf4 vf1;
    FloatSub sub;
    FloatAdd add;
    psycho_3_add_db psycho_3_add_db1;
    Mem LTtmR;
    LookupTable mem;
    Const const1_psycho_3;
    Const const2_psycho_3;
    Const const3_psycho_3;
    Const const4_psycho_3;
    Const const5_psycho_3;
    FloatSub sub1_psycho_3;
    FloatAdd add1_psycho_3;
    FloatAdd add2_psycho_3;
    FloatMul mul_psycho_3;
    FloatGreater gt_psycho_3;
    FloatLess lt_psycho_3;
    FloatGreaterEqual ge_psycho_3;
    Float2Int f2i_psycho_3;
#
    ////FLOAT av_tone = −1.525 − 0.275 * bark[k] − 4.5 + Xtm[k]
    ////FLOAT av_noise = −1.525 − 0.175 * bark[k] − 0.5 + Xnm[k];
    const1_av → av1:0;
    const2_av → av1:1;
    const3_av → av1:2;
    barkK → av1:3;
    Xtm → av1:4;

    ////dz[j] = bark[freq_subset[j]] − bark[k]
    freq_subset → bark;
    bark → sub:0;
    barkK → sub:1;

    ////dz
    sub → dzRange1:0;
```

```
////dz
sub → logic1:0;

////dz
sub → vf1:0;
logic1 → vf1:1;
Xtm → vf1:2;

av1 → add:0;
vf1 → add:1;

//////////////////////////////////////psycho_3_add_db operations

////DBMIN or LTtmR
DBMIN → mux2:0;
LTtmR:1 → mux2:1;

////fdiff = (10.0 * (a − b));
mux2[3] → sub1_psycho_3:0;
add → sub1_psycho_3:1;
const1_psycho_3 → mul_psycho_3:0;
sub1_psycho_3 → mul_psycho_3:1;
fdiff = mul_psycho_3;

////fdiff > 990.0
fdiff → gt_psycho_3:0;
const2_psycho_3 → gt_psycho_3:1;

////fdiff < −990.0
fdiff → lt_psycho_3:0;
const3_psycho_3 → lt_psycho_3:1;

////idiff = (int) fdiff
fdiff → f2i_psycho_3:0;
idiff = f2i_psycho_3;

////idiff >= 0
 → ge_psycho_3:0;
const4_psycho_3 → ge_psycho_3:1;

////a + mem→dbtable[idiff]
mux2[3] → add1_psycho_3:0;
idiff → mem;
mem → add1_psycho_3:1;

////b + mem→dbtable[−idiff]
add → add2_psycho_3:0;
_idiff = −idiff;
_idiff → mem:1;
mem:1 → add2_psycho_3:1;

////sel(1)
cond1_not = ~gt_psycho_3;
cond2_not = ~lt_psycho_3;
and1 = cond1_not & cond2_not;
```

```
    one = and1 ≪ const5_psycho_3;

    ////sel(0)
    cond3_not = ∼ge_psycho_3;
    and2 = cond2_not & cond3_not;
    or1 = lt_psycho_3 | and2;
    and3 = cond1_not & or1;
    zero = and3;

    ////sel mux4
    sel = zero ∧ one;

    ////////////////////////////////////psycho_3_add_db inputs and output

    mux2[3] → psycho_3_add_db1:0;
    add → psycho_3_add_db1:1;
    add1_psycho_3 → psycho_3_add_db1:2;
    add2_psycho_3 → psycho_3_add_db1:3;
    sel → psycho_3_add_db1:4;
    dzRange1 → psycho_3_add_db1:5;
    psycho_3_add_db1 → LTtmR:0;
}

////////////////////////////////////////////////////////////////////////////


module psycho_3_add_db_V1(a,b,add1,add2,sel){
    Mux4 mux4_psycho_3;
#
    ////return mux4
    a → mux4_psycho_3:0;
    b → mux4_psycho_3:3;
    add1 → mux4_psycho_3:2;
    add2 → mux4_psycho_3:1;
    sel → mux4_psycho_3:4;
    mux4_psycho_3 → out:0;
}


module start1() {
    psycho_3_add_db_V1 psycho_3_add_db1;
    psycho_3_add_db_V1 psycho_3_add_db2;
    Const const1_psycho_3;
    Const const2_psycho_3;
    Const const3_psycho_3;
    Const const4_psycho_3;
    Const const5_psycho_3;
    Mem LTtmR;
    Mem LTnmR;
    LookupTable mem;
    LookupTable mem1;
    Const bit_rate;
    Const const1;
    Const const2;
    FloatSub sub1_psycho_3;
    FloatSub sub1_psycho_31;
```

```
      FloatSub sub1_psycho_32;
      FloatMul mul_psycho_3;
      FloatMul mul_psycho_31;
      FloatGreater gt_psycho_3;
      FloatGreater gt_psycho_31;
      FloatLess lt_psycho_3;
      FloatLess lt_psycho_31;
      FloatLess lt_psycho_32;
      Float2Int f2i_psycho_3;
      Float2Int f2i_psycho_31;
      FloatGreaterEqual ge_psycho_3;
      FloatGreaterEqual ge_psycho_31;
      FloatAdd add1_psycho_3;
      FloatAdd add1_psycho_31;
      FloatAdd add2_psycho_3;
      FloatAdd add2_psycho_31;
      Conditional1 conditional_psycho_3;
      Mem freq_subset;
      LookupTable ath;
      Mem LTg;
#
      /////////////////////////////////////psycho_3_add_db operations
      ///// mem,a,b

      ////fdiff = (10.0 * (a − b));
      LTnmR → sub1_psycho_3:0;
      LTtmR → sub1_psycho_3:1;
      const1_psycho_3 → mul_psycho_3:0;
      sub1_psycho_3 → mul_psycho_3:1;
      fdiff = mul_psycho_3;

      ////fdiff > 990.0
      fdiff → gt_psycho_3:0;
      const2_psycho_3 → gt_psycho_3:1;

      ////fdiff < −990.0
      fdiff → lt_psycho_3:0;
      const3_psycho_3 → lt_psycho_3:1;

      ////idiff = (int) fdiff
      fdiff → f2i_psycho_3:0;
      idiff = f2i_psycho_3;

      ////idiff >= 0
      idiff → ge_psycho_3:0;
      const4_psycho_3 → ge_psycho_3:1;

      ////a + mem→dbtable[idiff]
      LTnmR → add1_psycho_3:0;
      idiff → mem;
      mem → add1_psycho_3:1;

      ////b + mem→dbtable[−idiff]
      LTtmR → add2_psycho_3:0;
      _idiff = −idiff;
      _idiff → mem:1;
```

```
mem:1 → add2_psycho_3:1;

////sel(1)
cond1_not = ∼gt_psycho_3;
cond2_not = ∼lt_psycho_3;
and1 = cond1_not & cond2_not;
one = and1 ≪ const5_psycho_3;

////sel(0)
cond3_not = ∼ge_psycho_3;
and2 = cond2_not & cond3_not;
or1 = lt_psycho_3 | and2;
and3 = cond1_not & or1;
zero = and3;

////sel mux4
sel = zero ∧ one;

//////////////////////////////////////psycho_3_add_db inputs and output

LTnmR → psycho_3_add_db1:0;
LTtmR → psycho_3_add_db1:1;
add1_psycho_3 → psycho_3_add_db1:2;
add2_psycho_3 → psycho_3_add_db1:3;
sel → psycho_3_add_db1:4;

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////



bit_rate → lt_psycho_31:0;
const1 → lt_psycho_31:1;

freq_subset → ath;
ath → sub1_psycho_31:0;
const2 → sub1_psycho_31:1;

lt_psycho_31 → conditional_psycho_3:0;
ath → conditional_psycho_3:1;
sub1_psycho_31 → conditional_psycho_3:2;

//////////////////////////////////////psycho_3_add_db operations
///// mem,a,b

////fdiff = (10.0 * (a − b));
conditional_psycho_3 → sub1_psycho_32:0;
psycho_3_add_db1 → sub1_psycho_32:1;
const1_psycho_3 → mul_psycho_31:0;
sub1_psycho_32 → mul_psycho_31:1;
fdiff1 = mul_psycho_31;

////fdiff > 990.0
fdiff1 → gt_psycho_31:0;
const2_psycho_3 → gt_psycho_31:1;
```

```
////fdiff < −990.0
fdiff1 → lt_psycho_32:0;
const3_psycho_3 → lt_psycho_32:1;

////idiff = (int) fdiff
fdiff1 → f2i_psycho_31:0;
idiff1 = f2i_psycho_31;

////idiff >= 0
idiff1 → ge_psycho_31:0;
const4_psycho_3 → ge_psycho_31:1;

////a + mem→dbtable[idiff]
conditional_psycho_3 → add1_psycho_31:0;
idiff1 → mem1;
mem1 → add1_psycho_31:1;

////b + mem→dbtable[−idiff]
psycho_3_add_db1 → add2_psycho_31:0;
_idiff1 = −idiff1;
_idiff1 → mem1:1;
mem1:1 → add2_psycho_31:1;

////sel(1)
cond1_not1 = ∼gt_psycho_31;
cond2_not1 = ∼lt_psycho_32;
and11 = cond1_not1 & cond2_not1;
one1 = and11 ≪ const5_psycho_3;

////sel(0)
cond3_not1 = ∼ge_psycho_31;
and21 = cond2_not1 & cond3_not1;
or11 = lt_psycho_32 | and21;
and31 = cond1_not1 & or11;
zero1 = and31;

////sel mux4
sel1 = zero1 ∧ one1;

//////////////////////////////////////psycho_3_add_db inputs and output

conditional_psycho_3 → psycho_3_add_db2:0;
psycho_3_add_db1 → psycho_3_add_db2:1;
add1_psycho_31 → psycho_3_add_db2:2;
add2_psycho_31 → psycho_3_add_db2:3;
sel1 → psycho_3_add_db2:4;
psycho_3_add_db2 → LTg:0;

}
```