RISC-V-based MPEG1/2 Layer II Encoder

Tiago Alves da Silva

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Prof. José João Henriques Teixeira de Sousa

October 2023

Abstract

Resumo

Contents

List of Tables

List of Figures

1 Introduction

2 Background

2.1 MPEG-1/2 Layers I/II

The Moving Picture Experts Group (MPEG) is a working group that sets standards for media coding, established by International Organization for Standardization (ISO) [1] and International Electrotechnical Commission (IEC) [2]. 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 [3] (explained in the next section), 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 human 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 which 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 up 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 a MPEG-1 bitstream, meaning that a MPEG-2 BC stream can be read and interpreted by a 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.

2.2 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 in a block of logic or data that is used in a semiconductor chip when making a field-programmable gate array (FPGA) [4] or application-specific integrated circuit (ASIC) [5]. 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 [6] or VHSIC Hardware Description Language (VHDL) [7], 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's 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.

2.2.1 CWda74 MPEG-1/2 – Layer I/II Audio Encoder

The *CWda74* [8] is an audio IP core capable of encoding one audio stream in real-time, provided by *Coreworks, S.A.* [9].

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) 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 [10] FPGA), Avalon (for Altera [11] 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: Table of *CWda74* features.

2.2.2 IPB-MPEG-SE MPEG-1/2 – Layer I/II Audio Encoder

The *IPB-MPEG-SE* [12] is an audio IP core capable of encoding up to two stereo audio streams in real-time, provided by *IPbloq* [13].

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. Table 2 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 @150 MHz for two audio streams

Table 2: Table of *IPB-MPEG-SE* features.

2.3 MPEG-1/2 Layers I/II Chips

2.3.1 CX23415 MPEG-2 Codec

The *CX23415* [14] 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* [15].

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.

2.3.2 Futura II ASI+IP™

The *Futura II* [16] 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 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).

2.4 MPEG-1/2 Layers I/II Systems

2.4.1 MPEG-2 Encoder CW-4888

The *CW-4888* [17] is a 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) [18] or Internet Protocol (IP) [19] 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 [3]. 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.

2.5 MPEG-1/2 Layers I/II Software

2.5.1 TooLAME/LAME

The *LAME* [20] is a high-quality MPEG Layer III audio encoder, licensed under the Lesser General Public License (LGPL) [21] 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) [22] (on Windows).

Based on portions of *LAME* and ISO dist10 code, the *TooLAME* [23] 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 existing many MP2 encoders.

2.5.2 TwoLAME

Despite being unmaintained since 2003, the *TooLAME* software was directly succeeded by the *TwoLAME* [24] 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 3 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 3: Table of *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, which includes the *main()* function, and two other files, *audio_wave.c* and *audio_wave.h*. Figure 1 shows the pseudo code for the first part of *simplefrontend.c*, mainly consisting of initialization.

```
Beginning of main function

    Allocate memory for the PCM audio data
    Allocate memory for the encoded MP2 audio data
    Initialize TwoLAME encoder and set default parameters
    Open the WAV input file
    Set the number of channels in the input stream
    Set the MPEG audio mode for the output stream
    Set the sample rate of the PCM audio input
    Set the sample rate of the MPEG audio output
    Set the bitrate of the MPEG audio output stream
    Validate parameters and allocate internal buffers
    Open the MP2 output file
```

Figure 1: 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. 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 2 shows the pseudo code for the second part of *simplefrontend.c*, mainly consisting of the encoding process.

```
Beginning of loop

    Read audio data samples from the WAV input file
    If there is any sample to read:
        Encode PCM audio data to MP2 using TwoLAME
        Write the MPEG bitstream to the MP2 output file
    If not:
        Exit from loop

End of loop
```

Figure 2: 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. 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 3 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 3: 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. 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.

2.6 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.

2.7 IOb-SoC

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

Figure 4 shows the IOb-SoC high-level block diagram.

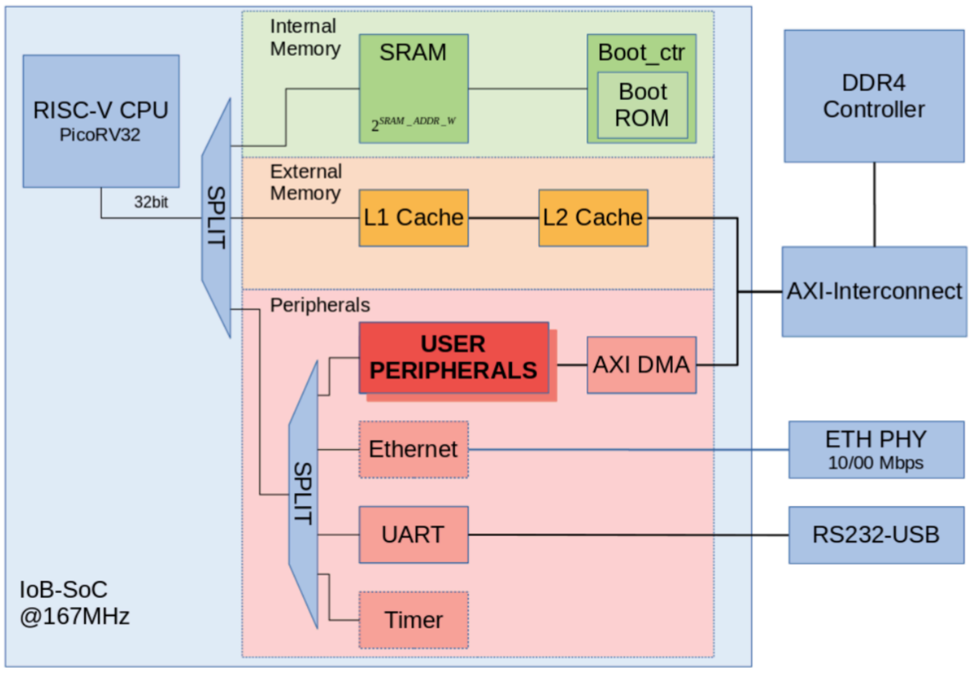


Figure 4: IOb-SoC high-level block diagram.

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** 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 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 in 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. In this work, the IOb-SoC system comprises seven main components/submodules: **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.

2.7.2 IOb-SoC Deliverables and FPGA 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 4 and 5 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 4: 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 5: Implementation resources for Intel Cyclone V Devices.

2.7.3 IOb-SoC Repository

The IOb-SoC Repository is a *GitHub* repository that allows the user to modify the system components easily. It contains specific directories and segments (Tool Command Language (TCL), Verilog, Makefile, etc), with the list below describing the main items.

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 ISO/SEC 11172-3: 1993 (E)

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.

2.8.1 Audio encoder

The 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 5 illustrates the basic structure of an audio encoder.

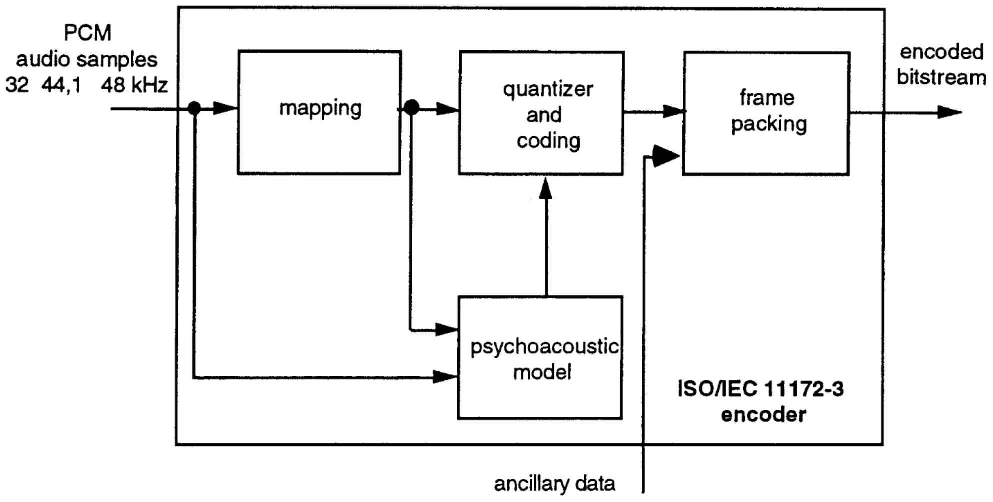


Figure 5: Audio encoder basic structure.

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).

2.8.2 Psychoacoustic encoder

The ISO/IEC 11172-3 (MPEG-Audio) psychoacoustic algorithm contains four primary parts, as shown in figure 6: *Filter Bank*, *Psychoacoustic Model*, *Bit or Noise Allocation* and *Bitstream Formatting*.

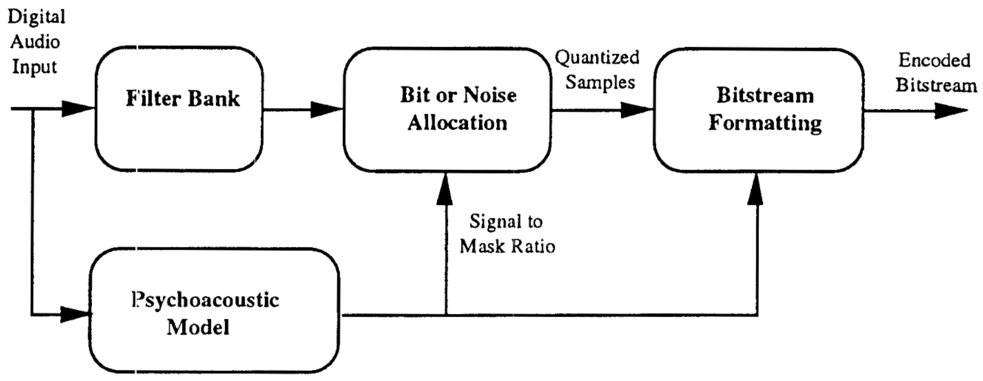


Figure 6: ISO/IEC 11172-3 encoder block diagram.

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/MDCT 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 formats all that information in an efficient way.
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 7 shows a more detailed Layer II Encoder flow chart.

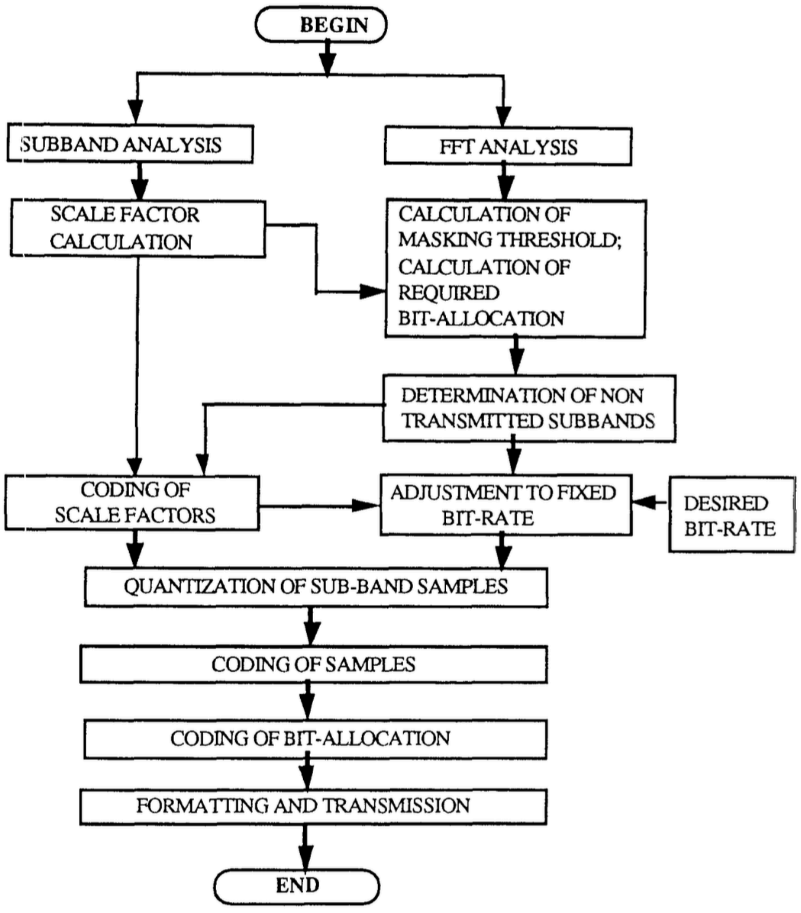

BEGIN
SUBBAND ANALYSIS
FFT ANALYSIS
SCALE FACTOR CALCULATION
CALCULATION OF MASKING THRESHOLD; CALCULATION OF REQUIRED BIT-ALLOCATION
DETERMINATION OF NON TRANSMITTED SUBBANDS
CODING OF SCALE FACTORS
ADJUSTMENT TO FIXED BIT-RATE
DESIRED BIT-RATE
QUANTIZATION OF SUB-BAND SAMPLES
CODING OF SAMPLES
CODING OF BIT-ALLOCATION
FORMATTING AND TRANSMISSION
END

Figure 7: Layer II encoder flow chart.

2.9 *Versat*

Versat constitutes a reconfigurable hardware accelerator tailored for integration within embedded systems. Operating on a Coarse Grained Reconfigurable Array (CGRA) architecture, *Versat* exhibits compact dimensions and low power consumption. The design features a simplified controller unit, facilitating self and partial reconfiguration.

The architecture is characterized by a modest number of functional units interlinked in a comprehensive graph topology to ensure optimal adaptability. The reduced functional unit count is offset by the ease of configuration and dynamic reconfiguration during runtime. Distinguishing itself from conventional CGRAs, *Versat* is proficient in mapping sequences of nested program loops rather than individual loops, with self and partial reconfiguration occurring between these nested loops.

Programming *Versat* is achievable through assembly language and a specialized C++ dialect. The ability to program using assembly language presents a novel and valuable aspect for program optimization and circumventing compiler or architectural challenges. Regarding practical utilization, *Versat* can serve as an accelerator for host processors on the same chip, enabling certain procedures to run faster and consume less energy when executed on *Versat*. To diminish dependency on the host Central Processing Unit (CPU), *Versat* incorporates a straightforward Controller responsible for algorithmic control, self-reconfiguration, and data movement. This frees up the host for more productive tasks. The Controller is programmable and encompasses an instruction memory storing *Versat* programs. Control over the various modules is administered through a Control Bus.

Incorporated within *Versat* is a dedicated Data Engine (DE) unit that conducts data computation. The DE unit comprises Functional Units (FUs) interconnected in a complete mesh, offering multiple configurations for a given datapath. This multiplicity aids in simplifying the compiler. The Configuration Module (CM) retains the DE's configuration, specifying the present datapath, and accommodates a register for preparing the subsequent configuration of the DE. Additionally, it possesses another register capable of temporarily storing numerous complete configurations, switchable during runtime. The Controller can write partial configurations to the CM and manage saving and restoring entire configurations from the configuration memory.

Furthermore, *Versat* integrates a Direct Memory Access (DMA) module, enabling autonomous and efficient data, program, and configuration transfers in and out of the device. The DMA operates through a master Advanced Extensible Interface – AXI4, an interface stemming from the Advanced Microcontroller Bus Architecture (AMBA) and designed by ARM.

Facilitating communication with the host processor, *Versat* is equipped with a shared Control Register File (CRF). The CRF incorporates two host interfaces, selectable during compilation: a Serial Peripheral Interface (SPI) and a parallel bus interface. The SPI slave interface is employed when the host system is an off-chip master device, primarily for debug and testing purposes. On the other hand, the parallel bus interface is utilized when the host is an embedded processor and may adhere to the AXI4 Lite format.

3 Software Architecture

This chapter addresses the software architecture. It starts by setting up the IOb-Soc and configuring the firmware to allow execution of *TwoLAME* encoding algorithm. Then, software optimizations are done in order to run *TwoLAME* in FPGA. At the end, a detailed profiling analysis defines which part of the *TwoLAME* software requires hardware acceleration.

Knowing beforehand that *TwoLAME* uses floating-point arithmetic (FP) [25], porting the software to PICORV32 CPU would not be the best option, due to its fixed-point arithmetic limitation. Therefore, the ideal option is porting *TwoLAME* to **VEXRISCV** (RV32IM) CPU, since it includes a floating-point unit (FPU) [26]. In this process, the first step was removing the *PICORV32* from the IOb-SoC and adding the VEXRISCV as the only CPU. This was straightforward, as *VEXRISCV* was previously implemented and tested by *IObundle, Lda*. Apart from adding the submodule to IOb-SoC'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*.

3.1 *Audacity*

Before porting the software, the *TwoLAME* itself had to be verified. This verification was just in a practical way since this open-source software was already tested and approved by other users in various systems, with positive feedback. Considering this, the verification consisted of listening to an original audio file and the corresponding encoded file, produced by *TwoLAME*.

To do so, four audio files were initially generated with *Audacity* software. This is a free open-source audio software, compatible with GNU/Linux (current working environment) and capable of multi-track audio editing and recording. In fact, there are some reasons for having more than one testing file, preferably with different characteristics. One of them is the number of frames, which varies depending on several factors, including the audio codec being used, the settings or parameters selected for encoding, and the duration of the audio being processed (the easiest to notice). Another reason is the ability to check if the encoding software behaves the same way for different specifications, which is a crucial point as the main goal of the intended IP core is to allow real-time encoding.

Thus, 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 Left and the other being distributed 100% to Right (*panning effect*).

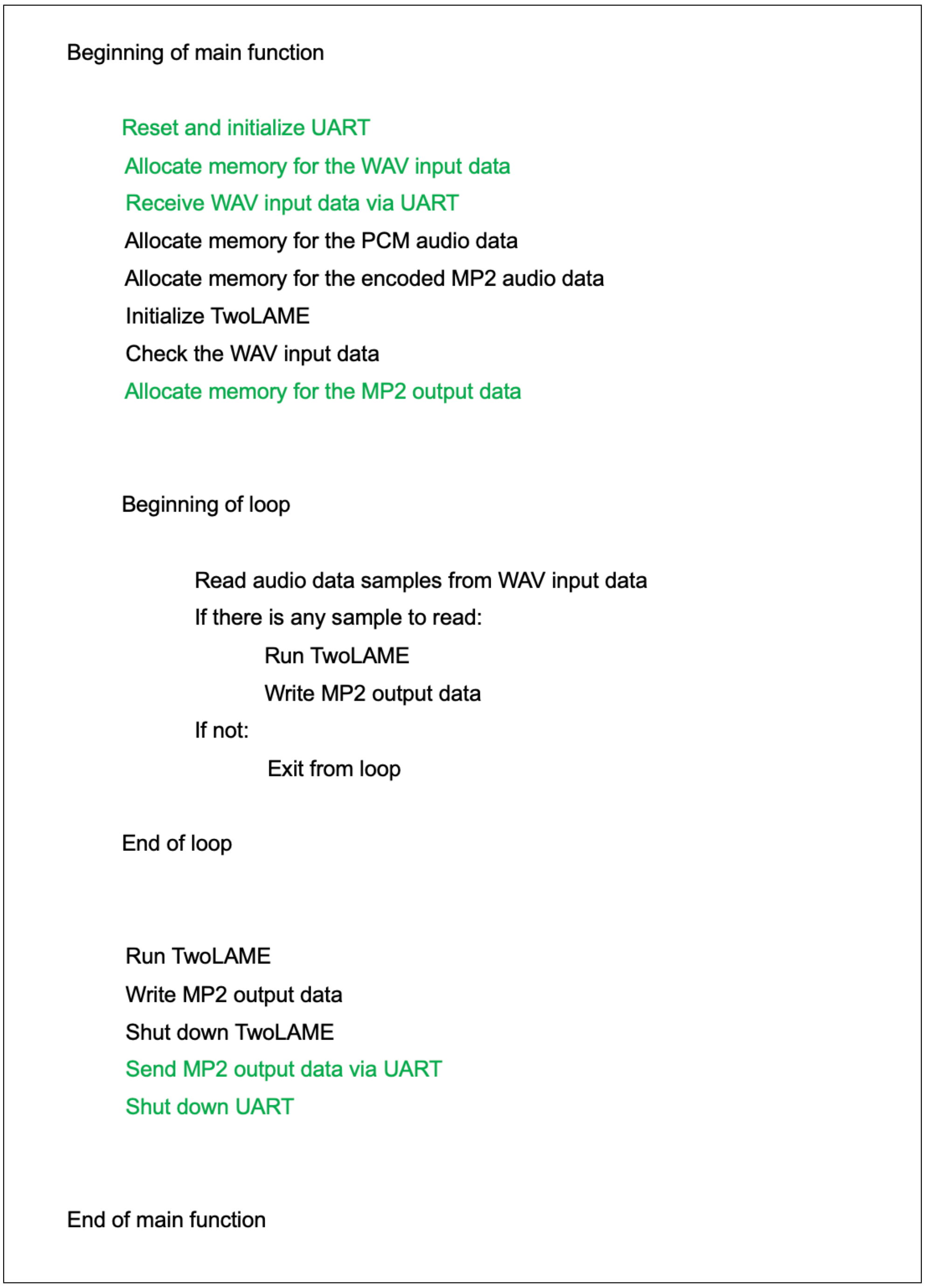
The fourth and last generated file was *vivaldi.wav*. This file was not generated from scratch in *Audacity*. Instead, it was simply converted to WAV format. The original file, entitled 'Vivaldi - Spring', was downloaded from Internet (for free) and opened on *Audacity*. After that, the track was reduced to the initial 4 seconds and exported as WAV. In fact, what motivated the use of this file was the recognised quality and complexity of Vivaldi songs.

With four audio files available at the repository, it was possible to verify the *TwoLAME* software. Using a standard *automake* process, composed of *./configure*, *make*, and *make install* commands, *TwoLAME* was successfully installed. Then, by simply typing *stwolame* followed by the input file name, white space, and the output file name (the one desired), the software would run and produce an MP2-encoded version of the original file. It is important to mention that both input and output file names should include the file extension. For this work, the relevant extensions are '.wav' as input and '.mp2' as output. From executing *stwolame short.wav short.mp2* command, it was verified that both the original and the encoded audio files sound the same (for a human listener). The same happened with the second and fourth file 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. This is because white noise is the worst case scenario for audio encoding, mainly due to its high requirements.

After testing the original *TwoLAME* software, it was necessary to implement it in IOb-SoC. The porting of the software requires a front end, as the encoding process uses functions of *libtwolame* (*TwoLAME*'s library).

3.2 Firmware

The front end should instantiate all the required functions to produce an MP2 audio. Considering this, the *simplefrontend* example was taken from the *TwoLAME* repository, improved, and implemented in *firmware.c*. To realize what modifications are necessary to run *TwoLAME* in IOb-SoC, figure 8 shows the pseudo code for the *main()* function in *firmware.c*.



```
Beginning of main function

    Reset and initialize UART
    Allocate memory for the WAV input data
    Receive WAV input data via UART
    Allocate memory for the PCM audio data
    Allocate memory for the encoded MP2 audio data
    Initialize TwoLAME
    Check the WAV input data
    Allocate memory for the MP2 output data


    Beginning of loop

        Read audio data samples from WAV input data
        If there is any sample to read:
            Run TwoLAME
            Write MP2 output data
        If not:
            Exit from loop

    End of loop


    Run TwoLAME
    Write MP2 output data
    Shut down TwoLAME
    Send MP2 output data via UART
    Shut down UART


End of main function
```

Figure 8: 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 between IOb-UART and IOb-Console. There are also two additional memory allocations for the input and output data. These operations are done through *malloc* using unsigned char type, because UART peripheral does transmit 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 owing to the fact that the IOb-SoC does not have 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 via UART but also the output audio data is sent 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*. Afterwards, 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-SoC system and *TwoLAME* front end. 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.

3.3 Optimization

With the *TwoLAME* already implemented in firmware.c, it is possible to emulate the program, by running it on a PC. It is also possible to simulate its execution on a PC. However, the major interest is in running *TwoLAME* in FPGA, as the goal is to encode MP2 audio in real-time. Considering this, some basic software improvements were done before evaluating the performance in FPGA.

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, without compromising the encoded audio quality. The other reason was IOb-SoC not supporting double precision, since it contains floating-point units with single precision, exclusively. Apart from altering the macro, one function in *subband.c* was also changed. The original code contained *modf*, a function that belongs to *math.h* and requires double precision. Therefore, it was substituted by *modff*, which has the same functionality (returns the fractional part) but uses single precision.

The second change was more strategic, as it involved a more careful analysis of the encoding software. It is known that trigonometric functions are computationally complex, and *TwoLAME*, as an audio-processing software, requires different mathematical functions. For this reason, some math functions were removed and substituted by tables, wherever it was possible. To evaluate this possibility, the *TwoLAME* was emulated on a PC, compiled with different 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 upper limit of 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 being 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*.

With basic software optimizations already implemented, it is desired to measure the execution time of the *TwoLAME* software for different input data. This requires a different memory management since the FPGA has memory space available, and so allocating memory during program execution is not ideal, except in special cases. Considering this, a macro called *PC_EMUL_RUN* is defined in order to control each memory allocation, through an if directive. In a 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-SoC macros from *config.mk*, the IOb-SoC configuration script (explained in the next section). 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.

3.4 Profiling

With everything set up, the last implementation in the software architecture was **profiling**. 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. To do so, an eight component was added to IOb-SoC, the **TIMER**. 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 understanding how it works. Similarly to the other components, the TIMER is tracked by a *GitHub* repository (forked from *IObundle, Lda*), integrating the IOb-SoC 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.

With this, IOb-SoC was ready for profiling. The process 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 6 shows the first phase of the profiling for all input files.

	Input files			
	short.wav	long.wav	noise.wav	vivaldi.wav
*TwoLAME*_init()	3	2	3	0
wave_init()	0	0	0	0
*TwoLAME*_set_num_channels()	0	0	0	0
*TwoLAME*_set_in_samplerate()	0	0	0	0
*TwoLAME*_set_bitrate()	0	0	0	0
*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()	0	0	0	0
*TwoLAME*_close()	0	0	0	1
TwoLAME total	1614	24719	20210	26467

Table 6: Execution time for all input files [ms].

The previous table 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. By analyzing *twolame_encode_buffer_interleaved*, it is noticeable that the relevant operation inside the function is *encode_frame()*, as all the other operations 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 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 7 shows the second phase of the profiling for all input files.

		Input files			
		short.wav	long.wav	noise.wav	vivaldi.wav
***TwoLAME* _encode _buffer _interleaved()**	*TwoLAME* function 1	0	9	5	3
	TwoLAME function 2	3	139	52	78
	TwoLAME function 3	0	16	7	8
	TwoLAME function 4	141	3487	2590	3662
	TwoLAME function 5	7	172	142	201
	TwoLAME function 6	0	13	3	6
	TwoLAME function 7	0	5	6	4
	TwoLAME function 8	1377	19062	16222	20829
	TwoLAME function 9	0	17	8	10
	TwoLAME function 10	16	424	343	485
	TwoLAME function 11	2	12	7	7
	TwoLAME function 12	0	4	1	4
	TwoLAME function 13	0	11	11	17
	TwoLAME function 14	1	22	20	27
	TwoLAME function 15	21	503	346	488
	TwoLAME function 16	12	306	149	210
	TwoLAME function 17	0	10	4	2
	TwoLAME function 18	0	5	4	2
	TwoLAME function 19	0	9	3	3
	TwoLAME function 20	0	3	0	0
	TwoLAME function 21	0	1	0	1
	TwoLAME function 22	0	7	1	3
	TwoLAME function 23	0	3	1	2
TwoLAME total		1614	24719	20210	26467

Table 7: Execution time for all input files [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. 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 8 shows the third phase of the profiling for all input files.

		Input files			
		short.wav	long.wav	noise.wav	vivaldi.wav
***TwoLAME* function 8**	psycho_3 function 1	635	642	635	635
	psycho_3 function 2	5	133	115	155
	psycho_3 function 3	6	117	77	120
	psycho_3 function 4	47	1138	886	1253
	psycho_3 function 5	40	1026	791	1115
	psycho_3 function 6	7	170	148	204
	psycho_3 function 7	7	153	258	173
	psycho_3 function 8	8	220	167	235
	psycho_3 function 9	0	4	2	4
	psycho_3 function 10	3	45	38	67
	psycho_3 function 11	611	15168	12975	16684
	psycho_3 function 12	0	23	14	29
	psycho_3 function 13	0	9	6	13
TwoLAME total		1614	24719	20210	26467

Table 8: Execution time for all input files [ms].

The previous table shows interesting information. First, it is noticeable that several functions have an insignificant execution time, compared to others. Second, it is clear that *psycho_3 function 11*, which is *psycho_3_threshold*, 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. Therefore, *psycho_3_threshold* should be the first function accelerated in Hardware.

4 Hardware Architecture

This chapter addresses the hardware architecture. It starts by setting up *Versat* in the IOb-SoC. Then, it checks the *Versat* Functional Units that are available in order to accelerate the *psycho_3_threshold* function. At the end, two hardware accelerators are developed, with each one executing part of the mentioned *TwoLAME* function.

4.1 Versat

After measuring the execution times of *TwoLAME* in IOb-SoC, through profiling, it was well-marked that *psycho_3_threshold* function needed hardware acceleration, i.e., it should be executed by custom-made hardware. This would bring the possibility to reduce between 37% to 64% total execution time, depending on the audio input. At this stage, the IOb-SoC was comprising eight main components: **AXI**, **CACHE**, **LIB**, **MEM**, **VEXRISCV**, **TWOLAME**, **UART** and **TIMER**. Given *Versat* being the optimal choice for accelerating the *psycho_3_threshold* function, integration into IOb-SoC becomes imperative. Therefore, the first step in this process was adding **VERSAT** to the IOb-SoC. This was simple as *VERSAT* was previously implemented and tested by *IObundle, Lda*. Apart from adding it as submodule to IOb-SoC's repository, some paths were changed to allow correct compilation. The second step was more complex, consisting in adding an interconnection between *Versat* and RAM. Iob-SoC usually contains a single AXI interface, used by both cache memory and CPU. Considering this, the solution was to double the size of the wires from the already existing AXI interface, providing an additional interface to *Versat* while avoiding a new separate interface.

4.1.1 Functional units

In practical terms, *Versat* is a reconfigurable hardware accelerator that contains several functional units in its source, all written in Verilog. Nonetheless, the user has the possibility of creating and adding new ones to the source. In this work, the hardware accelerators use a set of 14 functional units, of which 5 were specifically created to allow accelerating *psycho_3_threshold* function (*FloatLess*, *FloatGreater*, *FloatGreaterEqual*, *Mux4* and *Conditional1*). The remaining 9 units were already included in the units source.

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

Const

- This module receives a 32-bit input and outputs the same 32-bit value.

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.

FloatLess

- This module receives two 32-bit floating-point inputs and outputs a 32-bit value (repeated 1-bit result 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.

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.

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.

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 to 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.

4.1.2 Operators

Versat also contains several operators defined in its specification source. The operators execute either over the right operand or between the left and right operands. In this work, the hardware accelerators use a set of 6 operators that were already included in the source. A brief description of each operator is provided in the list below.

[−]

- This operator negates the right operand.

[$\sim$]

- This operator negates the MSB 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.

[$|$]

- 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.

Understanding the resources that *Versat* offers, the subsequent step involves examining the *psycho_3_threshold* function. This 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 in order 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.

4.2 First accelerator

Concerning hardware acceleration, a crucial step is to determine the control and data paths of the software that is intended to be accelerated. In other words, the paths refers 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 needs to be accelerated. It includes decisions, branching, loops, and other control structures that determine the program's behavior. 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.

Considering this, optimizing both the control and data paths is essential to develop an efficient hardware design that aligns with the software's requirements.

4.2.1 Control and Data paths

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

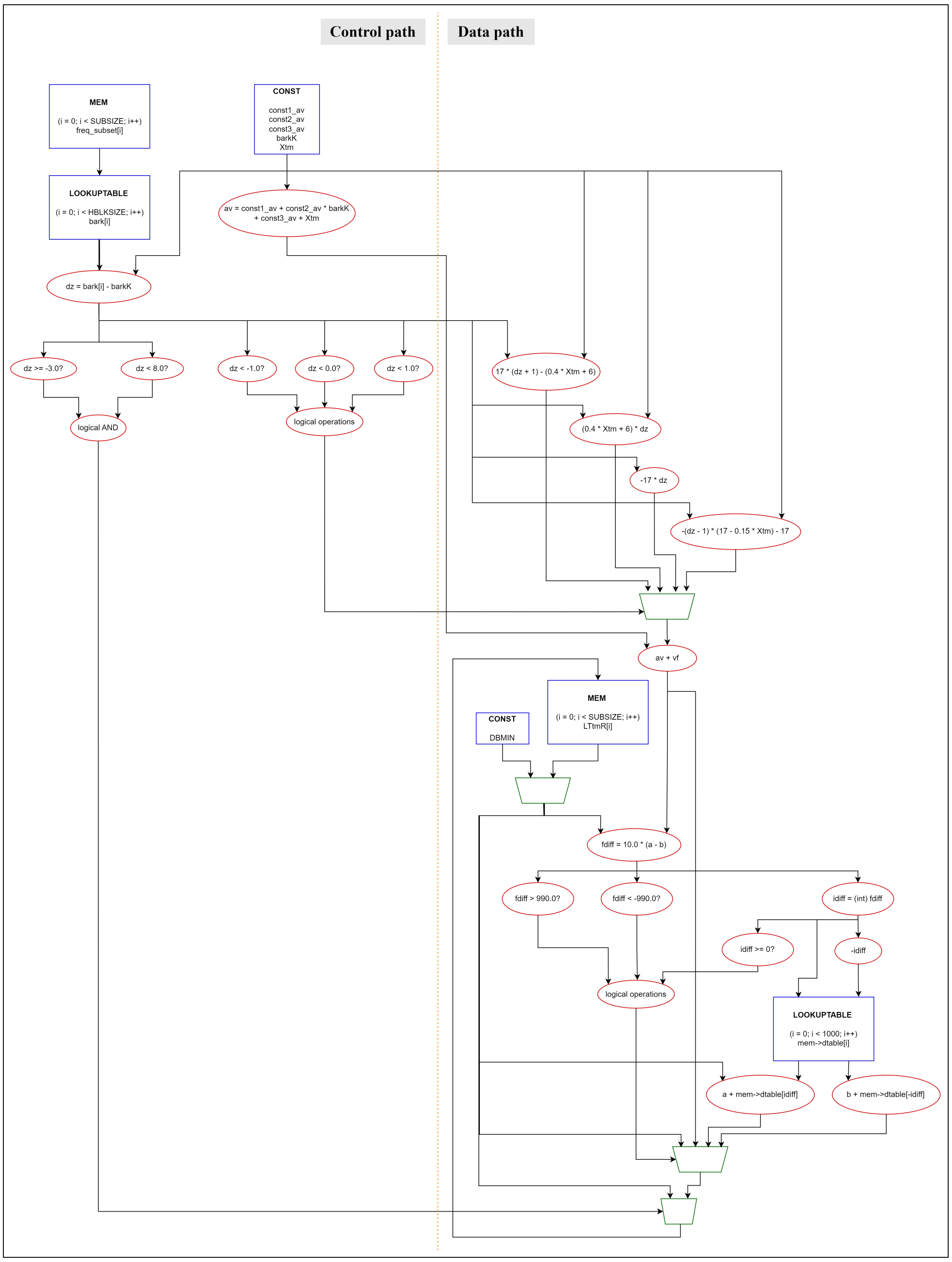


Figure 9: Control and data paths of the first hardware accelerator.

4.2.2 Modules

After analysing the control and data paths, the hardware accelerator is developed in *versatSpec.txt*. This file specifies the whole data process, which can be divided in 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

1. Calculation of *av_tone*:
   - Takes *barkK*, *Xtm*, *const1_av*, *const2_av* and *const3_av* to calculate *av_tone*.
2. Calculation of *dz*:
   - Calculates *dz* based on the inputs *freq_subset*, *bark* and *barkK*.
3. Calculation of *vf*:
   - Takes the output of *sub*, *logic1* and *Xtm* as inputs.
   - Outputs *vf*.
4. Float addition (*add1*):
   - Takes the output of *mul1* and *const1* as inputs.
   - Adds *mul1* and *const1*.
5. Input multiplexing (*mux2*):
   - Uses a multiplexer to select one of the inputs (*DBMIN*, *LTtmR*).
6. Calculation of *fdiff*:
   - Subtracts input *b* from the input *a*.
   - Multiplies the result by *const1_psycho_3*.
7. Greater than comparison (*gt_psycho_3:0*):
   - Compares the input *fdiff* with *const2_psycho_3* for greater than condition.
8. Less than comparison (*lt_psycho_3*):
   - Compares the input *fdiff* with *const3_psycho_3* for less than condition.
9. Float to integer conversion (f2i_psycho_3):
   - Converts the floating-point input *fdiff* to an integer *idiff*.
10. Greater or equal comparison (*ge_psycho_3*):
    - Compares the input *idiff* with *const4_psycho_3* for greater than or equal condition.
11. Float addition (*add1_psycho_3*):
    - Takes the output *mux2* and *mem* as inputs.
    - Adds *mux2* and *mem*.
12. Float addition (*add2_psycho_3*):

- Takes the output of *add* and *mem* as inputs.
- Adds *add* and *mem*.

13. Output *LTtmR* (*psycho_3_add_db1*):

    - Takes the outputs of *mux2*, *add*, *add1_psycho_3*, *add2_psycho_3*, *sel* and *dzRange1* as inputs.
    - Takes the output of *psycho_3_add_db1* as the final output of this module.

4.2.3 Firmware

Apart from specifying the data process, configuring 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 configure the accelerator with at least two sets of input data. This is done in two separate functions because part of the data configuration is valid for all the runs. Part of the configuration 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 configured by assigning the intended value to the constant name (from the hardware accelerator). For floating-point data the *PackInt* function is required, which receives the intended value as argument. There is also a memory unit configuration 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 configurations for *bark* and *mem→dbtable* arrays. These set two lookup tables, with SUBSIZE and HBLKSIZE, respectively.

In *configureVersat* four constants are configured as they differ between runs. An input of *mux2* multiplexer is also configured 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 argument the number of times that the accelerator should run with the exact same configuration, which is 1.

4.3 Second accelerator

4.3.1 Control and Data paths

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

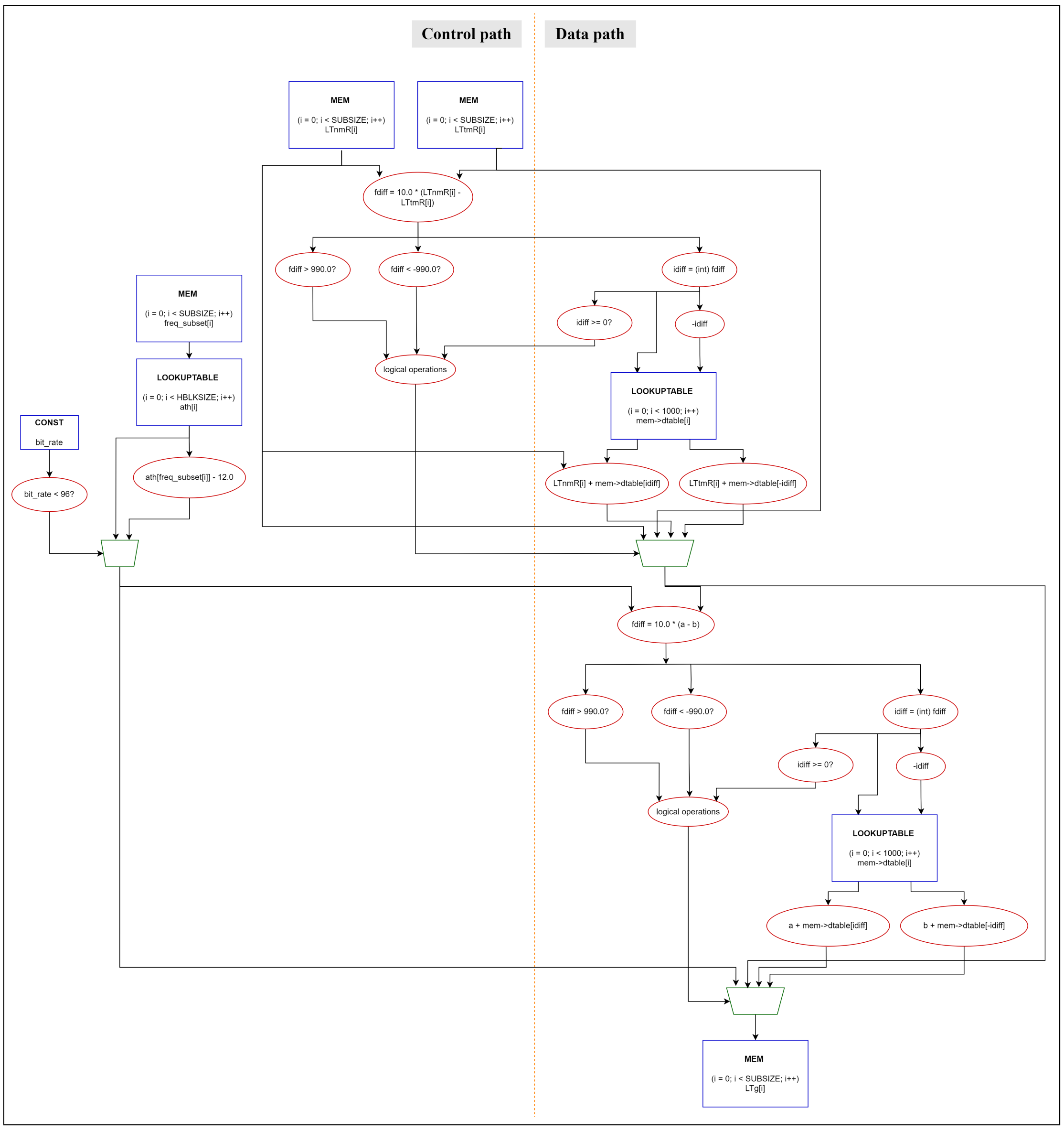


Figure 10: Control and data paths of the second hardware accelerator.

4.3.2 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 similarity with the *start* module of the first accelerator, since it contains most of the *start* module description but duplicated. In addition, it excludes the initial part related to the inputs *av*, *dz* and *vf*, and adds a new one related to conditional operation.

Therefore, a brief description of the conditional operation part is provided below.

1. Less than comparison (*lt_psycho_31*):
   - Compares the input *bit_rate* with *const1* for less than condition.
2. Calculation of $ath - 12.0$ (*sub1_psycho_31*):
   - Subtracts the input *const2* from *ath*.
3. Conditional operation (*conditional_psycho_3*):
   - Takes the output of *lt_psycho_31* and *ath* as inputs.
   - Takes the output of *conditional_psycho_3* as output. The selected input is determined by the value of *sub1_psycho_31*.

4.3.3 Firmware

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

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

5 Results

6 Conclusion

References

[1] International organization for standardization. URL https://www.iso.org/home.html.

[2] International electrotechnical commission. URL https://iec.ch/homepage.

[3] Iso/iec 11172-3:1993 information technology — coding of moving pictures and associated audio for digital storage media at up to about 1,5 mbit/s — part 3: Audio. URL https://www.iso.org/standard/22412.html.

[4] Field-programmable gate array. URL https://en.wikipedia.org/wiki/Field-programmable_gate_array.

[5] Application-specific integrated circuit, . URL https://en.wikipedia.org/wiki/Application-specific_integrated_circuit.

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

[7] IEEE Standard for VHDL Language Reference Manual. *IEEE Std 1076-2019*, 2019.

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

[9] Coreworks, s.a. URL https://coreworks.com.

[10] Xilinx, inc. URL https://www.xilinx.com/.

[11] Intel corporation. URL https://www.intel.com/content/www/us/en/homepage.html.

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

[13] IPbloq, Lda. URL https://ipbloq.com/.

[14] *MPEG-2 Codec CX23415*. Conexant Systems, Inc, 2003. URL https://datasheet.ciiva.com/26931/getdatasheetpartid-486546-26931824.pdf.

[15] Conexant systems, inc. URL https://en.wikipedia.org/wiki/Conexant.

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

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

[18] Asynchronous serial interface (asi), . URL hhttps://en.wikipedia.org/wiki/Asynchronous_serial_interface.

[19] Internet protocol (ip). URL https://en.wikipedia.org/wiki/Internet_Protocol.

[20] Lame. URL https://lame.sourceforge.io/about.php.

[21] Gnu lesser general public license (lgpl). URL https://en.wikipedia.org/wiki/GNU_Lesser_General_Public_License.

[22] Dynamic-link library (dll). URL https://en.wikipedia.org/wiki/Dynamic-link_library.

[23] Toolame. URL https://en.wikipedia.org/wiki/TooLAME.

[24] Twolame. URL https://www.twolame.org/.

[25] Floating-point arithmetic (fp). URL https://en.wikipedia.org/wiki/Floating-point_arithmetic.

[26] Floating-point unit (fpu). URL https://en.wikipedia.org/wiki/Floating-point_unit.