

Cuprins

1.Rezumat în limba Engleză.....	3
1.1 Theoretical Fundamentals	3
1.1.1 Sound Equalizer	3
1.1.2 Zybo	3
1.1.3 FPGA-Overview.....	4
1.1.4 Vivado Design Suite	4
1.1.5 VGA	4
1.2 Implementation	5
1.2.1 Software Implementation	5
1.2.2 Hardware Implementation	7
1.2.2.1 Zynq	7
1.2.2.2 AXI Interconnect	7
1.2.2.3 AXI IIC.....	8
1.2.2.4 Audio Core	8
1.2.2.5 Constant Core	8
1.2.2.6 Video.....	8
1.2.2.7 Processor System Reset.....	9
1.3 Experimental Results	9
2 Planificarea activității	11
3 Stadiul actual	12
4 Fundamentare teoretică.....	15
4.1 Egalizator de sunet	15
4.2 FPGA	17
4.2.1 Considerații generale.....	17
4.2.2 Construcția.....	18
4.2.3 Funcționarea.....	18
4.2.4 Aplicații	19
4.2.5 DMA (Sistem de acces direct la memoria de eșantionare)	19
4.3 Limbajul VHDL.....	19
4.4 VIVADO	20
4.5 Limbajul C	20
4.6 VGA	21
4.7 XILINX Vitis.....	22

4.8 Protocolul AXI4	22
4.9 SSM2603 Codec Audio de mică putere	24
5 Implementarea soluției adaptate	26
5.1 Implementare Hardware	26
5.1.1 Zynq7 processing System	26
5.1.2 AXI Direct Memory Access - Memoria de acces directă	27
5.1.3 AXI Interconnect	28
5.1.4 Processor System Reset.....	29
5.1.5 AXI IIC.....	29
5.1.6 Audio Core	29
5.1.7 Video	29
5.1.8 Funcționare Implementare Hardware	31
5.2 Implementare Software.....	33
6 Rezultate experimentale	40
7 Concluzii.....	48
8 Bibliografie.....	Error! Bookmark not defined.
9 Anexe	Error! Bookmark not defined.
10 CV	Error! Bookmark not defined.

1.Rezumat în limba Engleză

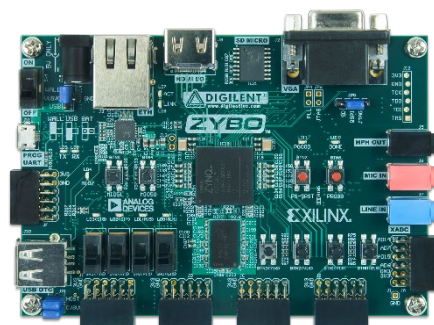
Using the Zybo development board as a sound equalizer. The Zybo is an entry-level embedded software and digital circuit development platform built around the smallest member of the Xilinx Zynq family. The Zybo is compatible with Xilinx's high-performance Vivado Design Suite, and Vitis SDK. Connecting the Zybo development board by the VGA to implement the graphical image.

1.1 Theoretical Fundamentals

1.1.1 Sound Equalizer

The sound equalizer, it's a hardware and a software platform used to process the sound. The sound signal, it's filtered and modified by the sound equalizer. To make a sound equalizer we need to implement different filters to our audio signal, every filter must do something different in order to modify the sound. A sound equalizer uses a Video interface to show the volume of the signal, using different colors to show how powerful the volume of the signal is at some moment. Another very important task that a sound equalizer is doing is cleaning the audio signal from noise and the most common attribute of a sound equalizer is the option to modify the volume.

1.1.2 Zybo



Figură 1. Zybo Board (Sursă: Digilent)

The Zybo is an entry-level embedded software and digital circuit development platform. It integrates a dual-core ARM Cortex-A9 (650Mhz) processor with Xilinx 7-series Field Programmable Gate Array logic. It also has a DDR3 memory controller with 8 DMA channels, high-bandwidth peripheral controllers (1G Ethernet, USB 2.0, SDIO), and SDIO Low-bandwidth peripheral controller (SPI, UART, CAN, I2C). Other features: Dual-role HDMI port, 16-bits per pixel VGA source port, MicroSD slot, an audio codec with headphone out, microphone, and line-in jacks. Information source at [26].

1.1.3 FPGA-Overview

An FPGA (field-programmable gate array) is an integrated circuit designed to be configured by a customer. The FPGA configuration is specified using a hardware description language.

FPGA contains an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects allowing blocks to be wired together, it can be used to solve any problem which is computable, this is trivially proven by the fact that FPGA can be used to implement a soft microprocessor. Information source at [25].

1.1.4 Vivado Design Suite

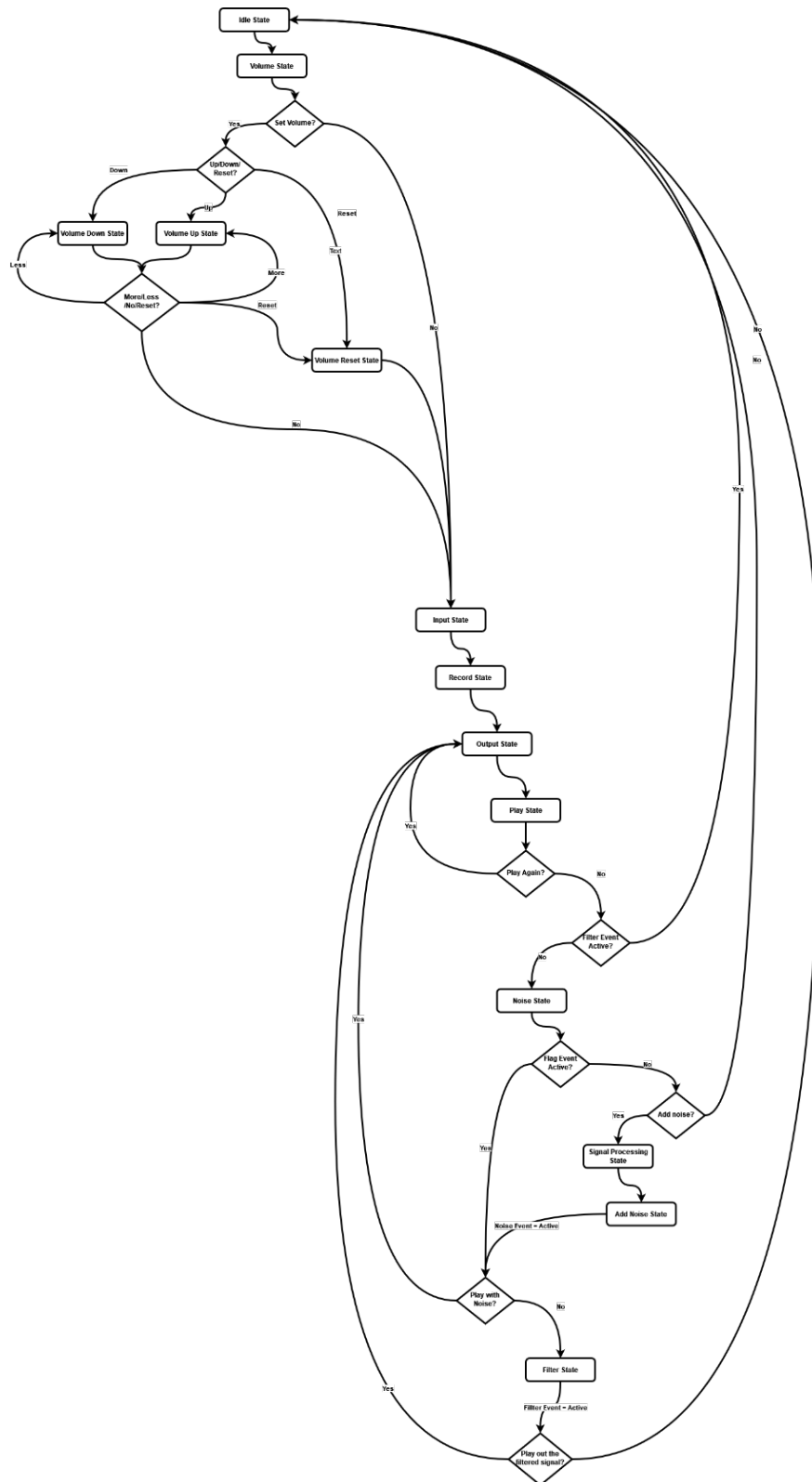
Vivado Design Suite is a software suite produced by Xilinx for the synthesis and analysis of hardware description language designs, superseding Xilinx ISE with additional features for system-on-chip development and high-level synthesis. Vivado enables developers to synthesize designs, examine RTL diagrams, and configure the target device. It is a design environment for FPGA products exclusively from Xilinx. Information source at [8].

1.1.5 VGA

VGA (Video Graphics Array) is a standard type of connection for video devices, an analog video standard created by IBM in 1987 for the IBM PS/2 series of computers. VGA cables have 15-pin connectors which are 5 pins at the top, 5 in the middle, and the other 5 at the very bottom. Computers commonly used VGA connections between a video card and a monitor. Today, this analog interface is used for high-definition (HD) video, including even resolutions of 1080p and higher. VGA systems provide a 720x400 resolution in text mode, and a 640x480 resolution in graphics mode, with 16 colors or monochrome. The VGA color signal is made from 5 different signals, Horizontal synchronization signal (HSYNC), vertical synchronization signal (VSYNC) and the RGB signals, blue, green and red. HSYNC it's responsible for every new line on our display, and VSYNC for a new row, these two signals will determine the resolution of the screen, and every pixel will be controlled by the color signals. Information source at [14].

1.2 Implementation

1.2.1 Software Implementation



Figură 2. Xilinx Project Hierarchy

In the main file it was constructed a software platform that implements a state machine who interact with the user. Before the state machine, in an embedded system, we need to initialize the drivers, so the project can start. The drivers that are initialized are the following:

- Interrupt Controller
- IIC Controller
- USER I/O driver
- Initialize DMA
- Initialize Audio I2S

If we want to filter all the data, it was necessary to allocate space in Heap using malloc function. The data type that was used it's u32 because it has a range of 4,294,967,295.

The diagram from the picture [Figura 2] represents the structure that is used to interact with the user, made with a state machine that was created in Vitis SDK using C embedded.

At the start of the diagram, we are in "Idle State", that's the default state, where nothing it's happening, we have no audio sound at this moment, only on the terminal will show us the menu.

The terminal that I used is Tera Term, where we need to select the serial com and to use a frequents of 115200Hz.

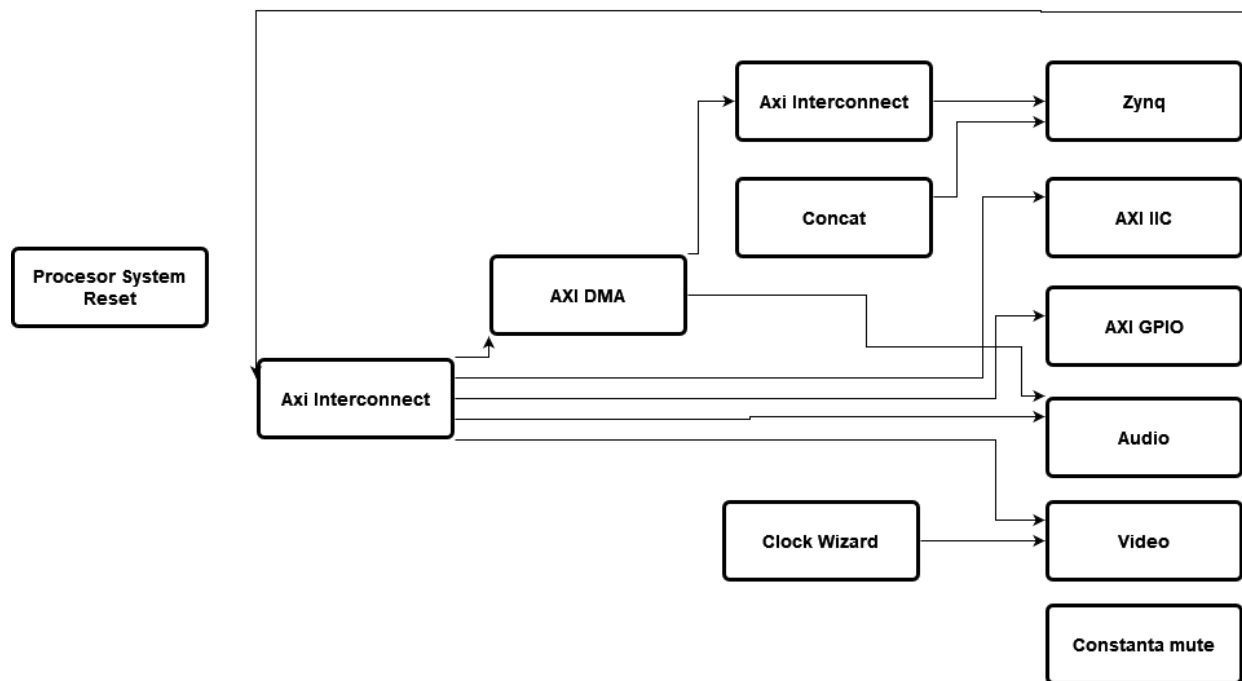
The next state, it's "Volume State", where we can change the volume by increasing the dB's or to lower the volume by decreasing the dBs of the signal. Another two options from that we can choose are reset, where we set the sound volume to it initialize value and the other one it's "no" if we don't want to change anything. In terminal, we wait for the value from the UART, which can be "u" for setting the value up, "d" for setting the value down, "r" for reset the value and "n" if we just want to move further.

In the function fnSetLineInputVolumeUp, which it's integrated in the SetVolumeUp state, we have the register's R0_LEFT_ADC_VOL and R1_RIGHT_ADC_VOL where we can set one of the 14 options for the volume. The volume can be increased from 0 to 33dB and decrease it. The audio codec that is used it's called SSM2603, it is made by Analog Devices and has a number of 18 registers that can be used.

The next state, it's "Input State", where we initialize the drivers for the record and the playback. Then we move on to the "Record State". In this case, we can record an audio signal with the length of 3 seconds. The next state it's "Output State", where the registers R4 and R5, who are responsible for the analog path are configured. After that, we move on "Play State", where we are asked if we want to play again the sound or to move further. If we don't want to play again, we move to "Noise State". In the "Noise State", if the flag that indicates that we already have noise on the signal, it's valid, then we are asked if we want to play the sound as it is. If we don't have noise on our signal, and we want to add, we will be moving towards the "Signal Processing State". In this state, it was created a new signal in Vitis SDK who will be added to the original recorded signal. After that we are moving in "Add Noise State", where if the flag that indicates if there is any noise is active then we will be asked if we want to play it. If we want that we are moved to the "Output State", or if we didn't want it, we are moved to "Filter State". In the filter state, we apply an EMA filter (Exponential Moving Averages), wight it's created using a low-pass filter. This type of filter its very used in trading and stock market to create a line who indicates the average moving of the stocks for certain amount of time.

This filter is used to smooth the signal by calculating for every new sample, the sum of the previous sample from the new signal and the new one, which it's calculated by the sum between the original sample and the previous sample from the new signal, this results it's multiplied by alpha. Alpha it's calculated using a low pass filter coefficient.

1.2.2 Hardware Implementation



Figură 3. Vivado Project Hierarchy

1.2.2.1 Zynq

Processing system 7 it's the software interface that it's created for Zynq-7000, is based on the Xilinx All Programmable system-on-chip architecture. The SoC style is integrating PS and PL unit. The Processing System 7 core acts as a logic connection between the PS and PL while helping you to integrate IP cores and costume modules. For the output we use DDR pin with communicates with the DDR memory, M_AXI_GP0, master slave for AXI4Lite communication, FCLK_RESET0_N who is a general reset signal from PS to PL, FCLK_CLK0 which operates the PL side at 100kHz frequency. The input IRQ_F2P signal controls the interrupts from the system.

1.2.2.2 AXI Interconnect

In this system we use two Axis Interconnect cores who facilities the AXI4Lite communication protocol. With this protocol, we can connect our cores to the Vitis platform. This type of communications facilities the use of registers, who are necessary in Vitis to configure the functionality of the cores. These are used in the design to connect the DMA memory with the Zynq core, working by the clock signal. Another Interconnect core works to connect all the other

cores with the AXI4Lite protocol. This protocol work with the master-slave style, so we have the master interconnect core with the slave AUDIO, GPIO, IIC and DMA core.

1.2.2.3 AXI IIC

AXI IIC core it provides a low-speed, two-wire, serial bus interface with it's used to communicate with the Zynq processing system by the IIC2INTC_Irpt signal which it's the core system interrupt output. It works by the reset and clock signal provided by the power system reset core.

1.2.2.4 Audio Core

The audio core its facilities the audio signal, it works by the AXI4 Lite protocol having master and slave signals. It receives data by SDATA_I which it's an input pin and send data by SDATA_O. It communicates directly with the DMA core by the AXI_S2MM signal. If we want the core to work, we must add some modifications. One it's adding a second PL fabric clock and the other one is enabling the I2C interface for the communication of the control signals between the Zynq PS and the core.

1.2.2.5 Constant Core

Constant value core it's used to generate a single bit value called dout [0:0]. Signal ac_muten controls the mute status for the audio signal.

1.2.2.6 Video

At the start, we need to see how the VGA works. It is required many pixels, the resolution where we want to display the sound bar is 1980x1080 pixels and the for that we need 5 different signals, one for the horizontal synchronization, one for the vertical synchronization and 3 signals that represent the 3 most important colors, green, red, blue.

For the VGA technology we need to have the back porche, front porche and the retrace area for the vsync and hsync signals. In the retrace area these signals need to be in 1 logic, for that we use 4 process s, 2 for the hsync signal and 2 for the vsync signal. These 2 processes (two sets) are used to count the h_cntr_reg from 0 to 2200 (total period on horizontal), the v_cntr_reg to 1125 (total period on vertical) and to set the h_sync_reg and v_sync_reg to 1 logic when the counter is in the retrace area.

For this component, we used the Hsync and Vsync counter signal, that are starting from the top left of the display area and are counting to the last pixel in horizontal and vertical direction. We define 8 white bars that will change color for every bit of 1 from our data that's send from Vitis. The first bar will be green, the second one will be light green and so on until we reach red which means the volume is at maxim. On the screen, each bar can take any color from the spectrum, due to the large number of colors that can be display by the combination between each of the 3 fundamental colors.

In the Vitis platform, we can give the data to be received by the video core in Vivado and to be display on the screen. For this we need to connect the video RTL core due to AXI4 interface.

The RTL core it needs a frequency of 148.5MHz which was calculated with the following formula $\text{pixels/line} \times \text{lines/screen} \times \text{screens/second} = 148.5\text{M pixels/second}$.

1.2.2.7 Processor System Reset

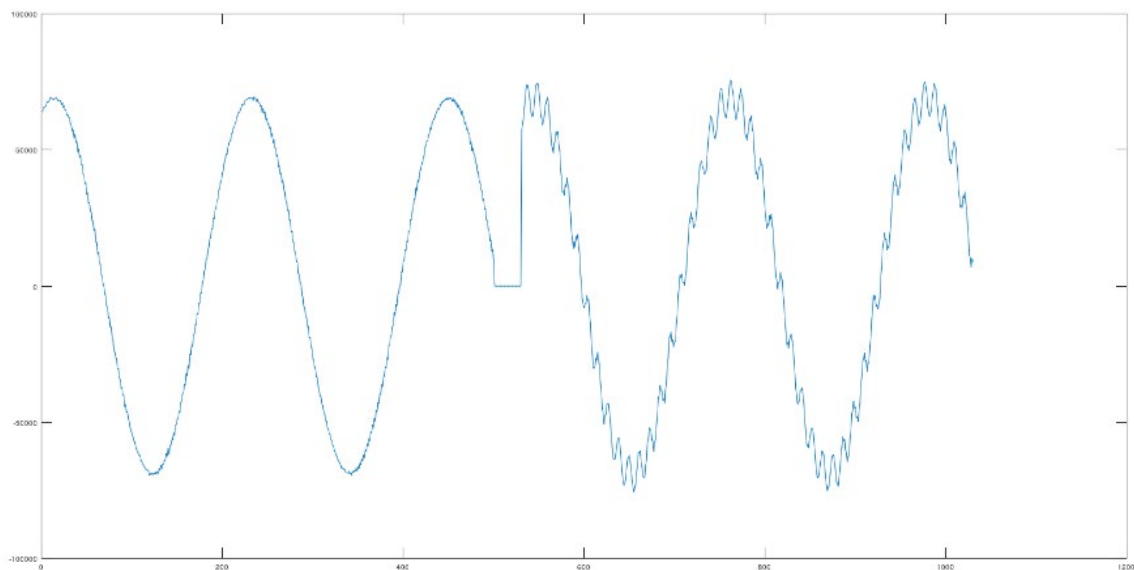
Processor System Reset it's a Xilinx IP that it was used to provide the reset signal for our system. We use two output signal interconnects `_aresetn` with provides the active-low reset to interconnect and peripheral `_aresetn` which provides the active-low reset to peripherals. The input signal `slowest_sync_clk` it's connected to the slowest synchronous clock (FCLK_RESET0_N) with a frequency of 100kHz and the `ext_reset_in` signal which it's used to reset the core.

1.3 Experimental Results

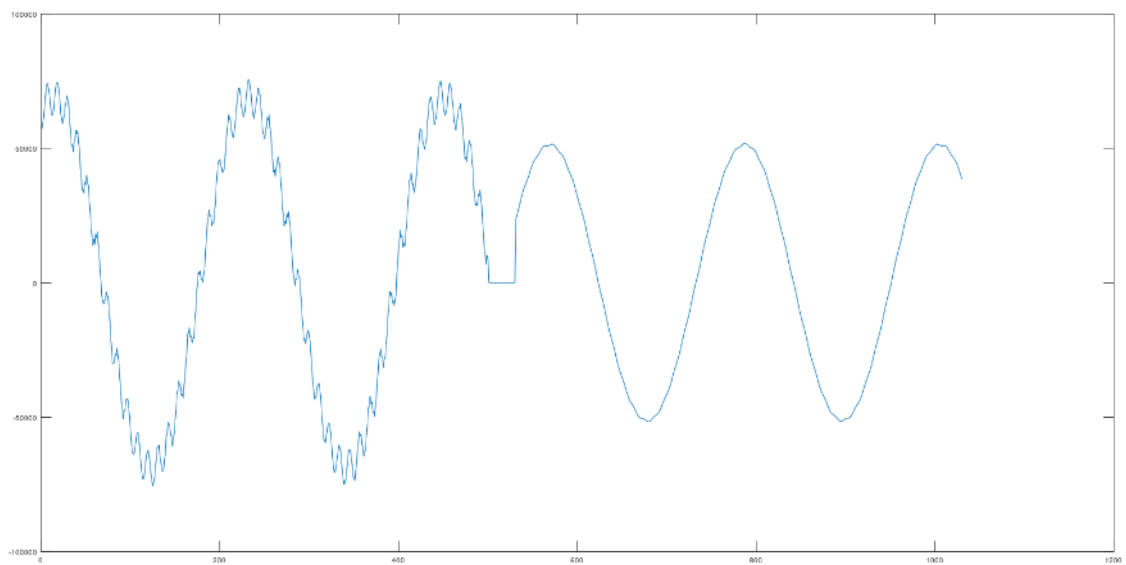
For the experimental results, I moved through the state machine. Firstly, we need an audio sound, I used Online Tone Generator to generate a sine signal with 440Hz frequency. For this result's I didn't change the volume. I added noise to the audio signal, in the first figure we can see that the audio sample it's very alternated by the noise. To see the difference between the two signals, I implement a line of 0's.

On the second image, I illustrate the filtered audio sound. The EMA filter that was applied to the signal with the noise is filtering the signal and make him smoother and liner, the new audio sample can be used now for other applications.

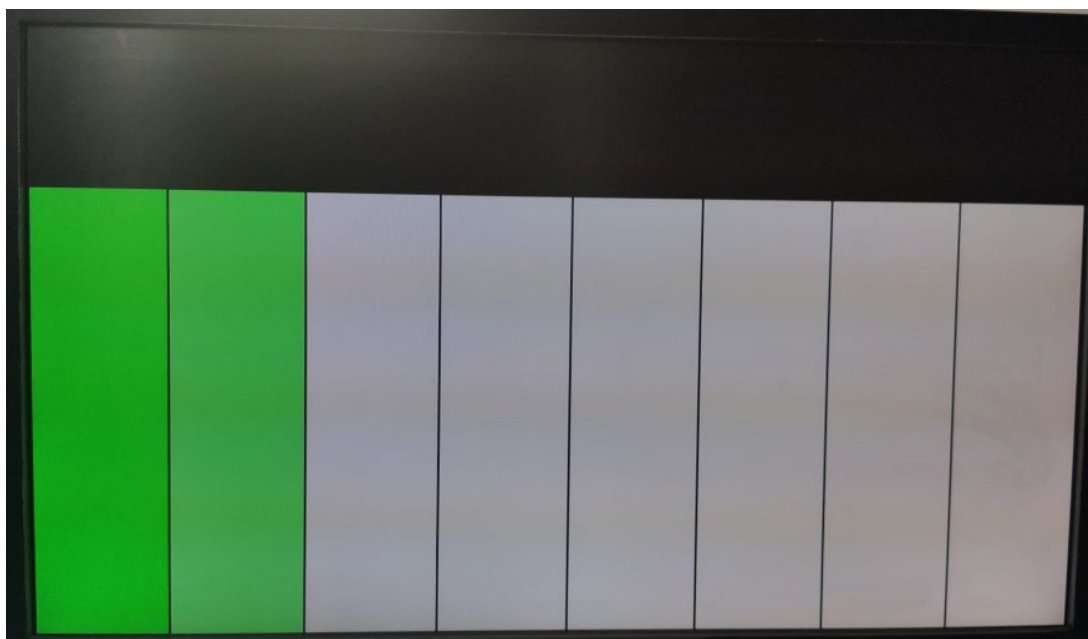
On the last image we can see the graphical part that was make using VGA, we can see that for a 14% volume, that was set from the laptop we have 2 green bars, because 14% it's equivalent to 0011 in binary.



Figură 4. Original sound (left) and the sound with noise (right).



Figură 5. The sound with noise (left) and the filtered sound (right)



Figură 6. Sound bar for 14% volume

2 Planificarea activității

Număr curent	Activitate	Data de început	Data de sfârșit
1.	Întâlnirea cu profesorul de licență și stabilirea temei	19.10.2022	19.10.2022
2.	Primirea plăcuței Zybo	22.10.2022	22.10.2022
3.	Implementarea proiect Hello World	14.11.2022	14.11.2022
4.	Versionare proiect GitHub	21.11.2022	21.11.2022
5.	Începere scriere parte teoretică	22.11.2022	22.11.2022
6.	Implementare cod hw și sw	11.03.2023	18.03.2023
7.	Continuare cod	18.03.2023	20.04.2023
8.	Implementare video	15.04.2023	25.04.2023
9.	Implementare filtru	25.04.2023	10.05.2023
10.	Scrie Documentație	10.05.2023	20.05.2023
11.	Revenire cod în C, implementare filtru	20.05.2023	30.05.2023
12.	Continuare documentație	30.05.2023	05.06.2023
13.	Verificare cod	05.06.2023	20.06.2023
14.	Terminare documentație	20.06.2023	28.06.2023

3 Stadiul actual

Un egalizator de sunet reprezintă o platformă ce aplică diferite efecte asupra semnalului înregistrat. Acestea au ca și rol filtrarea semnalului pentru diferite frecvențe, de eliminare a zgomotului, de adăugare a altor semnale sau alte aplicații ce pot altera semnalul. Lucrarea de licență cuprinde toți pașii necesari pentru a realiza un astfel de proiect. Pornind de la implementarea diagramei bloc în Vivado, împreună cu conectarea prin VGA. Pană la crearea interfeței în Vitis SDK, unde prin intermediul unei mașini de stare se poate modifica tema și funcționalitatea proiectului. Astfel, acest proiect poate să fie folosit pentru dezvoltarea ulterioară a altor proiecte, având baza deja făcută. În momentul actual se pot adăuga diferite componente în diagrama bloc din Vivado, precum switch-uri sau butoane, placa Zybo are 4 switch-uri și 4 butoane care pot să fie programate pentru a putea pune orice interacțiune dorită. Precum și activarea resturilor de porturi care în momentul de față au rămas libere. Aceste porturi sunt, SD Micro, HDMI I/O, Ethernet, USB. Mașina de stare implementată în Vitis poate să fie modificată și extinsă, precum implementarea unui număr mai mare de filtre sau să fie tratate toate cazurile posibile din fiecare State.

În proiectul de față s-au rezolvat o serie mare de erori, cele mai des întâlnite sunt erorile de timp deoarece diferite componente din diagrama bloc necesită o anumită frecvență de ceas. O altă problemă foarte des întâlnită este următoarea, executabilul Makefile din Vitis SDK se strica foarte des, acesta se regenerează la fiecare compilare nouă a programului și trebuiau adăugate următoarele linii de cod.

- | |
|--|
| <ol style="list-style-type: none">1. INCLUDEFILES=\$(wildcard *.h)2. LIBSOURCES=\$(wildcard *.c)3. OUTS=\$(wildcard *.o) |
|--|

Fișierele MakeFile sunt răspândite în platforma, acestea se găsesc în fișierul Zynq fsbl, în fișierul ps7_cortexa9_0, dar se pot regăsi și în alte fișiere din proiect. Aceste fișiere conțin un set de reguli și de comenzi care descriu modul în care este compilat programul.

Tot pentru rularea acestui proiect se recomandă să se folosească un computer cu o viteză mare de procesare, programe precum Vitis și Vivado folosesc o cantitate mare din resursele interne ale pc-ului, memoria RAM este recomandată să fie de minim 16Gb, proiectul să fie rulat și stocat de pe un SSD cu memorie de minim 256Gb (atât Vitis, Vivado cât și programele adiționale trebuie instalate pe acest SSD iar pachetul Xilinx 2022.1 are un volum de aproximativ 108Gb fără componentele de Ultra Scale) iar procesorul se recomandă să fie o generație cât mai nouă și cât mai puternică. În caz contrar o generare de fișier .bit în Vivado poate eșua, cu aceste cerințe minime, o simplă generare de fișier .bit poate dura până la 10 minute. Se recomandă să nu se îngreuneze căile către fișierele sursă, deoarece Vivado nu răspunde favorabil la o cale prea lungă. Un alt lucru foarte important este ca singurul compilator de pe pc să fie componentele de la Xilinx, există posibilitatea ca prin instalarea unui alt compilator care folosește aceleași librării să se piardă căile către librăriile din Vitis.

Acest proiect se poate modifica și porta pentru alte plăci precum Zybo Z7 sau ZedBoard.

O altă modificare ce se poate aduce la proiect și prin care putem aduce proiectul la un stadiu mai actual cu tehnologiile din ziua de azi, este de a modifica componenta video din afișare prin VGA în afișare prin HDMI. Pentru a face această schimbare este nevoie să înțelegem cum se face

afișarea prin HDMI, folosind parametrii de timp precum sincronizarea pe verticală și sincronizarea pe orizontală, pixel clock și așa mai departe. Respectiv trebuie să înțelegem cum funcționează protocoalele specifice pentru acest tip de afișare.

Tot o funcționalitate ce se poate adăuga asupra acestui proiect, este următoarea, prin folosirea switch-urilor să se modifice filtrul aplicat asupra semnalului. Avem un număr de 4 switch-uri pe placa Zybo, acestea pot să corespundă pentru 16 valori diferite (de la 0000, 0001 până la 1111) în care fiecare combinație să reprezinte o acțiune asupra filtrului.

Pentru a îndeplini cerințele de realizare a unui egalizator de sunet, trebuie să aplicăm un filtru asupra semnalului nostru. Filtrul ales este EMA, un filtru des folosit la bursă. În continuare se pot adăuga o multitudine de filtre adiționale precum, filtrul trece sus, filtru trece bandă, filtrul oprește bandă, filtru Gaussian sau diferite filtre de reducere a zgomotului.

Un alt pas ce se poate implementa este de a modifica proiectul să poată adăuga zgomot și să filtreze semnalul pe o scară mult mai largă, de exemplu o întreagă piesă muzicală, dar pentru ca acest lucru să poată să fie realizat este nevoie de o alocare uriașă a memoriei.

O bază a proiectului o constituie cartea Zynq Book, care se regăsește gratuit pe internet. Autorii acestei cărți sunt: Louise Crockett, Ross Elliot, Martin Enderwitz, Bob Stewart, David Northcote. Această carte cuprinde 5 capitole, prin care se trece prin toți pași necesari pentru a realiza un proiect care se bazează pe arhitectura unui FPGA, folosind ca și unelte de dezvoltare Vivado, Vitis SDK și Vivado HLS. Ca și plăci de dezvoltare care pot fi folosite pentru a putea parcurge toate cele 5 capitole sunt Zynq sau Zed. În primele capitole ale cărții suntem învățați cum să cream o platformă pentru proiectul nostru, folosind ca și placă de dezvoltare Zynq, apoi toți pași necesari pentru a realiza un bloc design în Vivado ce conține în primă fază, core-ul pentru sistemul de procesare Zynq7, core-ul pentru reset, core-ul pentru conectarea prin AXI și un core pentru GPIO, care ulterior sunt folosite pentru a manipula câteva leduri. Apoi sunt prezentați pașii care ne arată cum să exportăm un fișier xsa folosit anterior pentru a crea o platformă în SDK, respectiv cum să completăm un xdc prin care se comunică cu placa noastră. În continuare sunt prezentați toți pașii necesari pentru a realiza un IP, iar în ultimul capitol ne este arătat cum se realizează core-ul de audio, care ulterior este implementat în proiectul de licență. Cartea este reprezentată la sursa bibliografică [19].

Un alt proiect bazat pe FPGA a fost creat de către Raman Mangla care este student la universitatea din Toronto cu specializarea Computer Engineering. Acesta a făcut un egalizator de sunet digital creat pentru placa Altera DE1 SoC (Cyclone 5 FPGA) folosind limbajul de programare hardware Verilog HDL. Acest proiect procesează semnalul audio în timp real, folosind un filtru FIR (Impulse Response Filter). Acest filtru corespunde pentru trei frecvențe trece-jos (0-500Hz), trece-banda (550-3000Hz) și trece-sus (3000Hz+). Proiectul este reprezentat la sursa bibliografică [20].

Încă un proiect asemănător a fost realizat de către Vladi Litmanovich și Adi Mikler, studenți la facultatea de inginerie din Tel Aviv. Acesta a fost proiectul lor pentru concursul Xilinx Open Hardware 2017 University Design și a fost câștigător la categoria studenți. Acestea au creat un sistem care aplică diferite efecte asupra unei chitare electrice, folosind un FPGA și placa de dezvoltare Zed. Pentru măsurători aceștia au folosit un osciloscop, vizualizând fiecare efect în parte.

Un alt proiect a fost realizat la universitatea Ecole centrale de Lyon din Ecully Franța, acesta a fost finalizat în Ianuarie 2022 și a fost fondat de către ANR (French Research Concil). Acestea au creat o platformă bazată pe FPGA folosită pentru procesare audio în timp real, această platforma este accesibilă și ușor de programat folosind limbajul de programare Faust. Ca și semnale țintă, se axează pe semnale audio de înaltă performanță și foarte scăzute latentă audio. Placa de dezvoltare folosită este Digilent Zybo Z7 care este bazată pe Xilinx Zynq 7000 și se dorește adăugarea de suport pentru alte plăci. În acest proiect s-a folosit SSM2603 care este codecul integrat pe Zybo Z7, dar s-a folosit și codec-ul ADAU1787 creat de cei de la Analog Devices, acesta fiind mai performant. Pentru acest proiect s-a creat și un PCB care poate să fie folosit pe post de controler. Proiectul este reprezentat la sursa bibliografică [22].

Pentru a putea duce la bun sfârșit acest proiect, s-au folosit bucăți de cod din proiectul de audio creat de către cei de la Digilent. Referința pentru acest cod se află la referința bibliografică [27]. Acest proiect creat de către ei folosește Vivado 2016.1 și Vivado SDK 2016.1 și este folosit pentru a înregistra și reda o secvență audio timp de 5 secunde.

Pentru gestionarea proiectului s-a folosit Git și Github.

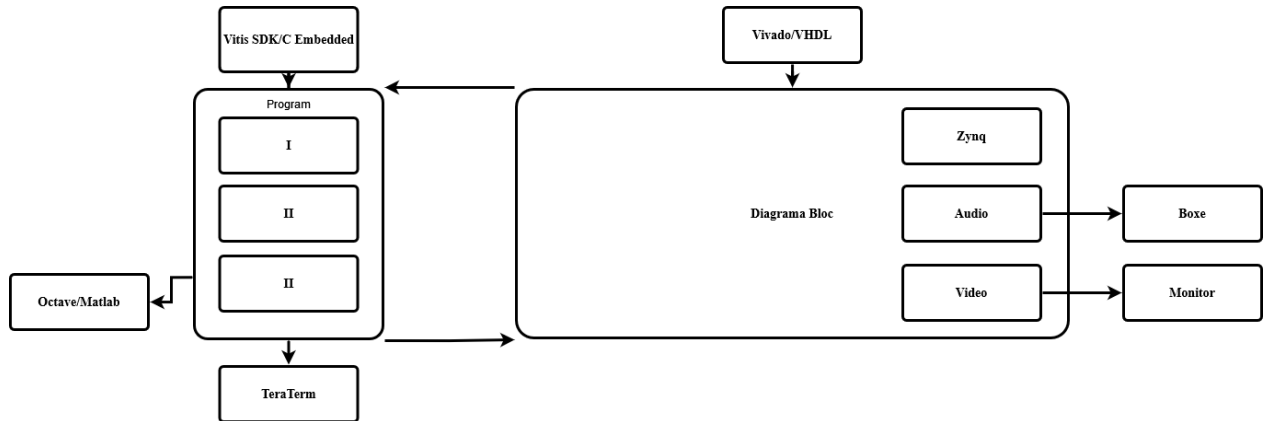
Git este un sistem de versionare a proiectelor care salvează o listă a modificărilor efectuate asupra fișierelor. Acesta este folosit de către programatori pentru dezvoltarea de noi proiecte și pentru a ține un istoric al lucrărilor. Principalele avantaje date de către git sunt, viteza mare de funcționare, înțelegerea ușoară a comenzilor și posibilitatea de a lucra pe același proiect de un număr mai mare de programatori, care efectuează modificări asupra diferitelor aspecte ale proiectului, lucrând pe o ramura diferită și ulterior fiecare parte nou adăugată să fie introdusă în proiectul final.

GitHub este o platformă care permite gestionarea proiectelor. Acestea sunt încărcate dintr-un repozitor local care se află pe unitatea noastră centrală (Pc/laptop s.a.m.d) într-un repozitoriu din această platformă. Are un grad foarte mare de securitate, proiectele pot să fie atât open-source (libere pentru toată lumea) sau private. Astfel un grup mare de colaboratori pot să lucreze în același timp la același proiect, se poate ține un istoric al muncii lor și se poate revine la o versiune ulterioară dacă se dorește acest lucru

Proiectele de pe platforma GitHub.com pot să fie accesate și manageriate folosind comenzi standard de Git, toate comenzile de Git sunt valide și funcționează pe GitHub. Platforma le permite utilizatorilor, să navigheze liberi prin repozitoriile publice. Acesta conține și o parte socială în care utilizatorii pot să urmărească alți utilizatori, și să vadă ultimele actualizări ale proiectelor respectiv ramura pe care lucrează. Documentație GIT[23-24].

În urma experimentelor făcute, s-a realizat ca nu este nevoie sa modificăm volumul semnalului de zgomot, ci este nevoie să modificăm doar frecvența, deoarece urechea umană este mai sensibilă la o frecvență mai mare și percepe semnalul ca fiind mai tare.

4 Fundamentare teoretică



Figură 7. Diagrama Bloc Egalizator de Sunet

4.1 Egalizator de sunet

Un egalizator de sunet este un dispozitiv atât software cât și hardware de procesare a sunetului. Se filtrează sau se aplică benzile de frecvențe diferite care alcătuiesc semnalele audio. Acest tip de procesare poate să fie utilizat în timpul înregistrării sunetului, amestecării sau întăririi sunetului.

O sursă sau un generator de sunet este alcătuit dintr-o multitudine de unde sonore care sunt împrăștiate pe un spectru larg de frecvențe audio. Un adult poate să perceapă valori cuprinse între 20Hz și 20KHz. Astfel este de dorit să se poată izola anumite benzi de frecvență asupra cărora dorim să aplicăm un anumit tratament. Date preluate din sursă bibliografică [11].

Cele mai multe egalizatoare de sunet folosesc 2 sau 3 filtre în componență. Un filtru este un dispozitiv electronic sau o prelucrare software care se aplică asupra unui semnal și modifică parametrii acestuia. Cele mai des utilizate filtre în momentul de față pentru prelucrarea semnalului audio sunt filtrul trece jos care rejectează eșantioanele mai înalte de un anumit prag, filtrul trece sus care rejectează eșantioanele ce sunt sub un anumit prag, filtru trece bandă care permite trecerea eșantioanelor care se află între anumite praguri și filtru oprește bandă care rejectează eșantioanele între anumite praguri. Pe lângă aceste filtre de bază se pot adăuga orice tip de filtru existent asupra semnalului audio. Alte filtre folosite asupra semnalelor sunt filtrul EMA (Exponential Moving Average) și SMA (Simple Moving Average). Documentație despre filtrul EMA [21].

În lucrarea de față s-a ales ca și filtru implementat filtru EMA, deoarece acesta are în componență lui un filtru trece jos și reușește să filtreze semnalul cu zgomot. Funcționarea filtrul EMA se bazează pe preluarea a două celor mai apropiate eșantioane dintr-un semnal la care s-a adăugat sau nu zgomot. Acest filtru calculează media valorilor unui număr predefinit de

eșantioane, pentru cazul nostru s-au ales 1500 de eșantioane din semnalul filtrat, deoarece acestea trebuie stocate și prelucrate iar memoria este limitată. Media rezultată se folosește pentru a obține o valoare filtrată pentru fiecare eșantion în parte, iar valorile recente o să fie mai importante decât cele precedente. Pentru primul eșantion se va calcula EMA inițial, acesta va fi valoarea de bază pentru valoarea următoare.

Pentru a putea realiza media necesară acestui filtru, avem nevoie de factorul de pondere, acesta are valori cuprinse între 0 și 1. Factorul de pondere se calculează prin diferite metode, metoda implementată este de a folosi un filtru trece jos care să calculeze această valoare. Avem nevoie de acest coeficient deoarece ne arată cât de important este eșantionul curent față de eșantionul trecut.

$EMA = EMA_anterior + factor_de_pondere * (valoarea\ curentă - EMA_anterior)$

Valoarea factorului de pondere se calculează cu ajutorul componentelor filtrului trece jos. Pentru filtrul trece jos de ordinul întâi (sau filtrul RC) componentele pe care se bazează sunt RC, Alpha, dt și CUTOFF.

- RC reprezintă produsul dintre rezista filtrului (R) și capacitatea filtrului (C).
- Dt este perioada de eșantionare.
- Alpha este factorul de pondere.
- CUTOFF este frecvența de tăiere, este acea marjă de la care filtru începe să atenueze.

$$RC = \frac{1}{CUTOFF * 2 * 3.14} \quad (1)$$

$$dt = \frac{1}{Rata\ de\ eșantionare} \quad (2)$$

$$Alpha = \frac{dt}{RC + dt} \quad (3)$$

Pentru a realiza un egalizator de sunet, probabil cea mai importantă întrebuintă este aceea de a putea modifica volumul semnalului. Placa de dezvoltare Zybo folosește codec-ul audio SSM2603 care are o serie de 15 registrii ce pot să fie folosiți pentru a modifica semnalul audio, respectiv pentru a putea înregistra și reda secvența audio. Prin acești registrii s-au implementat funcții ce modifică volumul semnalului.

Dacă ne dorim să vedem mai clar efectul unui filtru asupra semnalului audio, putem să adăugăm zgomot astfel încât acesta să fie distorsionat.

Toate aceste componente se adaugă printr-un state machine care se realizează în Vitis SDK folosind C embedded, acesta comunică prin Axi4Lite cu, componentele din diagrama bloc din Vivado.

Un sound bar are și o parte vizuală care permite utilizatorului să vizualizeze diferitele valori ale eșantioanelor dintr-un semnal. Această parte poate să fie realizată cu ajutorul blocurilor componente din Vivado, ulterior adăugată la diagrama bloc și conectată prin protocolul de comunicare AXI4. Ulterior aceasta să fie comandată din Vitis SDK, unde să se facă o filtrare a valorilor. Blocul component se scrie în limbajul de programare hardware VHDL prin care se comunică logica specifică pentru un sound bar ce comunică prin VGA cu un desktop.

Toate aceste componente care realizează un egalizator de sunet, sunt create pe placa de dezvoltare Zybo de la Digilent. Aceasta conține un FPGA (Field-Programmable Gate Array) de la Xilinx, și anume un FPGA Zynq-7000 SoC (System-on-Chip). Acest SoC se combină cu două nuclee ARM Cortex-A9. Documentație pentru VGA [14].

4.2 FPGA

4.2.1 Considerații generale

FPGA (Field Programmable Gate Array) este un circuit integrat digital configurabil, de către programator. Programarea unui FPGA se face cu ajutorul limbajelor de programare de descriere hardware (HDL/VHDL), s-au creat simulatoare care traduc instrucțiuni din limbajul C/C++ într-un limbaj de programare hardware, precum Vitis HLS care prelucrează din fișier C în VHDL. FPGA-urile sunt alcătuite din blocuri logice configurabile legate între ele de o serie de conexiuni. Date preluate din sursă bibliografică [25].

Un concept adaptat de către FPGA este cel de matrice logice LCA (Logic Cell Array), care include mai multe părți: Configurable Logic Block (CLB), IOB (Input Output Block) și canale folosite pentru legătură.

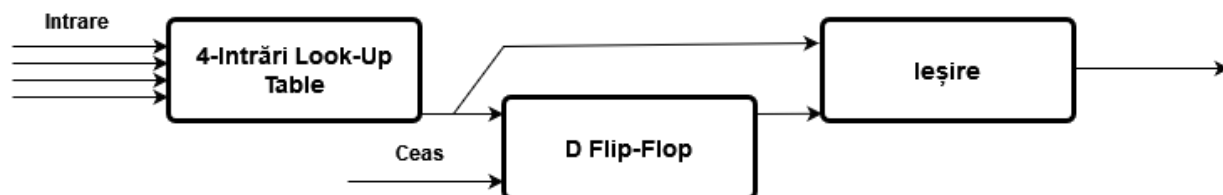
Față de circuitele logice bine cunoscute, FPGA are o structură diferită, astfel FPGA folosește un tabel de căutare pentru a implementa logică combinațională și fiecare tabel de căutare este conectat la un flip-flop-D, pot să fie realizate atât funcții logice secvențiale cât și funcții logice combinaționale.

Toate aceste module sunt conectate direct între ele, FPGA poate avea multe moduri de programare, astfel o să reamintim câteva dintre ele, primul este modul master-paralel, iar altele sunt module master-slave care poate suporta programarea mai multor FPGA-uri.

Având în vedere tehnologia folosită în procesul de fabricare pe piață au ieșit mai multe tipuri de FPGA-uri. Printre acestea reamintim SRAM, care au la bază celule SRAM, Antifuze care pot să fie programate doar cu ajutorul unui dispozitiv special, Fuse, asemenea cu cele menționate anterior, pot să fie programate doar cu ajutorul unui dispozitiv special, EPROM (Erasable Programmable Read-Only Memory), pot să fie programate doar o singură dată și după necesită ștergerea conținutului prin raze ultraviolete, EEPROM (Electrically Erasable Programmable Read-Only Memory) similare cu EPROM doar că sunt reprogramabile, iar ultimele sunt Flash care au la bază celule Flash.

În ziua de azi, FPGA-urile sunt tot mai folosite, astfel acestea se dezvoltă cu o viteză mare, urmărind să aibă o capacitate tot mai mare de prelucrare și să consume tot mai puțină energie.

4.2.2 Construcția



Figură 8. Arhitectura unui CLB din FPGA

În principal arhitectura unui FPGA are 3 tipuri de module. Acestea sunt blocuri de intrări/ieșire, Switch Matrix/ Interconnection Wires și blocuri logice configurabile (CLB). Arhitectura unui FPGA constă într-o matrice bidimensională de blocuri logice cu un mijloc ce îl lasă pe programator să utilizeze, respectiv să configureze conectarea între blocurile logice.

Configurable logic block (CLB) include logică digitală, mai multe intrări/ieșiri și implementează logica utilizatorului.

Interconnect oferă directive între blocurile logice folosite pentru a utiliza logica programatorului.

I/O Pad sunt folosite pentru a comunica cu tot ce ține de “exterior”.

Blocurile logice conținute de către un FPGA sunt următoarele:

- MUX, acesta este un dispozitiv al cărui rol este de a permite unul sau mai multe semnale de intrare analogice sau digitale de mare viteză să fie selectate. Un MUX poate să fie asemănat cu un comutator care are mai multe intrări și o singură ieșire.
- Flip-Flop-D este un dispozitiv cu ceas, numit și bistabil deoarece acesta poate avea două stări, S și R respectiv două ieșiri, Q și Q negat. Este des întâlnit în circuite master-slave. (Sunt o varietate mare de bistabile)
- LUT implementează funcțiile logice combinaționale.

Astfel un FPGA este creat din aproximativ sute sau mii de blocuri logice configurabile.

4.2.3 Funcționarea

Pentru dezvoltarea aplicațiilor pe FPGA s-au folosit limbajele de programare hardware VERILOG și VHDL. Limbaje care lucrează la nivele joase de abstractizare, acestea preiau din structura limbajelor de nivel înalt, doar că în cazul unui FPGA se formează descrierea arhitecturală a unui design. Prin aceste limbaje de programare ne folosim de porturi I/O (intrare și ieșire) care preiau informația și este transmisă la algoritmi de prelucrare. Aceste procese se execută concurrent unul cu celălalt și sunt implementate în funcție de un semnal de ceas. Limbajele de programare hardware diferă substanțial de cele pe nivele mai înalte, deoarece acestea se bazează pe un principiu de conectare a intrărilor/ieșirilor la o serie de blocuri logice.

Dacă dorim să verificăm funcționalitatea unui FPGA, vom crea un test-bench, care reprezintă aplicarea unui sau a mai multor stimuli la intrare și apoi verificarea ieșirii. Prin implementarea modulelor folosind un FPGA realizăm o sinteză. Sinteza este procesul de a transforma descrierea high-level a unui modul, ce nu are un corespondent în hardware, într-o descriere de nivel jos ce are un corespondent în hardware.

Pentru a putea programa un FPGA avem nevoie de un fișier XDC (Xilinx Design Constraints) pentru a asocia pinii de intrare și de ieșire ai FPGA-ului cu semnalele surselor create de către noi sau al IP-urilor importate. După ce am setat constrângerile putem implementa design-ul, pentru al putea încărca pe FPGA, avem nevoie de un fișier de programare cu extensia “.bit”.

4.2.4 Aplicații

Cele mai des întâlnite domenii de utilizare FPGA sunt:

- Automate celulare ca generatoare de secvențe pseudoaleatoare (numărătoare haotice).
- Aplicații A.I aproape de sursă, de înaltă performanță și cu consum redus de energie.
- În industria auto și moto pentru sarcini precum controlul luminilor, a camerelor.
- În industria medicală pentru dezvoltarea de aparatură modernă.
- Electronice de larg și mic consum.
- Audio, Video, procesare de imagini.

4.2.5 DMA (Sistem de acces direct la memoria de eșantionare)

Memoria de acces directă (DMA) este procedura prin care perifericele transmit direct datele de care au nevoie pentru a trimite sau a primi fără a se folosi de unitatea centrală de procesare în dezvoltare.

Prin DMA, articolele au proprietatea de a schimba informații cu memoria RAM fără a fi nevoie să ocupe timp de procesare. DMA folosește un controler dedicat, pentru a efectua tranzacții și a obține acces la magistrala de date a sistemului, obținând date de la CPU precum și pentru a primii anunțuri dacă procesul nostru a avut succes sau nu.

În timp ce sistemul de procesare Zynq are ca și default transfer memory-to-memory și nu stream-to-memory sau memory-to-stream, pentru a putea folosi canalele native DMA ale Zynq-ului, perifericele trebuie să aibă conexiune de interfațare pentru memory-to-memory transfer.

Diferența este proeminentă, deoarece procesare de tip memory-to-memory citește/scrie date dintre diferite spații de memorie în schimb ce stream-to-memory preia o reprezentare serială de date și o scrie pentru o locație specifică de memorie și atunci un memory-to-stream va citi de la o locație de memorie și după îl convertește într-un format serial.

4.3 Limbajul VHDL

Limbajul de programare VHDL este unul din limbajele standard folosite la ora actuală în industrie pentru a descrie sistemele numerice. Acronimul de VHDL vine de la VHSINC (Very High Speed Integrated Circuits) Hardware Description Language, un limbaj pentru descriere hardware a circuitelor integrate de foarte mare viteză. Cel mai înalt nivel de abstractizare este nivelul de descriere al funcționării numit în engleză behavioral. La acest nivel de abstractizare,

sistemul este descris prin ceea ce face, se facilitează conexiunea dintre intrări și ieșiri. Descrierea poate să fie o expresie booleană sau o descriere mai abstractă, la nivelul transferului între registre sau la nivelul algoritmilor. Nivelul structural, descrie un sistem ca o mulțime de porți și componente care sunt conectate între ele. Limbajul de programare hardware VHDL permite realizarea sistemelor la nivel funcțional sau structural. Nivelul funcțional cuprinde două tipuri de reprezentare, Data Flow și Algoritmice. Reprezentarea Data Flow prezintă modul cum circulă datele prin circuit, realizându-se transferul de date între regiștrii (RTL). În reprezentarea de tip algoritmic, codul este redat secvențial și se execută în ordinea implementării.

Un sistem este descris ca și o entitate principală care, poate să conțină alte entități la rândul ei, acestea fiind declarate ca și componente ale entității principale. Fiecare entitate este modelată de o declarație a entității și de corpul arhitectural. Declarația identității este o interfață cu mediul extern care definește semnalele de intrare și de ieșire, iar corpul arhitectural conține descrierea entității. Date preluate din sursă bibliografică [9].

4.4 VIVADO

Compania Xilinx a introdus un mediu de dezvoltare care se bazează pe limbajele de programare hardware Verilog și VHDL, aceasta poartă numele de Vivado, prin acest program se face sinteza și analiza modelelor HDL. Vivado a venit cu o multitudine de noi componente care sunt menite să-l ajute pe programator, include un simulator logic și un nivel înalt de sinteză.

Vivado permite integrarea de IP-uri costum sau din lista de IP-uri predefinite. Prin folosirea IP-urilor și a modulelor care sunt create de către programator se realizează schema block care simulează un circuit electronic.

Vivado este revizuit la scară largă, venind cu aproximativ 4 noi versiuni an de an, iar toate proiectele făcute într-o versiune anterioară necesită updatate la versiunea actuală, în momentul de față lucrarea este realizată pe versiunea de Vivado 2022.1, iar în prezent a apărut versiunea 2023.1. Simulatorul Vivado este o componentă Vivado Design Suite, un simulator de limbaj ce este compilat și care suportă limbaje mixte, scripturi Tcl, Ip-uri criptate și verifică funcționalitatea programului. Date preluate din sursă bibliografică [8].

4.5 Limbajul C

Limbajul de programare C este standardizat. Implementat pe majoritatea platformelor de calcul existente azi. Este apreciat pentru eficiența codului obiect generat de compilatoarele C și pentru portabilitatea sa. Sintaxa C stă la baza multor limbaje de programare din ziua de azi precum C++, Java, C#, JavaScript și multe altele. C lucrează în strânsă legătură cu hardware-ul, fiind cel mai apropiat de limbajul de asamblare. Limbajul C aparține clasei limbajelor de programare de nivel scăzut, acesta având o strânsă legătură cu echipamentul hardware. Chiar dacă C are destul de multe dezavantaje, doar un cod scris în limbajul de asamblare poate să fie mai performant decât un cod scris în C.

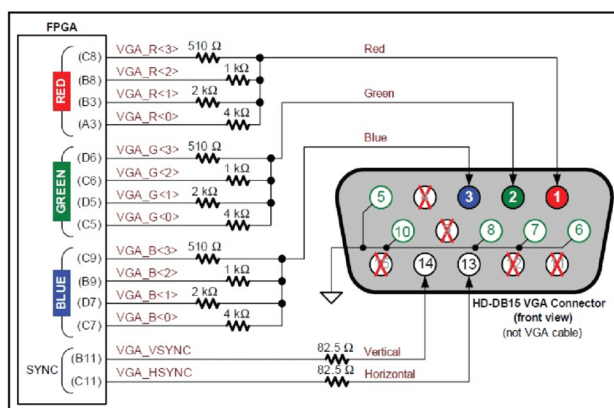
C are următoarele caracteristici importante

- Este un limbaj de bază, care conține importante funcționalități precum cele matematice și de manipulare a fișierelor.

- Este focalizat pe paradigma programării procedurale, care facilitează programarea într-un mod structural.
- Utilizează un simplu set de tipuri de date.
- Folosește un limbaj preprocesor, pentru sarcini cum ar fi definirea de macroui și includerea mai multor fișiere sursă.
- Permite accesarea la nivel scăzut a memoriei calculatorului prin utilizarea pointerilor.

Limbajul de programare C cuprinde tipuri de date cum ar fi, întregi de diferite dimensiuni, cu sau fără semn, enumerări, structuri și uniuni. Se utilizează foarte mulți pointeri, care este o referință care păstrează adresa unui obiect din memorie. Această adresă poate să fie manipulată cu ajutorul aritmeticii pointerilor. Un pointer este un tip de date complex, ce reprezintă atât adresa de memorie cât și tipul de date. Pointeri pot avea un larg mod de utilizare, precum alocarea dinamică a memoriei sau șiruri de caractere care adesea sunt reprezentate printr-un pointer la un vector de caractere. Date preluate din sursă bibliografică [10].

4.6 VGA



Figură 9. Cablul VGA (Sursă: ResearchGate)

VGA (video graphics array), este un sistem de afișare grafică, a fost introdus pentru PC-urile dezvoltate de către IBM, omniprezent în industria PC-urilor. În modul text, acesta asigură o rezoluție de 720 x 400 pixeli, iar în modul de funcționare grafic, rezoluția este de 640 x 480 sau 320 x 200. În comparație cu standardele grafice MDA, CGA, EGA, VGA utilizează semnale de natură analogică, în schimbul celor digitale. Astfel un monitor care a fost creat pentru standarde digitale mai vechi, nu poate să folosească interfața VGA. IBM a intenționat să înlocuiască VGA cu standardul Extended Graphics Array (XGA).

În ziua de azi standardul VGA este folosit pentru afișarea imaginilor video statice de înaltă rezoluție precum 1920x1080p, deși VGA poate afișa și imagini de o rezoluție mai mare.

Semnalul video color VGA este format din cinci semnale diferite: un semnal de sincronizare pe orizontală (HSYNC), un semnal de sincronizare pe verticală (VSYNC), un semnal

pentru culoarea verde, un semnal pentru culoarea roșie și un semnal pentru culoarea albastră formând astfel spectrul culorilor RGB.

Semnalul de sincronizare pe orizontală face ca fasciculul de electroni să își reia traseul pentru fiecare linie de pe lungimea ecranului VGA, asta după ce fasciculul a terminat de parcurs linia precedentă. În consecință, acest semnal va fi întotdeauna responsabil de începerea unei noi linii a ecranului. Similar, semnalul de sincronizare pe verticală va duce la afișarea unei noi imagini pe ecran. În consecință, semnalele de sincronizare pe orizontală și pe verticală vor determina rezoluția ecranului, unde culoarea afișată de fiecare pixel va fi controlată de către semnalele de culoare (R, G, B).

Semnalele RGB iau valori analogice continue, cu o arie cuprinsă între 0V pentru valoarea de negru și 0.7V pentru valoarea de alb. Fiecare semnal controlează un bus de electroni care determină fosforul ecranului să lumineze ecranul VGA. Fiecare culoare va fi un amestec vizual de diferite nuanțe ale celor 3 culori primare.

4.7 XILINX Vitis

Xilinx Vitis este utilizat pentru a crea platforma software bazată pe platforma hardware și de a crea aplicații folosind embedded C. Platforma de dezvoltare Vitis conține toate componentele software dezvoltate de către Xilinx. Această platformă software suportă atât Vitis embedded software development flow, pentru Xilinx Software Development Kit (SDK) folosit pentru crea o nouă tehnologie, și Vitis application acceleration development flow, folosit pentru dezvoltare software utilizând tehnologi bazate pe FPGA. Vitis integrated design environment (IDE) face parte din platforma de dezvoltare Vitis. Acesta este creată pentru dezvoltare asupra aplicațiilor software folosind procesoare Xilinx. Funcționează împreună cu Vivado Design Suite. Platforma de dezvoltare include. Date preluate din sursă bibliografică [12].

- C/C++ editor și compilator.
- Managementul proiectelor.
- Construcție de aplicație și generare automată Makefile.
- Navigare Erori..
- Componente focusate pe FPGA.
- Programarea Flash.
- Linii de comandă utilizate.

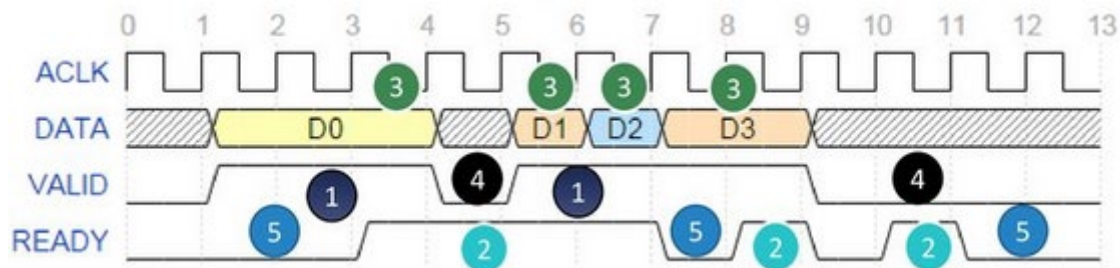
4.8 Protocolul AXI4

AXI (Advanced Extensible Interface) este un protocol de comunicare pe chip creat de către ARM și face parte din Advanced Microcontrolled Bus Architecture 3 (AXI3) și 4 (AXI4). A fost introdus în 2003 de către AMBA3, în 2010 la apariția AMBA4 se introduc AXI4, AXI4-LITE și AXI4-STREAM. AXI este folosit pentru comunicarea componentelor într-un sistem embedded precum cele bazate pe Zynq.

Protocolul de comunicare AXI4 folosește o interfață master/slave, astfel toate componentele trebuie să comunice prin același limbaj și același protocol. În circuitul nostru s-a folosit protocolul AXI4-Lite deoarece este îndeajuns pentru complexitatea circuitului. Acesta folosește 1 burst de

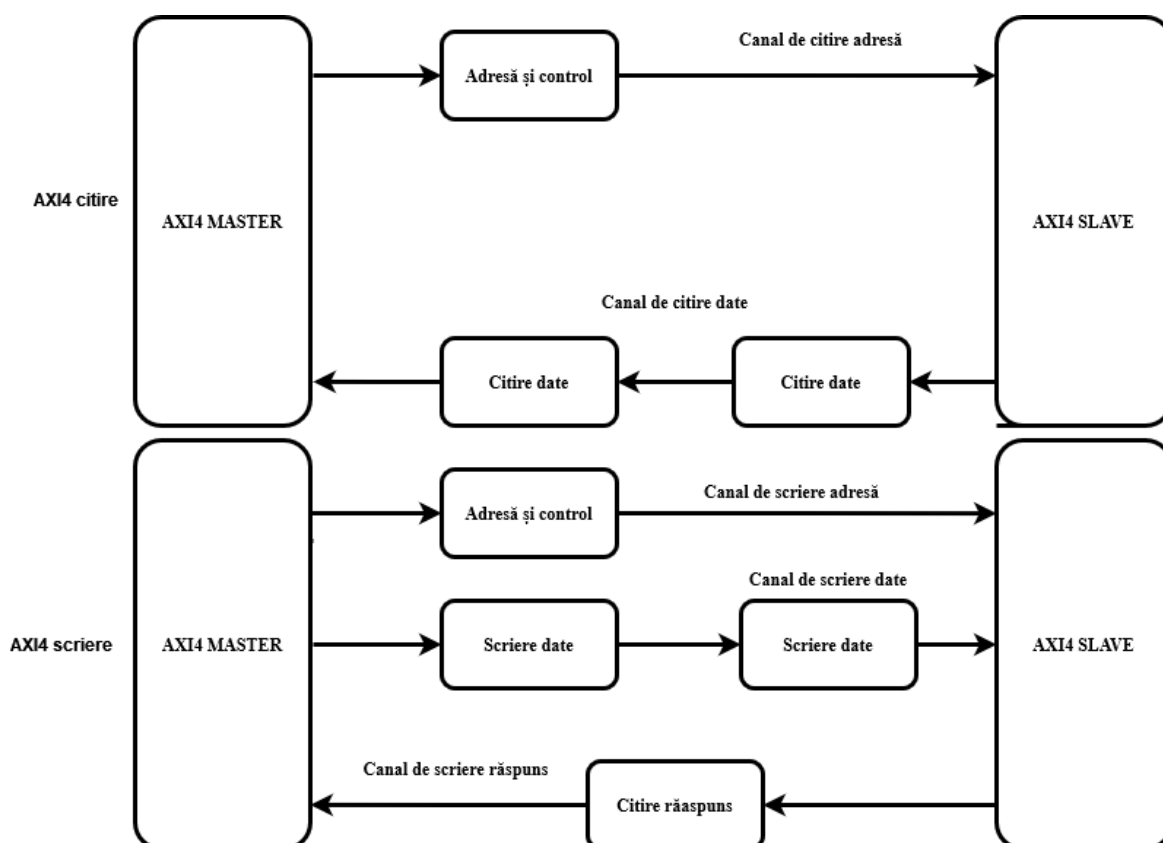
date, este de la 32 până la 64 de biți și folosește metoda single address/single data. Pentru o mai bună înțelegere definim vocabularul pentru o interfață de comunicare

- Channel - Semnale AXI asociate unui semnal VALID.
- Interface - Colecție de una sau mai multe
- Channel- care realizează conexiunea master slave a unui IP core.
- Bus- Un semnal de mai mulți biți.
- Transfer - O perioadă de ceas în care este transmisă informația.
- Transaction- Toată informația transmisă de-a lungul unui Channel.
- Burst- Transaction care sunt făcute cu unu sau mai multe Transfer.



Figură 10. AXI4 Semnale (Sursă: SlidePlayer)

- 1-Master validează și menține VALID cât timp date se pot transfera.
- 2-Slave validează READY dacă dorește să accepte.
- 3-Transfer de date când VALID și READY este în 1.
- 4-Master transmite data/alte semnale sau invalidează VALID.
- 5-Slave invalidează READY dacă nu mai acceptă date.



Figură 11. Citire și Scriere AXI4Lite

4.9 SSM2603 Codec Audio de mică putere

SSM2603 este un codec audio de mică putere, de calitate înaltă pentru aplicații audio care au în componența lor un PGA (stereo programmable gain amplifier) și un microfon. Acesta conține două 24-bit ADC (analog to digital converter) și două 24-bit DAC (digital to analog convertor). Poate lucra atât ca și master cât și slave, suportă o varietate mare de frecvențe de clock precum 12MHz sau 24MHz pentru device-uri conectate prin USB, și poate procesa un număr mare de eșantioane audio precum 96KHz, 88.2 KHz, 48KHz, 44.1KHz, 32KHz , 24KHz, 22.05KHz, 16KHz, 12KHz, 11.025KHz și 8KHz.

Funcționează la tensiuni joase până la 1.8V pentru circuite analogice și 1.5V pentru circuite digitale. Tensiunea maximă admisă este de 3.6V.

ADC-ul primește semnal audio analog fie de la stereo line input sau de la microfon. De reținut că se poate primi informații doar de la o singură intrare, așa că utilizatorul trebuie să aleagă una dintre aceste intrări folosind INSEL bit (Registrul R4, Bit D2). Informația odată convertită este apoi preluată de filtrele ADC-ului. Complementar la cele două canale de ADC, sunt două canale DAC care convertesc informația digital audio din filtrele interne ale DAC-ului într-un semnal analog audio. Canalele de DAC pot să fie oprite folosind DACMU bit (Register 5, Bit D3) din lista registrilor de control.

Atât DAC cât și ADC folosesc filtre digitale asupra semnalului de 24 de biți. Acestea sunt folosite atât pentru record cât și pentru play-back. Pentru record, informația neprocesată de la ADC intră în filtrul ADC-ului și este convertită la frecvența aleasă, apoi este trimisă la interfața digitală audio. Pentru play-back, filtrele de la DAC convertesc informația din interfața digitală audio în informație supra-amorsată, folosind o frecvență de eșantionare selectată de către utilizator. Informația supra-amorsată este apoi procesată de către DAC și trimisă la ieșirea analog prin activarea DACSEL bit (Register R4, Bit D4).

Utilizatorul are posibilitatea de a selecta dacă orice offset găsit să fie rejectat sau nu. Pentru acest lucru, trebuie activat filtrul trece sus prin ADCHPF bit (Register R5, Bit D0).

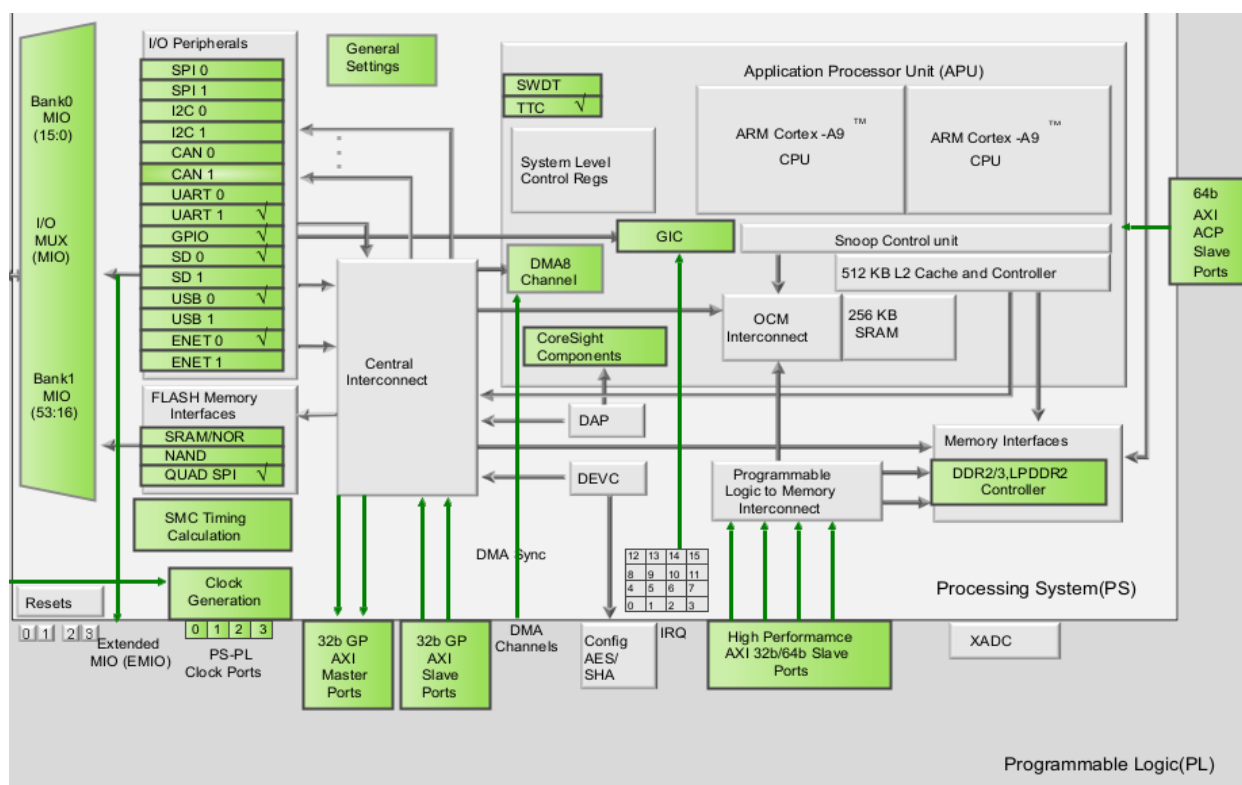
Reg.	Address	Name	D8	D7	D6	D5	D4	D3	D2	D1	D0	Default	
R0	0x00	Left-channel ADC input volume	LRINBOTH	LINMUTE	0	LINVOL[5:0]						010010111	
R1	0x01	Right-channel ADC input volume	RLINBOTH	RINMUTE	0	RINVOL[5:0]						010010111	
R2	0x02	Left-channel DAC volume	LRHPBOTH	0	LHPVOL[6:0]						001111001		
R3	0x03	Right-channel DAC volume	RLHPBOTH	0	RHPVOL[6:0]						001111001		
R4	0x04	Analog audio path	0	SIDETONE_ATT[1:0]		SIDETONE_EN	DACSEL	Bypass		INSEL	MUTEMIC	MICBOOST	000001010
R5	0x05	Digital audio path	0	0	0	0	HPOR	DACMU	DEEMPH[1:0]		ADCHPF	000001000	
R6	0x06	Power management	0	PWROFF	CLKOUT	OSC	Out	DAC	ADC	MIC	LINEIN	010011111	
R7	0x07	Digital audio I/F	0	BCLKINV	MS	LRSWAP	LRP	WL[1:0]		Format[1:0]		000001010	
R8	0x08	Sampling rate	0	CLKODIV2	CLKDIV2	SR[3:0]				BOSR	USB	000000000	
R9	0x09	Active	0	0	0	0	0	0	0	0	Active	000000000	
R15	0x0F	Software reset	Reset[8:0]									000000000	
R16	0x10	ALC Control 1	ALCSEL[1:0]		MAXGAIN[2:0]			ALCL[3:0]					001111011
R17	0x11	ALC Control 2	0	DCY[3:0]			ATK[3:0]					000110010	
R18	0x12	Noise gate	0	NGTH[4:0]					NGG[1:0]		NGAT		000000000

Figură 12. SSM2603 regiștrii (Sursă: SSM2603 Datasheet)

5 Implementarea soluției adaptate

5.1 Implementare Hardware

5.1.1 Zynq7 processing System



Figură 13. Zynq7 Processing System Diagram (Sursă: Vivado)

Ca și sistem de procesare s-a folosit Xilinx LogiCORE IP Processing System 7 care este interfața software creată în jurul Zynq-7000. Sistemul de procesare se comportă ca o conectare logică între PS (processing system) și PL (Programmable Logic) în timp ce integrează componentele din diagrama block, și anume IP Core-uri pentru comunicare și procesare.

Processing System 7 wrapper instanțiază secțiune PS pentru PL și logica externă a plăcuței Zynq, precum și leagă semnalele cu restul circuitului embedded în PL. Interfața dintre PL și PS constă în mare parte din trei mari grupuri: EMIO (Extended multiplexed I/O), programmable logic I/O și AXI I/O. Sunt activate un număr mare de componente din multitudinea oferită de către Zynq, acestea se pot vedea în figura [Figura 9]. Ca și pinuri active pentru periferice sunt folosite UART1 (MIO 48-49), GPIO care cuprinde GPIO MIO, ENET Reset, USB Reset, I2C Reset, apoi este activ SD0 care cuprinde CD și WP, apoi USB 0 ce conține MIO 28 până la 39 și ultimul este ENET 0 (MIO 16-27), se conectează către blocul component ce conține Bank0, Bank1 și MUX-uri.

Setările generale sunt predefinite în Zynq și au fost lăsate standard, Baud Rate pentru UART1 este de 115200Hz și frecvența dată de către Zynq este de 100MHz.

Semnale de ceas diferă pentru componentele active din Zynq, se folosesc 650kHz pentru CPU, 525kHz pentru DDR iar perifericele folosesc o frecvență cu un range între 50 și 200kHz.

Pentru conectarea prin interfața de comunicare AXI4 sunt activate blocuri componente, High Performance AXI 32b/64b Slave Ports care preiau informația și o transmit din PS în PL, sunt activate porturile de interfață S AXI HP0 care au o performanță ridicată. Pentru citire 32b GP AXI Master Ports și pentru scriere 32b GP AXI Slave Ports, acestea comunică cu Central Interconnect. Este activ un block de slave 64b AXI ACP Slave Ports care comunică direct cu APU (Application Processor Unit).

Memoria folosită este DDR3 cu un Data Width de 32 de biți, o frecvență de 525kHz (aceasta poate să fie selectată între 200kHz și 534kHz), și un burst length de 8 biți care reprezintă numărul minim de data beats pe care controlerul îl poate folosi. Pentru memoria Flash internă se folosește interfața SPI de tip Single SS 4-bit I/O.

În interiorul APU (Application Processor Unit) avem componenta GIC care gestionează întreruperile de la PL spre PS și vice versa, este conectată la porturile perifericelor noastre. Este activat DM8 Channel care este un bloc component de interfațare cu DMA. CoreSight Component folosit pentru test și Debug.

APU folosește două CPU ARM Cortex-A9, are 256KB memorie SRAM.

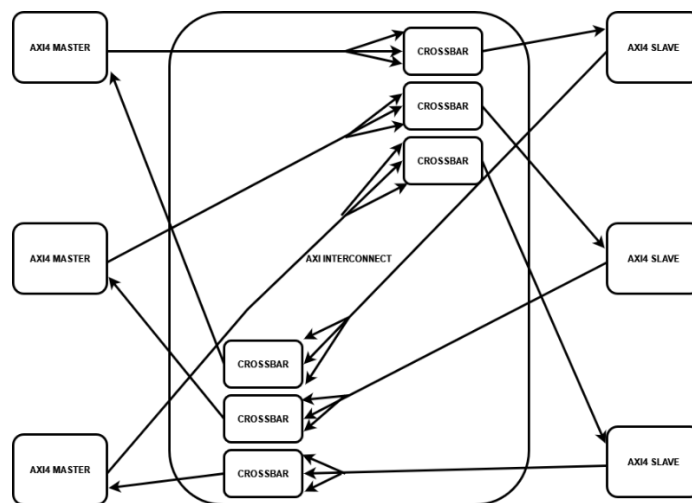
La ieșirea core-ului regăsim pinul DDR care facilitează comunicarea cu memoria DDR, M_AXI_GP0, semnal de tip master pentru comunicarea AXI4Lite, FCLK_RESET0_N care este semnal de reset de la PS spre PL, FCLK_CLK0 care operează partea PL la o frecvență de 100kHz. Ca și semnal de input s-a folosit IRQ_F2P care controlează întreruperile sistemului.

5.1.2 AXI Direct Memory Access - Memoria de acces directă

Pentru a putea stoca datele s-a introdus un core de memorie, The Xilinx LogiCORE IP AXI Direct Memory Acces (AXI DMA). AXI DMA vine cu o viteză mare de accesare a memoriei între memoria AXI4 și interfața AXI4 IP. Ca și semnale de intrare sunt folosite semnalele de clock `s_axi_lite_aclk`, `m_axi_mm2s_aclk`, `m_axi_s2mm_aclk` care facilitează conectarea prin interfața AXI4 și acest core are nevoie de un semnal de reset `axi_resetn`. Datele transmise de către M00_AXI sunt preluate apoi de către portul I/O S_AXI_LITE (folosit pentru control) și stocate. Pinul S_AXI_S2MM este conectat la pinul de I/O AXI_S2MM al core-ului de audio care transmite date sub format serial și o scrie într-o locație de memorie în core-ul dma.

Blocurile de tip AXI DMA IP lasă diferite echipamente periferice ce folosesc interfața de comunicare AXI să acceseze memoria principală prin transfer de tip stream-to-memory sau memory-to-stream. Se poate vedea că S_AXIS_S2MM este portul AXI pentru scriere date în memorie și M_AXIS_MM2S este portul pentru citire date din memorie. Unde S2MM stă pentru stream-to-memory-map și MM2S stă pentru memory-map-to-stream.

5.1.3 AXI Interconnect



Figură 14. Structura AXI Interconnect

În sistemul nostru se folosesc două astfel de core-uri, scopul lor este de a conecta blocurile IP prin interfața AXI4Lite la softul Vitis unde s-a realizat funcționarea lor.

Un core funcționează după semnale de ceas și face conexiunea dintre memoria DMA și portul de intrare al sistemului de procesare Zynq prin interfața AXI4Lite, acesta folosește un număr de 2 porturi slave și un singur port de master. Cele două porturi de slave sunt folosite pentru a prelua informația de la core-ul dma și cel de master este folosit pentru a transmite mai departe către Zynq. Cel de al doilea core este folosit pentru a conecta core-urile de audio, iic și dma la protocolul de comunicare AXI4, acesta folosește un număr de 5 porturi de master și un port de slave, din cele până la 16 care se pot folosi atât pentru slave cât și pentru master. Porturile de master sunt folosite pentru a putea comunica cu memoria dma, blocul audio, IIC iar cel de slave preia informația direct de la Zynq.

AXI Interconnect are un predefinit interval pentru banda care suportă valorile 32, 64, 128, 256 512 și 1024 bits. Toate canalele de date care trec prin Crossbar sunt modificate la una dintre valorile native de bandă pentru AXI Interconnect, pentru primul core se folosește o bandă de 64 de biți iar cel de al doilea folosește o bandă de 32 de biți. Dacă orice canal de date are mărime diferite, AXI Interconnect introduce aceste core-uri de modificare a bandei înainte de a putea ajunge la crossbar.

Conversia bandei depinde de funcționalitate și anume, se poate dori ca banda să fie mai lată atunci când se face transmiterea de la SI la MI și atunci numim upsizing sau avem nevoie de o bandă mai mică și definim downsizing.

În cazul circuitului se dorește un upsizing. Se folosesc două astfel de core-uri. Când banda de date este mai largă în partea de MI decât cea de SI, atunci upsizing este aplicat. Ca și rezultat al tranzacției în partea de MI, numărul de biți este redus.

5.1.4 Processor System Reset

S-au folosit două semnale de ieșire și anume `interconnect_aresetn` care ne dă un reset pe frontul descrescător pentru Interconnect, și `peripheral_aresetn` care ne dă un reset pe frontul descrescător pentru periferice. Semnalele de intrare `slowest_sync_clk` este conectat la cel mai lent semnal de ceas din sistem (`FCLK_RESET0_N`) cu o frecvență de 100MHz și `ext_reset_in` signal folosit pentru a reseta core-ul.

5.1.5 AXI IIC

AXI IIC dă o interfață serială conectată prin două semnale, de viteză redusă, de bus serial care este folosită pentru a comunica cu procesorul Zynq prin semnalul `IIC2INTC_Irpt` care este semnalul de output dând întreruperi. Acest core funcționează prin semnalele de reset și de ceas date de către core-ul Power system reset.

Protocolul I2C este un protocol de comunicație serială sincron, multi-master și multi-slave care conține următoarele semnale, SDA care este linia de date și SCL care este semnalul de ceas.

5.1.6 Audio Core

Core-ul de audio facilitează procesarea semnalului audio, acesta este conectat la platforma Vitis prin interfața de comunicare `AXIS4Lite` folosind atât semnale de master cât și de slave. Prin pinul `AXI_S2MM` se scrie seria de date și o transmite la portul `S_AXIS_S2MM` care este port de intrare pentru AXI Direct Memory Access, unde este stocată în memoria DMA. Pentru a putea conecta core-ul de audio, a fost nevoie să facem câteva modificări în Zynq, aceste modificări sunt introducerea unui semnal de ceas secundar și să permitem comunicarea core-ului prin protocolul de comunicare I2C.

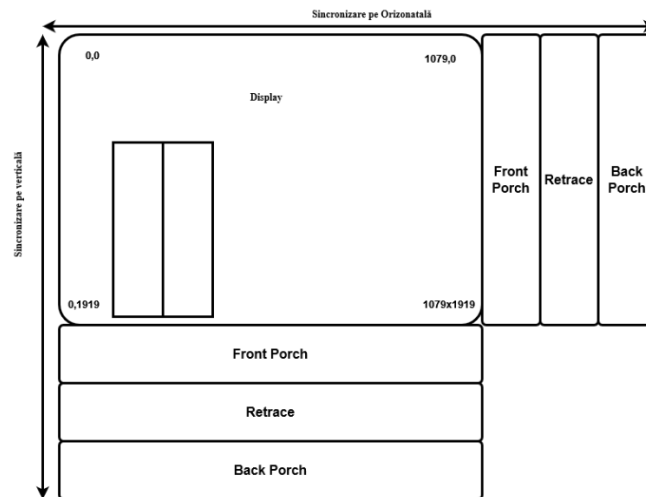
5.1.7 Video

Pentru componenta video s-a realizat proiectarea unor bare verticale pe un ecran cu rezoluția de 1920x1080p. Conexiunea s-a realizat folosind un cablu de conectare de tip VGA. Astfel pentru entitate avem nevoie de semnalele de sincronizare și semnalele de culoare pentru a defini fiecare pixel pe ecran.

Imaginea pe un desktop prin intermediul VGA se realizează prin cele 5 semnale de bază, acestea au fost portate ca fiind porturi de ieșire, `vga_hs` responsabil pentru formarea imaginii pe orizontală și `vga_vs` responsabil pentru formarea imaginii pe verticală. Fiecare pixel va prelua o culoare formată din cele 3 culori fundamentale care sunt portate de către semnalele `vga_r`, `vga_b`, `vga_g`. Tehnologia VGA folosește un număr mai mare de pixeli pentru a putea face afișarea, sunt definite zone de Back Porch, Retrace și Front Porch pentru verticală și pentru orizontală. Zona activă, este zona de Display video unde se pot vedea barele verticale. S-a implementat procese pentru a pune pe 1 registrul de control pe orizontală și verticală, atunci când se află în zone de Retrace și procese pentru a număra până la valoarea maximă a cadranului, când ambele semnale ajung în capătul cadranului se resetează la 0. Rezoluția aleasă este de 1980x1080p, în logica VGA rezoluția reală este mai mare datorită zonelor aferente. Cu ajutorul semnalelor de sincronizare pe verticală și pe orizontală se pot delimita zone în care să se situeze barele pentru a forma un sound

bar. Astfel barele se afla între zonele 2-238, 242-478, 482-718, 722-958, 962-1198, 1202-1438, 1442-1678, 1682-1918.

Asemenea unui sound bar culorile încep de la verde și se termină cu roșu, astfel s-au ales 8 culori care realizează această tranziție. Astfel culoarea verde v-a însemna ca volumul semnalului este la o valoare mică, galbenă înseamnă că volumul va fi la o valoare medie iar roșie înseamnă că volumul va fi la o valoare mare.



Figură 15. Logica VGA

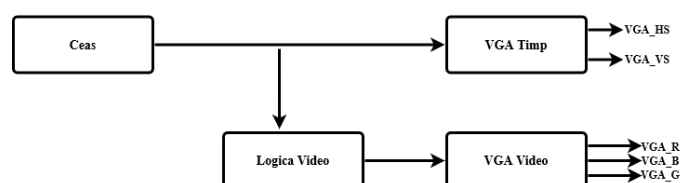
Frecvența necesară de 148.5MHz s-a găsit datorită formulei

$$P * L * S = 148.5 \text{ MHz}$$

- P= pixeli pe linie
- L= linii pe ecran
- S= ecrane pe secundă

(4)

Pentru reprezentare s-a implementat un set de else/if care cuprinde cele 8 baruri. Aceasta descrie poziția lor pe verticală respectiv pe orizontală și pentru fiecare bit s-a implementat două stări, de pornit (1 logic) sau nu este folosit. Biții sunt formați prin combinarea celor 3 culori fundamentale, roșu, verde și albastru. S-a ales culoarea roșu sau verde sau galbenă, când bara este activă (`vga_red <="1111"`), celelalte culori rămânând în negru ("`0000`") și fundalul pentru tub este standard în alb, a fost implementat prin punerea în 1 a tuturor celor 3 culori (`vga_blue <="1111"`).



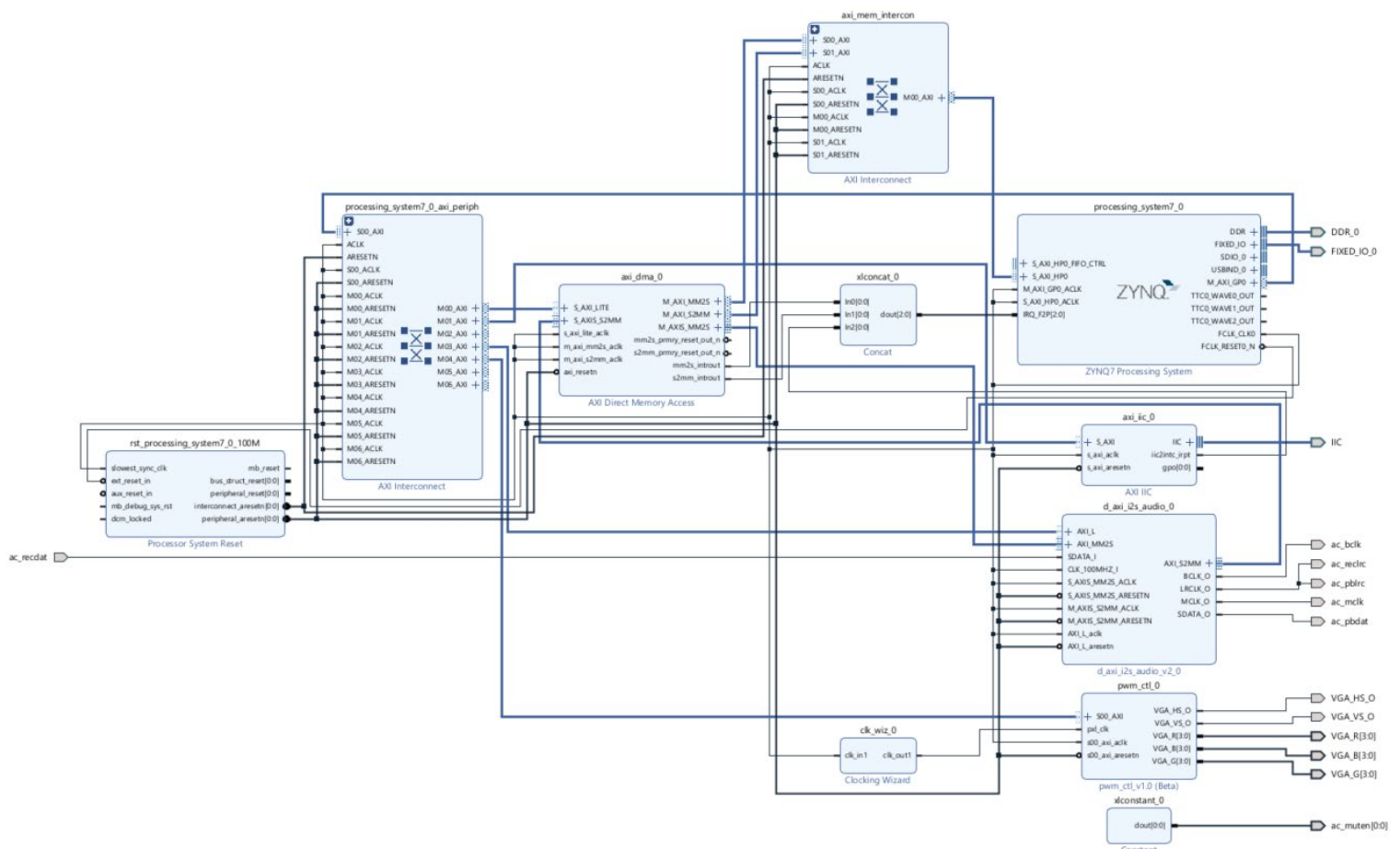
Figură 16. Logică implementare VGA



Figură 17. Sound Bar la volum maxim

În urma implementării în Vivado a blocului component acesta are nevoie de o frecvență de ceas de 148.5MHz care este realizată cu un clock Wizard ce primește frecvența de la core-ul Zynq. Pentru a putea primi date și pentru a putea controla din Vitis, core-ul este conectat prin interfața AXI4 prin blocuri componente și prin stocarea adresei core-ului într-un registru din fișierul „xparameters.h”. Ieșirea core-ului este conectată în xsa-ul Zybo pentru porturile de vga.

5.1.8 Funcționare Implementare Hardware



Figură 18. Diagrama bloc Vivado

În figura [Figura 18] este prezentată diagrama bloc a circuitului realizată în Vivado 2022.1. Pentru aceasta au fost importate mai multe IP-uri care împreună realizează funcționarea hardware a circuitului.

Core-ul Processor System Reset este responsabil pentru a da întregului nostru circuit semnale de ceas atât pentru sistemele interne cât și pentru periferice, astfel va aproviziona fiecare core cu numărul necesar de semnale de ceas și de reset.

Constanta are rol de a genera un single-bit constant value, numit dout[0:0]. Semnalul ac_muten este un semnal care controlează statul de mute a semnalul audio. Conectând dout[0:0] la ac_muten[0:0], vom controla starea de mute al audio-ului nostru cu o valoare constantă. De exemplu, dacă constanta ia valoare "0" atunci semnalul ac_muten[0:0] este stopat și va lasă audio să treacă prin output, iar dacă constanta este setată la "1" atunci semnalul audio o să fie stopat.

Core -ul AXI IIC este folosit pentru a comunica prin interfața I2C bus cu alte componente. Când diferite evenimente apar pe bus-ul I2C, AXI IIC core generează o întrerupere de tip semnal ("iic2intc_irpt") care este conectat la core-ul "Concat". Acest core combină acest semnal cu alte semnale de întrerupere, iar semnalul rezultat ("dout[3:0]") este conectat la portul de intrare pentru întreruperi ("IRQ_F2P[3:0]"). Acesta îi dă voie core-ului Zynq7 să răspundă la întreruperile generate de AXI IIC core și la alte periferice din sistem. Intrarea acestui port realizează comunicarea cu Zynq7 Processing System core.

Core-ul de audio este conceput să funcționeze cu interfața IIS (Inter-IC Sound) care este un protocol de comunicare serială folosit pentru audio.

Regăsim următoarele ieșiri:

- AXI_S2MM care este conectat la S_AXI_S2MM port de ieșire al DMA (Direct Memory Access) core. Acest port este folosit pentru a transfera date de la interfața IIS către memorie.
- BLCK_O este un pin de ceas conectat la ac_blen output pin, acesta ne indică datele de timp pe interfața I2S.
- LRCLK_O este un pin LEFT-RIGHT, acesta arată dacă datele curente sunt pentru canalul din stânga sau din dreapta.
- MCLK_O este un pin de master care sincronizează transmiterea de date între transmițător și receptor.
- SDATA_O este un pin serial de date care conține semnalul audio.

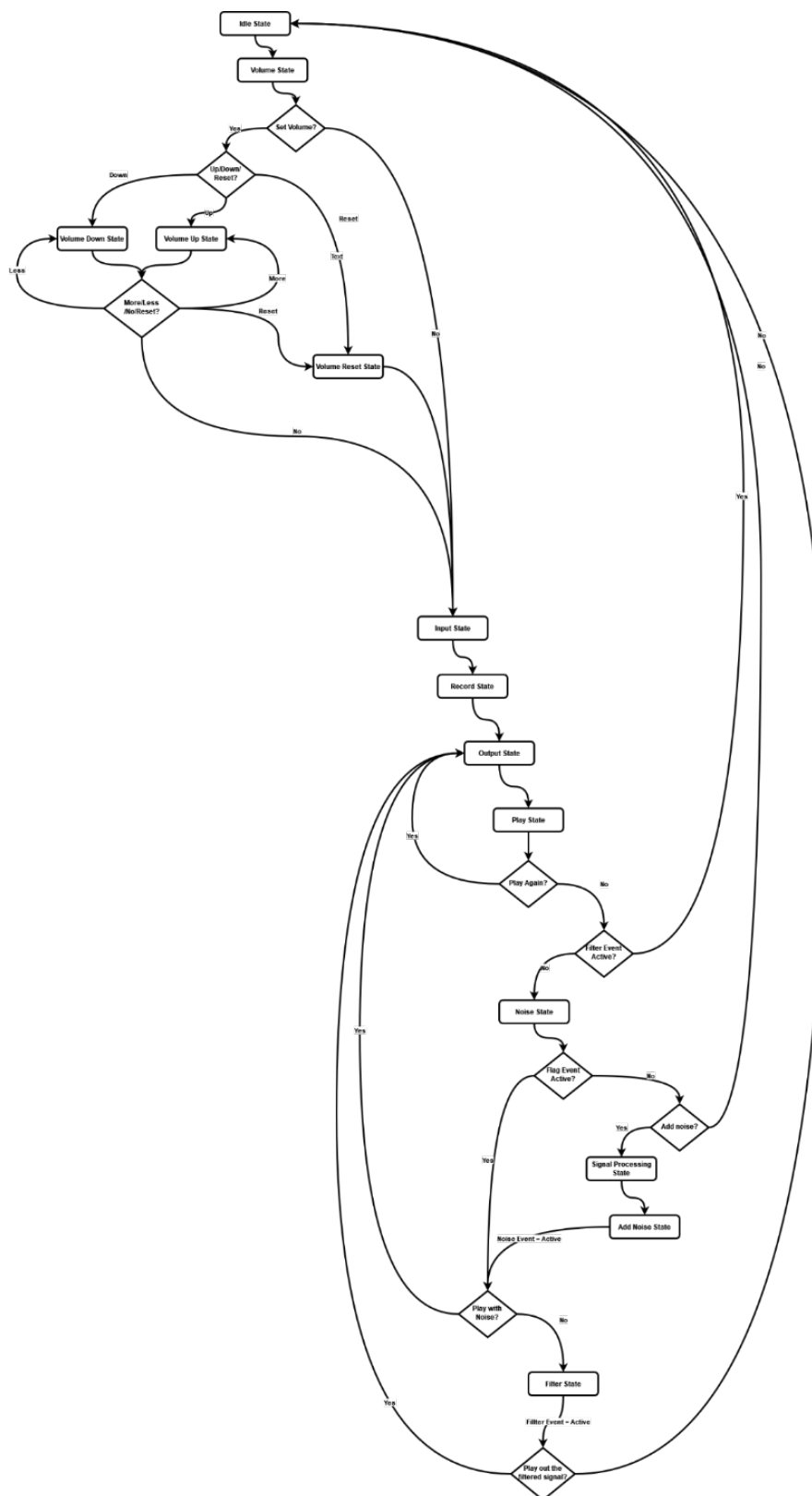
Regăsim următoarele intrări:

- AXI_MM2S conectat la M_AXIS_MM2S output pin de la DMA core, folosit pentru a transfera date de la memorie la interfața I2S.
- AXI_L este folosit pentru a transfera semnale de control, precum reset și enable.
- SDATA_I este un pin de date serial și este conectat la ac_recdat input pin. Acest pin preia date de la interfața I2S.

Pe scurt, core-ul audio este folosit pentru conectarea cu interfața audio IIS și pentru a transfera date către și de la memorie folosind AXI_MM2S și AXI_S2MM port.

După ce s-a realizat toată această implementare hardware, s-a exportat fișierul .bit împreună cu xsa, și ulterior s-a creat aplicația în Vitis SDK pe baza acestei platforme.

5.2 Implementare Software



Figură 19. Diagrama bloc Vitis

În fișierul principal s-a realizat structura schelet a sistemului de control care preia semnalul de la porturile jack de pe placa de dezvoltare Zybo și îl prelucrează după schema bloc care este reprezentată în figura [Figura 19].

Într-un sistem embedded este nevoie de inițializare a driverelor, astfel acestea sunt apelate în prima parte a programului, driverele necesare sunt:

- Inițializarea controlerului de întrerupere.
- Inițializarea controlerului de IIC
- Inițializarea driverului pentru controlul interfațării cu utilizatorul
- Inițializarea DMA
- Inițializare pentru controlerul Audio I2S

Pentru inițializarea controlerului de întreruperi se face trimitere către funcția `fnInitInterruptController` din fișierul `intc.c`, aceasta primește ca și argument un pointer `*psIntc` către instanța de driver. În prima parte a funcției se instanțiază driverul controlerului de întreruperi prin funcția `XIntc_Initialize` care primește două argumente, pointer-ul la structura controler-ului și ID-ul dispozitivului, dacă inițializarea a reușit, va returna `Success` în caz contrar se returnează `Failure`, funcționalitate dată de funcția `RETURN_ON_FAILURE` care înglobează funcția `XIntc_Initialize`. Apoi se pornește controlerul de întreruperi prin funcția `XIntc_Start`, acesta primește doi parametri, pointerul la structura `psIntc` și `XIN_REAL_MODE` care este un macro cu rolul de a verifica valoarea returnată de către `XIntc_Start`. În continuare se inițializează driverul controlerului de întrerupere pentru a putea fi folosit. Asemănător se inițializează fiecare driver din cele enumerate mai sus.

```
XStatus fnInitInterruptController(intc *psIntc)
{
    int result = 0;
#ifdef XPAR_XINTC_NUM_INSTANCES
    RETURN_ON_FAILURE(XIntc_Initialize(psIntc, INTC_DEVICE_ID));
    RETURN_ON_FAILURE(XIntc_Start(psIntc, XIN_REAL_MODE));
    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
        (Xil_ExceptionHandler)XIntc_InterruptHandler,
        psIntc);
#endif
}
```

Pentru a putea filtra toate datele a fost nevoie să alocăm spațiu de memorie dinamic, folosind funcția `malloc`. S-a atribuit unei variabile pointer-ul returnat de către funcție, prin aceasta se calculează dimensiunea necesară pentru a stoca dinamic eșantioanele semnalului audio, s-a ales ca și tip de date `u32` deoarece aceasta are un range de 4,294,967,295.

```
buffer_ = (u32*)malloc(NR_AUDIO_SAMPLES * sizeof(u32));
if (buffer_ == NULL) {
    printf("Memory not allocated for buffer_.\n");
    return 0;
}
```

Diagrama din figură [Figura 19] reprezintă drumul parcurs de către utilizator prin program și este realizată cu o mașină de stare. La începutul structurii ne aflăm în “Idle State”, acesta reprezintă starea default, moment în care se afișează pe terminal meniul. S-a folosit ca și terminal

Tera Term, în acesta se setează portul serial și din meniul intern al terminalului se setează frecvența de 115200Hz.

```
switch(eNextState){
    case Idle_State:{
        xil_printf("%s",logger(Idle_State));
        if(Set_Input_Event == eNewEvent){
            eNextState = SetVolumeHandler();
        }
    }
    break;
```

Următorul pas este “Volum State”, unde suntem întrebați dacă dorim să modificăm volumul semnalului. Acesta poate să fie crescut, scăzut, resetat la valoarea inițială sau nu dorim să schimbăm nimic la semnalul nostru. În terminal se așteaptă valoarea preluată de la UART care fie este “u” pentru a ajunge în state-ul “SetVolumeUpHandler”, “d” pentru a ajunge în state-ul “SetVolumeDownHandle”, „r” pentru a ajunge în „Set Volume Default State/ Volume Default State” sau „n” pentru a nu modifica valoarea, în acest caz se trece direct în „Input State”. Prin Volume Down State și prin Volume Up State putem modifica valoarea volumului de ori de câte ori ne dorim, s-a impus o singură condiție, pentru siguranță, în momentul în care volumul este dat prea tare acesta o să revină la starea de default.

În funcția fnSetLineInputVolumeUp, care se apelează în case-ul “SetVolumeUpHandler”, se încarcă în registrul R0_LEFT_ADC_VOL și în registrul R1_RIGHT_ADC_VOL, valoarea corespunzătoare în funcție de alegerea noastră. Sunt 14 valori posibile, realizate printr-o mașină de stare care cresc volumul până la 33dB. De exemplu, dacă dorim să creștem până la valoarea de 16.5dB, selectăm până ajungem la case-ul 3 și se încarcă în registrii RO/R1 valoarea 0x22 (00100010 în binar). Acești doi regiștrii se regăsesc în structura internă SSM2603 care este un codec audio de putere mică creat de către cei de la Analog Devices și este implementat în structura proiectului nostru. Registrii sunt folosiți ca Left-Channel ADC input volume și Right-Channel ADC input volume. Asemănător pentru funcția fnSetInputVolumeDown.

Pentru starea de reset se apelează funcția fnSetLineInputVolume care scrie în registrii “R0_LEFT_ADC_VOL” și “R1_RIGHT_ADC_VOL” valoarea “0x17” care este valoarea default setată în “SSM2603”.

LEFT-CHANNEL ADC INPUT VOLUME, ADDRESS 0x00

Table 11. Left-Channel ADC Input Volume Register Bit Map

D8	D7	D6	D5	D4	D3	D2	D1	D0
LRINBOTH	LINMUTE	0						LINVOL[5:0]

Table 12. Descriptions of Left-Channel ADC Input Volume Register Bits

Bit Name	Description	Settings
LRINBOTH	Left-to-right line input ADC data load control	0 = disable simultaneous loading of left-channel ADC data to right-channel register (default) 1 = enable simultaneous loading of left-channel ADC data to right-channel register
LINMUTE	Left-channel input mute	0 = disable mute 1 = enable mute on data path to ADC (default)
LINVOL[5:0]	Left-channel PGA volume control	00 0000 = -34.5 dB ... In 1.5 dB steps 01 0111 = 0 dB (default) ... In 1.5 dB steps 01 1111 = 12 dB 10 0000 = 13.5 dB 10 0001 = 15 dB 10 0010 = 16.5 dB 10 0011 = 18 dB 10 0100 = 19.5 dB 10 0101 = 21 dB 10 0110 = 22.5 dB 10 0111 = 24 dB 10 1000 = 25.5 dB 10 1001 = 27 dB 10 1010 = 28.5 dB 10 1011 = 30 dB 10 1100 = 31.5 dB 10 1101 to 11 1111 = 33 dB

Figură 20. R0 SSM2603 (Sursă: SSM2603 Datasheet)

Din acest punct se face trecerea în “Input State”, dacă nu se dorește modificarea volumului se poate trece direct în “Input State”. În acest moment se inițializează cele 24 de ADC-uri și 24 de DAC-uri ale codec-ului de audio pentru a putea face înregistrare și redare, iar apoi se trece în “Record State”.

În state-ul de Record se înregistrează o nouă secvență audio cu durată de 3 secunde, iar în fiecare secundă se înregistrează un număr de 96000 de eșantioane din semnalul înregistrat. Semnalul audio este înregistrat prin porturile jack de pe placă. Astfel, conectăm placa Zybo la portul jack de la laptop sau Pc și redăm o secvență la alegere. Poate să fie generat un semnal folosind un program specific sau fie reda o secvență deja existentă. Trebuie ca și cablul jack să fie conectat la portul Line In de pe placă, o altă variantă este de a conecta portul jack la Mic In de pe placă și să folosim un microfon pentru a înregistra un semnal. Iar pentru a reda acest semnal, se conectează un set de boxe la portul HPH Out. Când înregistrăm secvența audio este important să setăm volumul din laptop sau calculator la o valoare recomandată între 0 și 35%, de precizat că se înregistrează și zgomotul și atunci nu o să avem afișat zero în caz real.

După se face trecerea în Output state care prin funcția fnSetHpOutput se apelează funcția fnAudioWriteReg, aceasta scrie în registrul R4_ANALOG_PATH valoarea 0b000010110 și în registrul R5_DIGITAL_PATH valoarea 0b000000000.

Valoarea 0b000010110 este folosită în stadiul de record pentru a selecta microfonul ca și input, funcționalitate dată de bitul INSEL (Registrul R4, Bit D2), informația receptată este luată de ADC, apoi după ce este convertită, este preluată de filtrele ADC-ului unde este convertită la frecvența aleasă de către utilizator și trimisă la interfața digitală audio. Pentru play-back, filtrele de la DAC convertesc informația din interfața digitală audio în informație supra-amorsată, folosind o frecvență de eșantionare selectată de către utilizator. Informația supra-amorsată este apoi procesată de către DAC și trimisă la ieșirea analog prin activarea bitului DACSEL (Register R4, Bit D4). Iar prin registrul R5 se activează cele 24 de DAC-uri.

ANALOG AUDIO PATH, ADDRESS 0x04

Table 19. Analog Audio Path Register Bit Map

D8	D7	D6	D5	D4	D3	D2	D1	D0
0	SIDETONE_ATT[1:0]	SIDETONE_EN	DACSEL	Bypass	INSEL	MUTEMIC	MICBOOST	

Table 20. Descriptions of Analog Audio Path Register Bits

Bit Name	Description	Settings
SIDETONE_ATT[1:0]	Microphone sidetone gain control.	00 = -6 dB (default) 01 = -9 dB 10 = -12 dB 11 = -15 dB
SIDETONE_EN	Sidetone enable. Allows attenuated microphone signal to be mixed at device output terminal.	0 = sidetone disable (default) 1 = sidetone enable
DACSEL	DAC select. Allows DAC output to be mixed at device output terminal.	0 = do not select DAC (default) 1 = select DAC
Bypass	Bypass select. Allows line input signal to be mixed at device output terminal.	0 = bypass disable 1 = bypass enable (default)
INSEL	Line input or microphone input select to ADC.	0 = line input select to ADC (default) 1 = microphone input select to ADC
MUTEMIC	Microphone mute control to ADC.	0 = mute on data path to ADC disable 1 = mute on data path to ADC enable (default)
MICBOOST	Primary microphone amplifier gain booster control.	0 = 0 dB (default) 1 = 20 dB

Figură 21. R4 SSM2603 (Sursă: SSM2603 Datasheet)

DIGITAL AUDIO PATH, ADDRESS 0x05

Table 21. Digital Audio Path Register Bit Map

D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	HPOR	DACMU	DEEMPH[1:0]	ADCHPF	

Table 22. Descriptions of Digital Audio Path Register Bits

Bit Name	Description	Settings
HPOR	Stores dc offset when high-pass filter is disabled	0 = clear offset (default) 1 = store offset
DACMU	DAC digital mute	0 = no mute (signal active) 1 = mute (default)
DEEMPH[1:0]	De-emphasis control	00 = no de-emphasis (default) 01 = 32 kHz sampling rate 10 = 44.1 kHz sampling rate 11 = 48 kHz sampling rate
ADCHPF	ADC high-pass filter control	0 = ADC high-pass filter enable (default) 1 = ADC high-pass filter disable

Figură 22. R5 SSM2603 (Sursă: SSM2603 Datasheet)

Din starea de “Output State” se trece în “Play State” unde se așteaptă comanda de la UART prin terminal, se citește valoarea primită. Suntem întrebați dacă ne dorim să redăm din nou ce avem stocat, dacă dorim acest lucru se introduce “y” în terminal, și se face trecerea în “Output State”, dacă valoarea introdusă este diferită se verifică dacă flag-ul pentru filtru este activ, care ne spune dacă avem deja un filtru adăugat, dacă nu este activ se face trecerea în “Noise State”. Prin funcțiile apelate în Play State se realizează filtrarea semnalului pentru a putea fii citit de către componenta video din Vivado.

În interiorul funcției de Audio Play se face alocarea spațiului în Heap prin funcția malloc. În interiorul acestei funcții se apelează o altă funcție care realizează medierea semnalului audio. Semnalul audio este mediat cu 300 de eșantioane la stânga și 300 de eșantioane la dreapta, pentru fiecare eșantion în parte (aceste valori pot să fie ajustate), prin acest model ne dorim să ajungem la o valoare constantă, deoarece și volumul este o valoare constantă. Ulterior valoarea calculată trebuie împărțită cu o constantă step. Această constantă step determină cât de multe bare pe ecran corespund pentru valoarea mediată. După împărțire valoarea rezultată este comparată într-un switch/case de 8 de valori, acestea pornind de la 0x01 până la 0xff, acestea sunt valorile trimise către componenta de video din Vivado. (0x00 corespunde pentru 0000 0001 și 0xff corespunde pentru 1111 1111).

În “Noise State” se verifică flag-ul pentru noise, pentru activ, suntem întrebați dacă ne dorim să redăm semnalul cu zgomot, dacă răspunsul este da atunci introducem în terminal “y” și ne trimite înapoi în “Output State” de unde se poate face redare. Dacă nu este activ flag-ul pentru zgomot atunci suntem întrebați dacă ne dorim să adăugăm zgomot, dacă răspunsul este da atunci se merge în “Signal Processing State”, în caz contrar se trece în starea de “Idle”.

În “Signal Processing State”, se poate afișa semnalul nostru original, împreună cu semnalul la care s-a adăugat zgomot. Semnalul cu zgomot s-a realizat prin apelul funcției `signal_processing`.

În interiorul funcției `signal_processing` se convertesc datele. Pentru a putea prelucra datele primite acestea trebuie convertite din reprezentarea complement față de 2 în reprezentare binară și apoi în integer. Acest lucru se realizează prin compararea fiecărui eșantion original cu 1 prin aplicarea unei porți AND, dacă comparația este validă se încarcă 1 în matrice, dacă nu, se încarcă 0. Apoi se verifică dacă masca 0x400000 (bit de semn) este activă, prin aplicarea unei operații AND între masca și eșantionul original, dacă această mască este activă, se parcurge șirul de valori și pentru toate pozițiile mai mari sau egale de 24 se încarcă în matricea precedentă valoarea 0 pe toate pozițiile echivalente. Iar pentru celelalte valori se inversează biții. Valoarea încărcată în matrice este transformată cu ajutorul unei funcții în format întreg. Acest lucru se realizează prin înmulțirea fiecărei valori binare cu 2 la puterea corespunzătoare și rezultatul stocat într-o variabilă finală. În final este verificată masca 0x800000, tot prin aplicarea operației AND asupra eșantioanelor originale, dacă aceasta este activă se aplică extensia de semn prin înmulțirea valorii din buffer cu -1, prin aceasta se verifică dacă numărul este negativ. Toata aceasta logică a fost implementată deoarece semnalul înregistrat nu recunoaște valori negative, și atunci acesta trebuie reconstruit din valorile primite. Diferența dintre valorile pozitive și valorile negative este foarte mare, de aproximativ 16000000, și atunci se impune condiție de a prelua și modifica doar valori pozitive.

În continuare se afișează eșantioane originale, la aceste eșantioane se adaugă un semnal sinusoidal. Acest semnal are o amplitudine de 10 ori mai mare decât semnalul original pentru a evidenția ulterior suprapunerea.

```
for (i=0; i<u32NrSamples; i++)
{
    t = (double)i/(double)u32SamplingFreq;
    sample = ((double)u32Amplitude * sin(2 * 3.14 * (double)u32SinFreq * t));
    *(buffer + i) += (int)sample;
    *(puSoundIn + i) = *(buffer + i) ;
};
```

Pentru a putea evidenția într-un singur cadran ambele semnale, s-a adăugat o linie de 30 de valori de 0, care face tranziția dintre semnalul original și semnalul alterat prin zgomot. Apoi se afișează semnalul asupra căruia s-a adăugat sinusul.

În continuare se trece în “Add Noise State”, unde se verifică flagul de zgomot, dacă acesta este activ, suntem întrebați dacă dorim să redăm semnalul cu zgomot, dacă da, se revine în “Output State”, iar dacă nu, se trece în “Filter State”.

“Filter state”, unde se aplică filtrul asupra semnalului. În interiorul funcției `filter_data` este conceput un filtru EMA (Exponential Moving Average). Acest tip de filtru este foarte folosit de către cei ce vând și cumpără acțiuni la bursă, crypto sau a altor componente ale căror valoare se modifică constant, și se dorește să se afle o medie a valorilor, sub forma unei linii pe durata a unui

timp. Asemănător acestui tip de filtru, mai există și filtrul SMA (Simple Moving Average) în care toate valorile pentru un anumit timp sunt tratate egal. Prin filtru EMA se calculează media valorilor unui anumit număr de eșantioane consecutive din semnalul nostru, asupra căruia s-a adăugat zgomot și folosește media calculată pentru a obține o valoare filtrată pentru fiecare eșantion în parte, se observă că valorile mai recente sunt mai importante decât cele precedente, semnalul fiind astfel mult mai ușor de interpretat și de prelucrat pentru viitoarele aplicații.

Pentru a realiza un filtru EMA, este nevoie să parcurgem câțiva pași, în primul rând se aleg numărul de eșantioane pentru care o să se aplice filtrul, am ales 1500. Pentru primul eșantion se va calcula EMA inițială, care va da startul pentru valorile următoare. Apoi se calculează factorul de pondere, acesta s-a numit Alpha și este o valoare cuprinsă între 0 și 1, acesta determină cât de mult să se atribuie pondere valorilor recente în comparație cu cele vechi. Valoarea Alpha s-a calculat print-un filtru trece jos. Apoi se calculează EMA pentru fiecare eșantion după următoarea formulă:

$$EMA = EMA_anterior + factor_de_pondere * (valoarea\ curen\ ta - EMA_anterior). \quad (5)$$

Componentele unui filtru trece jos au fost declarate astfel, frecvența de tăiere (CUTOFF) să fie 500, RC este $1/(CUTOFF * 2 * \pi)$, variabila de timp dt este 1 supra rata de eșantionare, variabila Alpha o să fie subunitară, aceasta reprezintă factorul de netezire. Prin această valoare se determină influența eșantionului curent asupra eșantionului final, o valoare mai mică duce la un semnal mai neted, cu răspuns mai mare, iar o valoare mai mare a lui Alpha duce la un semnal mai puțin neted care urmărește semnalul de intrare mai îndeaproape.

```
int CUTOFF = 500;
u32 filter_data(u32 *puSoundInRAW,u32 *puSoundInProcessed,u32 u32NrSamples){
    int j;
    u32 i;
    float RC = 1.0/(CUTOFF*2*3.14);
    float dt = 1.0/AUDIO_SAMPLING_RATE;
    float alpha = dt/(RC+dt);
    float filteredArray[number_of_sample_to_process];
    filteredArray[0] = puSoundInProcessed[0];
    j=u32NrSamples;
    while(--j){
        xil_printf("%d\n\r",*(puSoundInProcessed+j));
    }
    for(i=0;i<30;i++){
        printf("%d\n\r",0x00);
    }
    for(i=1;i<number_of_sample_to_process;i++){
        filteredArray[i] = filteredArray[i-1] + (alpha*(*(puSoundInProcessed+i) - filteredArray[i-1]));
        printf("%f\n\r",*(filteredArray+i));
    }
}
```


6 Rezultate experimentale

Pentru a putea demonstra funcționalitatea programului, avem nevoie de un monitor care să permită o rezoluție de 1980x1080, un cablu VGA care o să fie conectat la placa de dezvoltare Zybo și la monitorul aferent, de un cablu USB și cu un cap micro-usb conectat la placă, prin care realizăm programarea plăcii, cablu de alimentare, iar pentru a putea reda și stoca semnalul audio avem nevoie de un cablu jack conectat la portul HPH OUT pe placă, acesta să fie cablul unui sistem de redare audio, precum un set boxe sau căști, care vor fi folosite pentru a reda secvența audio înregistrată și cea la care am adăugat zgomot, și un cablu cu două intrări jack, unul conectat la portul LINE IN de pe placă și unul conectat la laptop, se poate folosi și un microfon, utilitatea acestuia fiind de a înregistra o secvență audio fie din PC/Laptop sau din mediul ambiant.

Ca și semnal se poate folosi orice secvență audio, am ales un site de specialitate, Online Tone Generator (<https://onlinetonegenerator.com/>) unde se pot genera semnale cu frecvență specifică, sau tipuri diferite de semnale precum sinus, triunghi, dreptunghi sau dinte de fierăstrău.

Următorul pas este de a încărca programul pe placă, din meniul din Vitis SDK se rulează programul. În terminalul TeraTerm se setează portul COM 6 (sau portul la care este conectată placa, denumirea poate să difere) și viteza de 115200 din meniul intern al terminalului.

În momentul în care s-a realizat conectarea, se aude în boxe un ton și se afișează în terminal meniul. La începutul programului, nu avem nimic înregistrat și ne aflăm în Idle State, așa cum se poate vedea în figura [23], apoi se trece în Volume State, de unde se poate seta volumul până la valoarea maxima de 33dB sau îl putem scădea. Alte variante sunt de a reseta volumul și atunci acesta este setat la starea inițială sau de a nu îl modifica și se trece în Input State.



```
--- Entering main() ---  
Idle State...  
Volume State...  
Set volume? <u-up/d-down/r-reset/n-no>
```

Figură 23. Tera Term Meniu

Ne dorim să creștem volumul, se poate vedea în figura [24], introducem „u” de la tastatură, ne este precizat la ce valoare în dB este volumul momentan, și suntem întrebați dacă dorim să-l creștem sau să revenim la valoarea inițială. Dacă am modificat volumul avem posibilitatea să îl resetăm din acest punct la valoarea sa inițială, prin introducerea „r” în terminal, acesta ne va trimite în Volume Reset State. Doresc să cresc volumul de două ori, astfel introduc de la tastatură de două ori „y” și se setează la 16dB.


```

Volume set to 0dB
Increase the volume or go to the default value?(y/n/r)r->this will jump to the reset state
Set Volume Up State...

Volume set to 12dB
Increase the volume or go to the default value?(y/n/r)r->this will jump to the reset state
Set Volume Up State...

Volume set to 16dB
Increase the volume or go to the default value?(y/n/r)r->this will jump to the reset state

```

Figură 24. Volume Up Meniu

După ce am crescut volumul la 16dB, introducem „n” de la tastatură și suntem trimiși în Input State, apoi Record State, apoi Output State și în urmă în Play State, aceste etape se pot vedea în figura 25. În aceste momente se înregistrează secvența audio și se redă automat în boxe sau în căști, la final suntem întrebați dacă dorim să redam din nou, dacă dorim acest lucru introducem „y” de la tastatură, dacă nu, introducem „n” și mergem mai departe în program. De menționat că secvența audio pe care dorim să o înregistrăm trebuie să fie redată în tip real dacă ne aflăm în această etapă.

```

Set Record State...

Recording Done...
Output State..

Play State...

Play again?(y/n)

```

Figură 25. Record/Play Meniu

Mergem mai departe în program, se verifică flag-ul de zgomot, dacă acesta nu este activat suntem întrebați dacă ne dorim să adăugăm zgomot. Cum suntem la prima interacțiune cu programul, nu vom avea zgomot deja înregistrat și atunci îl adăugăm noi. Se merge mai departe în Signal Processing State și Add Noise State, se pot vedea în figura [26]. În această etapă se pot afișa 500 de eșantioane din semnalul original împreună cu 500 de eșantioane din semnalul alterat prin zgomot, între aceste două secvențe s-a introdus o serie de 30 de valori de 0 pentru a distinge trecerea între cele două semnale.

```

Playback Done...
Noise State...

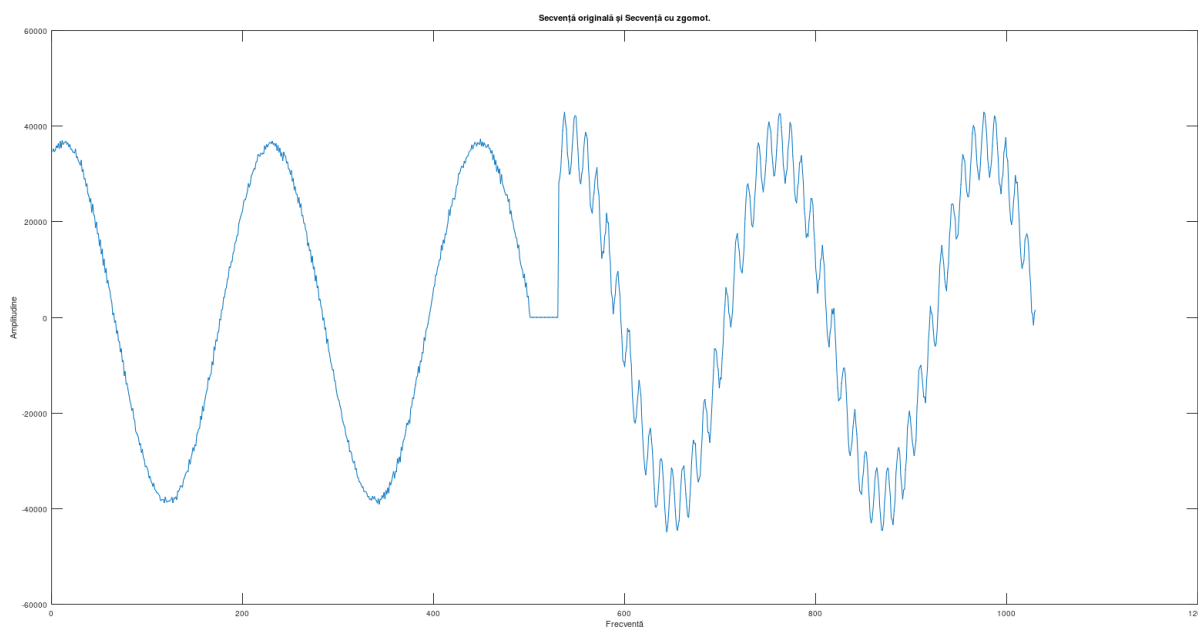
Add Noise?(y/n)
Set Signal Processing State...

Do you want to plot a sample from the incoming signal, together with the added noise?(y/n)
<Note that between the incoming signal and the altered one there are a couple of 0s>

```

Figură 26. Signal Processing State/Noise State Meniu

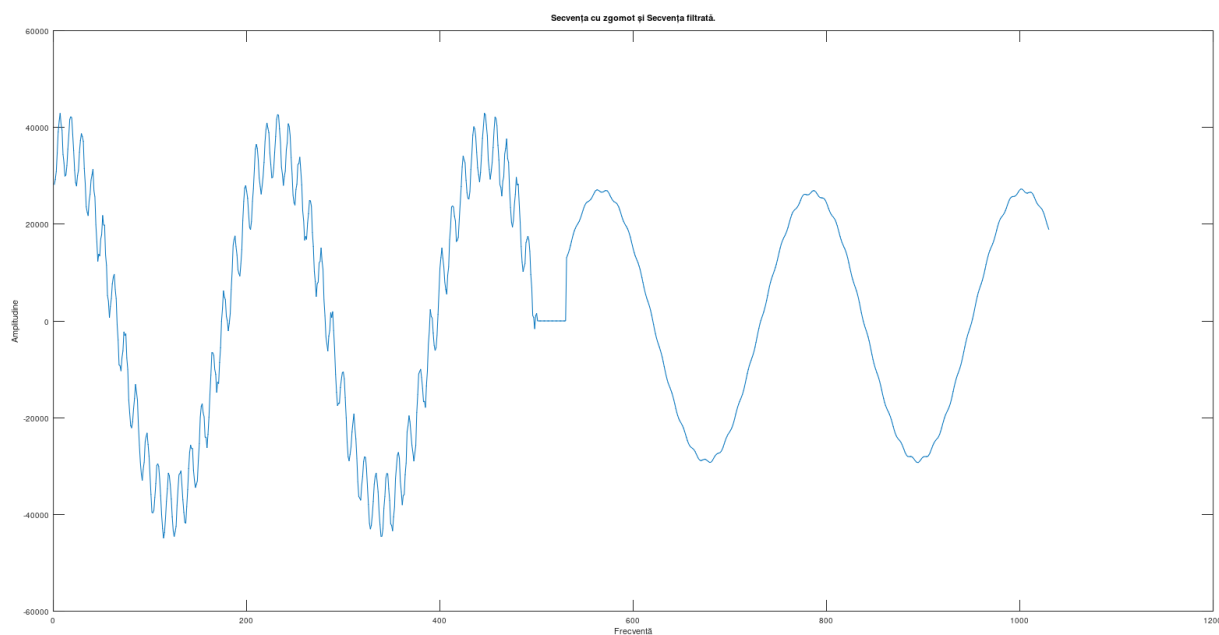
În continuare se afișează în terminal cele două semnale care sunt reprezentate în figura [27]. Aceste semnale pot să fie preluate din terminal și plotate în Octave sau Matlab. Ne sunt prezentate cele 500 de eșantioane din cele două semnale, primul semnal este cel original stocat în memoria plăcii Zybo, acesta este generat prin Online Tone Generator, cu o frecvență de 440Hz. Peste acest semnal se adaugă un alt semnal numit zgomot, care este tot de formă sinusoidală, cu o frecvență de 10 ori mai mare decât cea a semnalului audio și cu o amplitudine mai mare deoarece ne dorim să fie cât mai evident raportul dintre cele două semnale. Se poate observa cum semnalul este puternic distorsionat. Prin adăugarea unui sinus de frecvență diferită rezultă o sinusoidă complexă care prezintă variații în amplitudine și frecvență.



Figură 27. Secvență original (stânga) și Secvență cu zgomot (dreapta)

În următorul pas suntem întrebați dacă ne dorim să redăm secvența cu zgomot, apoi se trece în Filter State unde se aplică EMA asupra semnalului nostru. După aplicarea filtrului se afișează în terminal sub formă numerică semnalul cu zgomot și după semnalul filtrat, se poate vedea în figura [28]. La fel acestea se pot introduce în Matlab sau Octave și pot să fie plotate.

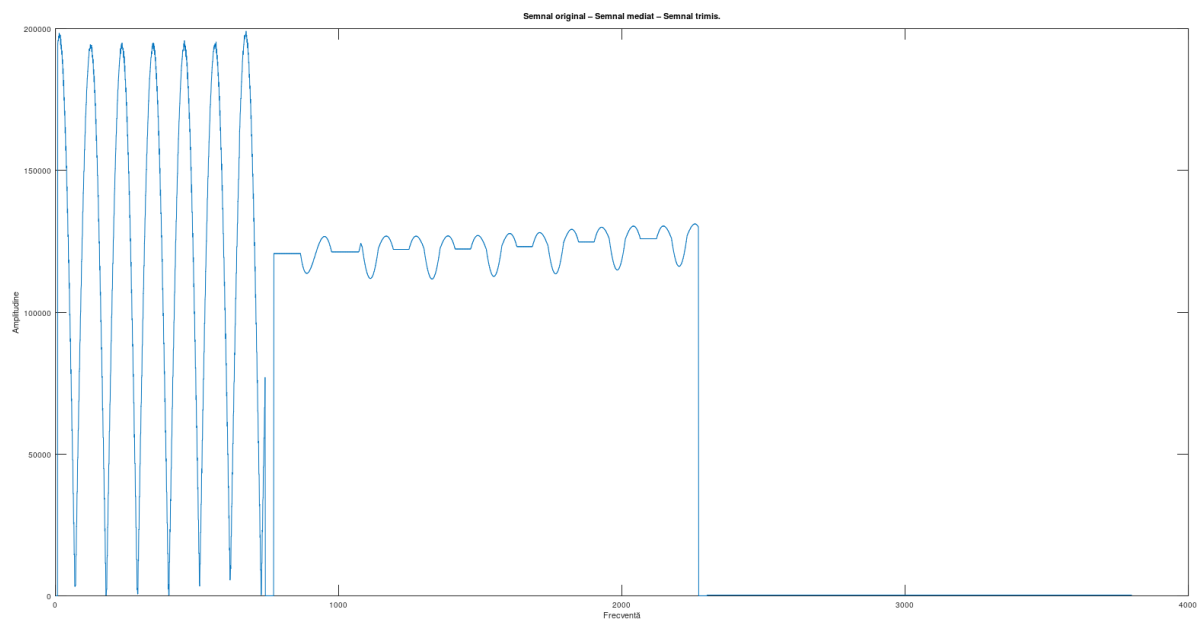
Așa cum s-a precizat și în partea de implementare software, filtrul EMA (Exponential Moving Average) este folosit pentru prelucrarea semnalelor. Rolul acestuia este de a preciza tendința de mișcare a datelor printr-un mod mai sensibil la valorile recente. Acest filtru atribuie pondere mai mare valorilor mai recente și pondere mai mică valorilor mai vechi. Se ia fiecare eșantion în parte și se calculează în funcție de sistemul de calcul EMA. Rezultatul este un semnal mediat, cu o amplitudine mai mică. Cu cât semnalul are mai puține distorsiuni cu atât și semnalul filtrat este mai lin.



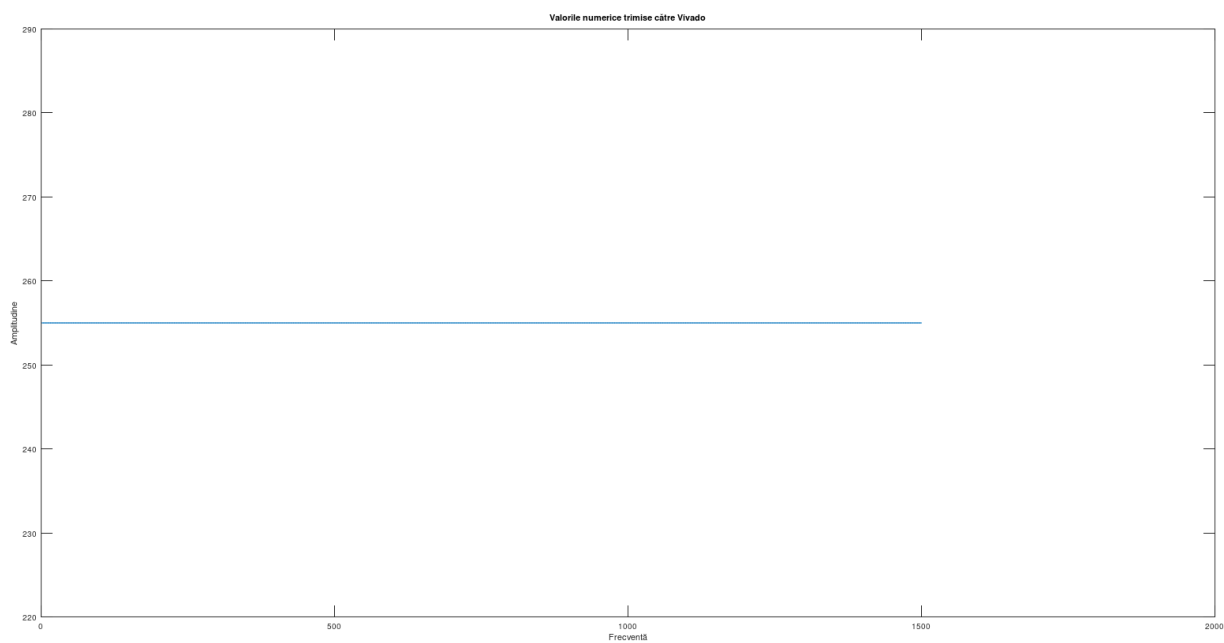
Figură 28. Secvența cu zgomot (stânga) și Secvența filtrată (dreapta)

Pentru afișarea grafică prin VGA se preia amplitudinea semnalului, care este dat de volumul setat din Laptop sau din PC. Se afișează atât semnalul original, pentru care ne interesează doar valorile pozitive, deoarece volumul este o valoare pozitivă, apoi se afișează semnalul mediat, acesta a fost calculat să ajungă cât mai aproape de o valoare constantă iar ultimul este semnalul trimis către componenta de video din Vivado. Se poate vedea în figura 31 și în figura 32.

Un sound bar prezintă diferite valorile ale volumului într-un anumit timp. Acest principiu a fost replicat și implementat folosind logica VGA. Astfel s-au creat 8 bare verticale care reprezintă volumul la diferite valori, cu cât volumul este mai mare cu atât o să se aprindă mai multe bare, culoarea lor merge de la verde (valoarea cea mai mică) până la roșu (valoarea cea mai mare). Funcționarea lor este prezentată în figurile [33 – 36]. Toată această afișare se realizează în Play State prin apelarea funcțiilor definite în acest state.

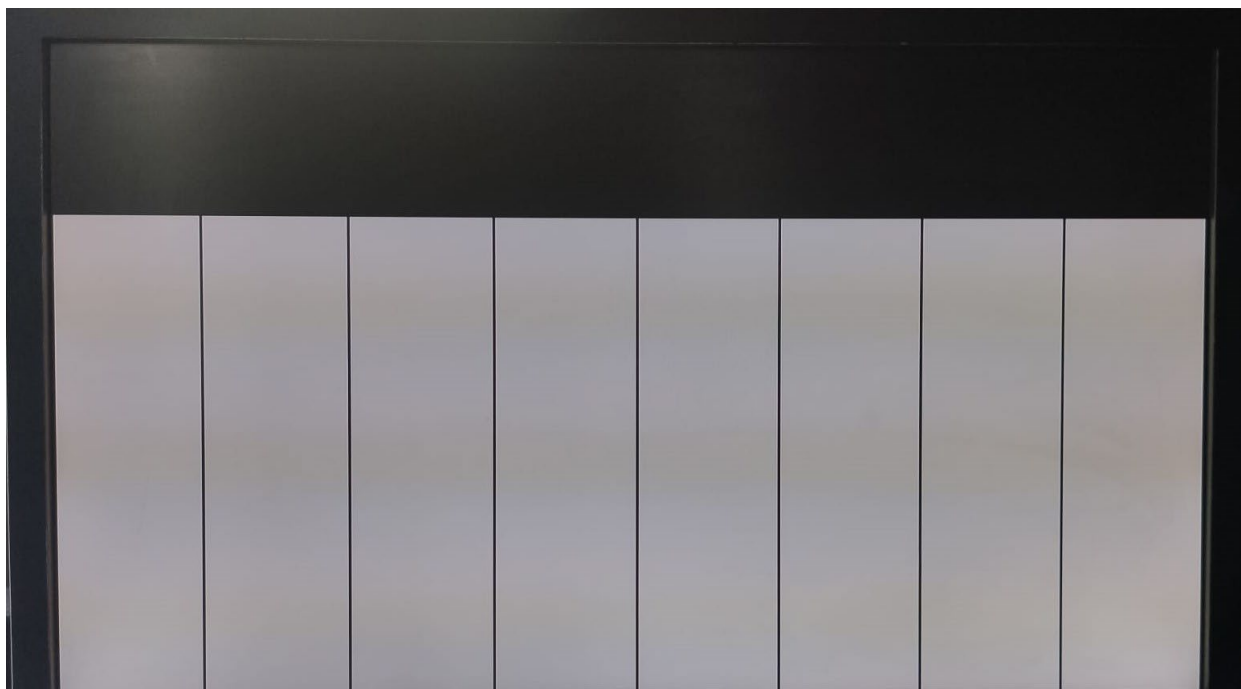


Figură 29. Semnal original (stânga)-Semnal mediat (mijloc)-Semnal trimis (dreapta)

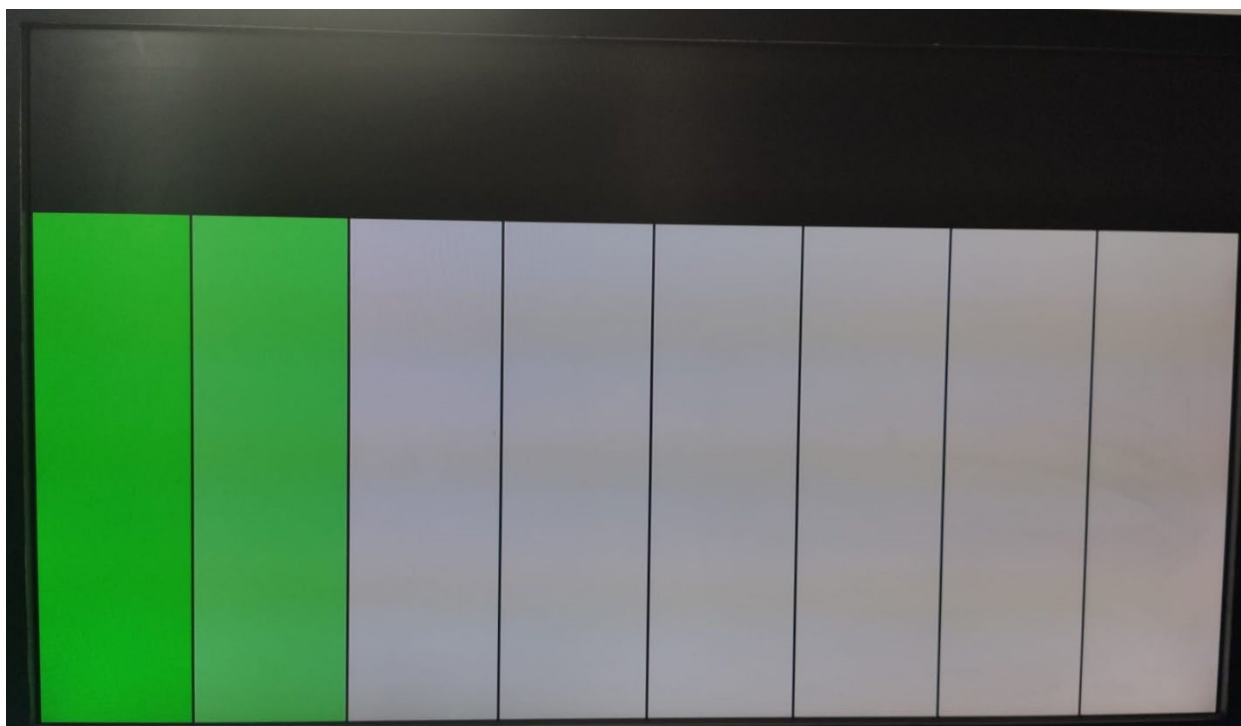


Figură 30. Valorile numerice trimise către Vivado

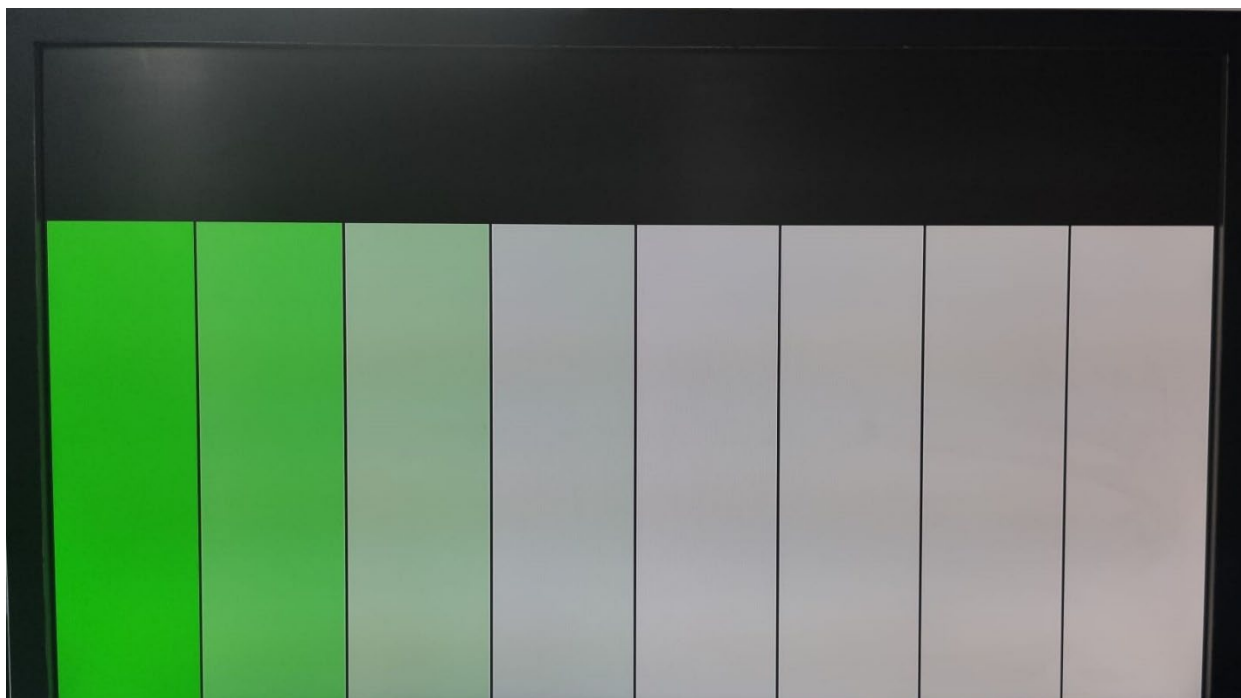
Pentru valoarea de 255 citită în decimal, aceasta este reprezentată ca fiind 1111 1111 în binar, atunci o să fim în cazul reprezentat din figura [34], unde se vor aprinde toate băbile de volum.



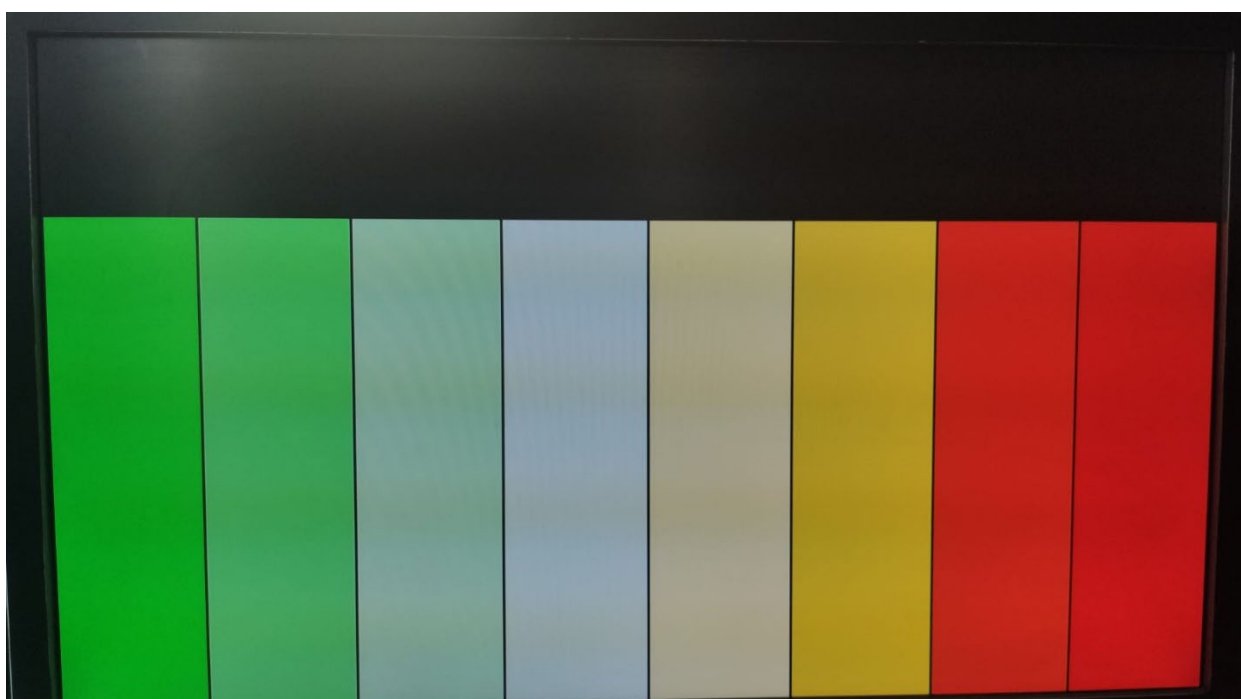
Figură 31. Sound bar la valoarea 0



Figură 32. Sound bar la valoarea 14



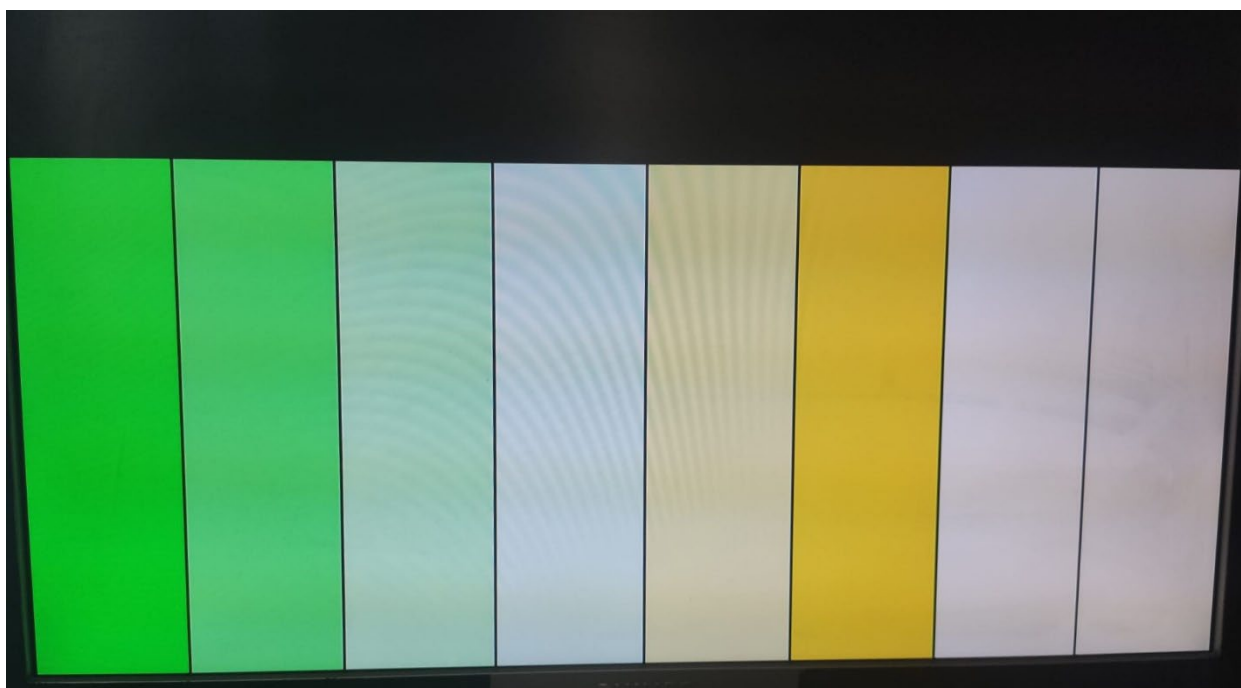
Figură 33. Sound bar la valoarea 20



Figură 34. Sound bar cu volumul la 32



Figură 35. Sound bar cu volumul la 26



Figură 36. Sound bar cu volumul la 28

7 Concluzii

În urma rezultatelor experimentale se poate trage concluzia că sistemul este funcțional și îndeplinește cerințele de bază a unui sistem audio. Am definit un egalizator de sunet ca fiind o componentă software care modifică amplitudinea și frecvența semnalului audio, aceste lucruri au fost efectuate în cadrul state mașinului iar prin intermediul componentei VGA s-a realizat și partea grafică.

Pentru modificarea volumului semnalului audio s-au realizat funcții care scriu în registrii interni ai SSM2603, acesta fiind codec-ul audio, valorile necesare pentru a crește volumul treptat până la valoarea de 33dB, respectiv pentru a scădea volumul se apelează o funcție asemănătoare care scade volumul treptat, în timpul creșterii și a scăderii volumului s-au adăugat și semnale de avertizare care ne indică dacă volumul este la un nivel prea mare sau la un nivel prea mic. Acestea au rolul de a proteja atât utilizatorul cât și aparatele folosite în acest proces. Adicional s-a adăugat și cazul de reset a volumului deoarece se poate întâmpla frecvent ca volumul setat să nu fie cel dorit.

Pentru modificarea structurii semnalului am dorit să adăugăm zgomot și apoi să fie procesat printr-un filtru cunoscut. Pentru a adăuga zgomot a trebuit implementat în cod un sinus cu o frecvență asemănătoare și cu o amplitudine considerabil mai mare decât semnalul original, deoarece ne dorim să fie cât mai vizibilă influența zgomotului asupra semnalului. Filtrul adăugat ulterior asupra semnalului este un filtru EMA, acesta este un filtru foarte folosit în industria gestionării bunurilor a căror valoare fluctuează în timp, precum instrumentele de acțiune la bursă, imobiliare, metale prețioase, bunuri valorice și mai recent industria crypto.

Pentru afișarea video prin VGA, valorile eșantioanelor audio au trebuit să fie încadrate între anumite limite, acestea au fost alese atent astfel încât să fie distribuite egal, să nu rămână valori libere și să corespundă un anumit set de valori cu reprezentarea grafică.

Se poate afirma că acest proiect preia atributele unui egalizator de sunet și poate să fie o bază pentru viitoare proiecte ce pot să fie dezvoltate pe structura acestuia. Pentru a rula acest proiect este nevoie doar de placa de dezvoltare Zybo Digilent, de un cablu VGA și un monitor VGA, recomandat să suporte rezoluția de 1980x1080 pixeli, de un sistem audio, un microfon sau un cablu cu ambele capete de tip jack și de cablurile aferente pentru alimentare și programare a plăcii.