RECONFIGURABLE HARDWARE DESIGN

SADHANA RAMU

STUDENT NO: 230611159

SUBJECT CODE: EEE8088

Dr. Alex Bystrov
MSC – EMBEDDED SYSTEM AND IOT
SCHOOL OF ENGINEERING

TABLES OF CONTENTS

ABSTRACT

The goal of this project is to use the DE1-SoC development board to design and construct a Finite Impulse Response (FIR) filter that is especially suited for audio signal processing on an FPGA platform. The principal goals were twofold: first, to enhance comprehension and practical abilities in digital signal processing, FIR filters, and FPGA technology through education; second, to accomplish the technical goal of effectively implementing a low-pass FIR filter that is fully functional and integrated with a CODEC initialiser and S2P adaptor, utilising the I2C protocol for efficient communication and control.

The process used an organised approach, starting with a thorough design step where the specifications for the FIR filter were established. This was followed by a thorough implementation phase where the FPGA was programmed using VHDL. Using the I2C protocol, the CODEC was configured; an S2P adapter was created for serial-to-parallel data conversion; and the FIR filter was designed to adhere to the necessary audio processing requirements.

The primary outcomes of the trials showed that the FPGA-based system effectively reduced noise and improved signal clarity by filtering audio signals using the intended FIR filter. The filter's response matched theoretical predictions after extensive testing, demonstrating the system's capability to handle high-fidelity audio inputs and validating its performance.

In an overview, the project achieved its technical and instructional goals, strengthening the real-world applications of digital signal processing theories and offering insightful information about FPGA-based audio processing systems. In addition to serving as an excellent example of how hardware and software can work together in signal processing applications, the FIR filter's successful implementation on the FPGA platform also set the stage for further research and development in the field of reconfigurable hardware design.

AIMS AND OBJECTIVES

AIMS:

Educational aim: The goal of this project is to provide a comprehensive grasp of digital signal processing (DSP) and how it applies to communications. The topics of this study are the Altera DE1 Field-Programmable Gate Array (FPGA) platform and its use in the design and implementation of a digital FIR filter for audio signal processing.

Technical Aim: The technological goal of this work is to create an advanced digital FIR filter system that processes audio signals as efficiently as possible. The project intends to efficiently interface, compile, and simulate the three main pieces of the design—the S2P Adapter, Codec initialization, and FIR filter—to guarantee operation. The practical goal is to create a high-quality, VHDL-programmed FIR filter system on an FPGA platform to demonstrate the usefulness of DSP strategies in real-world applications.

Objective:

1. The objective of this study is to establish the necessary specifications and architecture for the design and execution of the FIR filter, which is used for audio signal processing on the FPGA platform.
2. Codec Specifications and Procedures: comprehending and evaluating the requirements and workings of the integrated circuit Codec (WM8731), which is essential to audio processing.
3. I2C interface that the codec is initialised over, with an emphasis on its setup and protocol for communication.
4. A Review of the Template: Examining the DSP interface specifications for effective audio signal processing in the FIR filter design.
5. ModelSim simulations are used to verify and assess the functionality and efficiency of the FIR filter system that is created.
6. Use Signal Tap, Signal Generator, and Oscilloscope for debugging purposes.

1. INTRODUCTION

The project "Design and Implementation of Communication Digital FIR Filter for Audio Signals on the FPGA Platform" focuses on the creation of a finite impulse response (FIR) filter for audio signal processing on a field-programmable gate array (FPGA) platform. Among the discussion's main ideas are digital signal processing (DSP), FIR filter.

Due to its high performance, digital signal processing (DSP) is frequently used in telecommunications. The FIR filter is necessary for processing audio signals on FPGA systems. The three primary blocks that comprise the design are the FIR filter block, the codec initialization block, and the S2P adaptor block. To handle audio signals properly, these blocks are interfaced together.

Applications involving audio processing can function well thanks to Field Programmable Gate Arrays (FPGA) systems, which offer reconfigurable designs with high-performance processing capabilities. FPGAs are programmed using the Hardware Description Language (VHDL) for very high-speed integrated circuits. It outlines the behaviours of the various parts so that designs may be simulated before being put into practice.

The WM8731 Codec is an integrated circuit that employs low power consumption in stereo mode for audio processing. It has interfaces for audio input, output, and control. The band pass, high pass, band pass, and band stop filters are among the various variations of the linear phase response FIR filter, which is a non-recursive digital filter used to reject unwanted frequencies and choose desirable ones.

One FPGA series that stands out for its simplicity is the one that offers processor support and embedded DSP multipliers. Furthermore, as seen in Figure 1, the Altera DE1 board has an extensive feature set thanks to its Cyclone II FPGA. This makes it a great option for a range of design projects and the development of intricate digital systems.

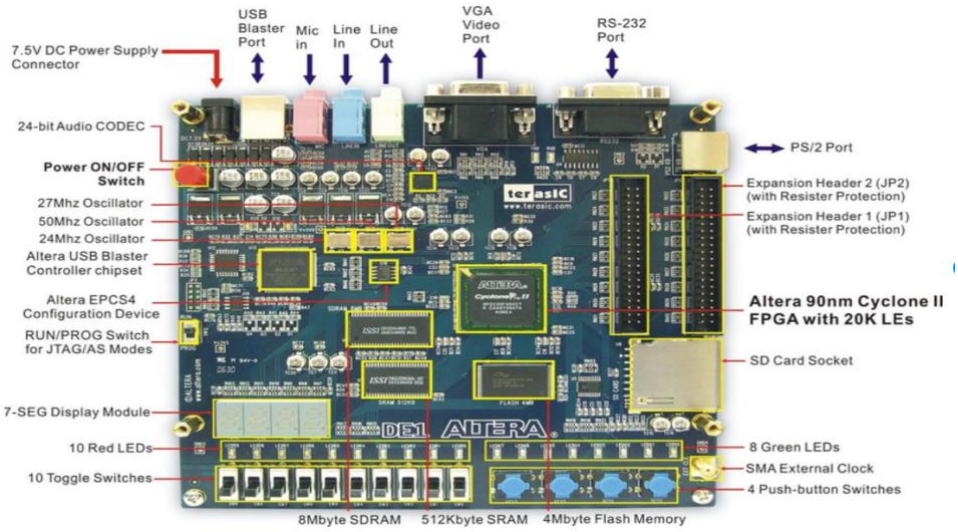


Fig.1: Altera DE1 Board

Group Management:

Each member of the team created their own programming and waveform analysis on their own. We then looked over and talked over the code and outcomes as a group to make sure we had a common understanding and strategy.

2. METHODOLOGY

To be able to ensure that requirements are satisfied, designs are refined, implementations are accurate, and validations are comprehensive, complex systems must be designed and developed using the Specification-Refinement-Implementation-Validation (SRIV) cycle. This cycle is carried out in the following ways within the framework of the entire system, which consists of the FIR filter, S2P adaptor, and codec chip:

- **Specification**:

  - The cycle starts with the specification phase, during which the system's functional specifications and needs are established. Determining each component's functioning, the interfaces that connect them, and the general behaviour of the system are all included in this.

- The specifications for the data format conversion in the S2P adaptor, the filtering function of the FIR filter, and the initialization of the codec chip are laid down in this step. This entails being aware of the timing restrictions, clocking specifications, and data formats.

- **Refinement:**

  - The refinement phase concentrates on improving the initial designs by the established specifications after they have been established. This entails disassembling the system into more manageable parts or subsystems and continuously improving their designs.
  - The serial-to-parallel and parallel-to-serial conversion processes may be divided into discrete modules in the context of the design of the S2P adaptor, each having its own set of specifications and features.

- **Implementation:**

  - After the designs are finalised, the next step is to implement them on the FPGA by converting them into real hardware description language (HDL) code, such as Verilog or VHDL.
  - To ensure modularity and reusability, every system component—including the FIR filter, s2p adaptor, and codec initializer—is implemented as a separate module with documented interfaces.

- **Validation:**

  - The validation phase, which concludes the SRIV cycle, is essential for confirming that the system that has been constructed satisfies the necessary specifications and operates as intended under a range of circumstances.

- This entails testing the system both at the system and module levels. System-level testing guarantees correct integration and interaction between components, whereas unit testing is used to test specific modules, like the FIR filter or codec initializer, to confirm their operation.
- Verifying the system's performance against anticipated outcomes, such as signal fidelity, latency, and resource usage, is another aspect of validation.
- Iterations may take place during this SRIV cycle as problems are found and fixed, guaranteeing that the installed system ultimately satisfies the intended requirements and performance standards. By reducing risks and uncertainties through iteration, a strong and dependable system design is produced.

2.1 Component-system-hardware chain

The Components-System-Hardware (CSH) chain of the entire system including the codec chip, s2p adapter, and FIR filter defines the hierarchy and interconnectivity of individual hardware components to create a coherent system architecture meet Let's take a closer look at how this works Using the CSH chain:

At the lowest level of the CSH series, individual hardware components, each serving a specific function in the system. These factors include:

- **Codec chip**: Interface with external audio equipment, responsible for providing serial input data and receiving serial output data.
- **s2p Adapter**: Acts as interface between codec chip and FIR filter, converting serial data to parallel format and vice versa.
- **FIR Filter**: Performs signal processing on the input data received from the s2p adapter, using an 8-tap FIR filter function.

2.2 SYSTEM:

- The system layer of the CSH series represents multiple hardware components that are integrated and coordinated to achieve high performance. In this case, the system includes a codec chip, s2p adapter, and FIR filter along with any additional components or subsystems necessary for overall system functionality.
- The s2p adapter acts as an important interface between the codec chip and the FIR filter, facilitating data exchange and synchronization between serial and parallel formats.
- The system orchestrates data flow between components, ensuring proper timing, synchronization, and signal integrity throughout the signal processing pipeline.

2.3 HARDWARE:

- The hardware layer of the CSH series consists of the physical implementation of the system on FPGA hardware. This includes the development of hardware description language (HDL) code for each component and its integration into the FPGA platform.
- Each component is implemented as a hardware module in the FPGA, using components such as logic elements, flip-flops, and memory blocks to achieve the desired functionality.

- Communication between components is established through dedicated signal lines, clock stations and data buses to ensure smooth communication and data transfer between modules.

Overall, the CSH series provides a hierarchical system and implementation of system hardware components from individual modules to integrated subsystems, culminating in a symbolic processing pipeline that is fully functional at On FPGA platform This approach provides efficient hardware design, modularization, and scalability.

3. DESIGN SPECIFICATION AND REFINEMENT

3.1 Design Specification

- Analog Audio Codec Interface: The system is the analog audio codec that acts as a source and destination for audio signals. A codec converts analog audio signals into a digital stream and vice versa.
- Digital Stream Format: The digital stream is represented in 16-bit format, maintains audio fidelity, and operates at a sampling frequency of 44.1 kHz, which complies with CD-quality audio standards.
- Configuration mode: The system operates in the main channel configuration, where the FPGA controls the data exchange with the audio codec, ensuring synchronization of communication and data.
- Processing Module: The FPGA hosts the FIR filter module which controls the signal processing tasks. The FIR filter uses 8 taps to perform filtering functions on a digital audio stream, enhancing audio quality or removing relevant features.
- Language: VHDL was chosen as the programming language to be used in the system operation, providing a robust and flexible platform for digital circuit design and simulation
- Audio Interface: The system uses DSP mode for the audio interface, with the left/right clock (LRP) set to 1. This setting ensures proper synchronization and timing between the FPGA and the audio codec when data is exchanged.
- Master Clock: The FPGA uses a master clock frequency of 50 MHz, which is used to represent all internal processing and data processing functions.

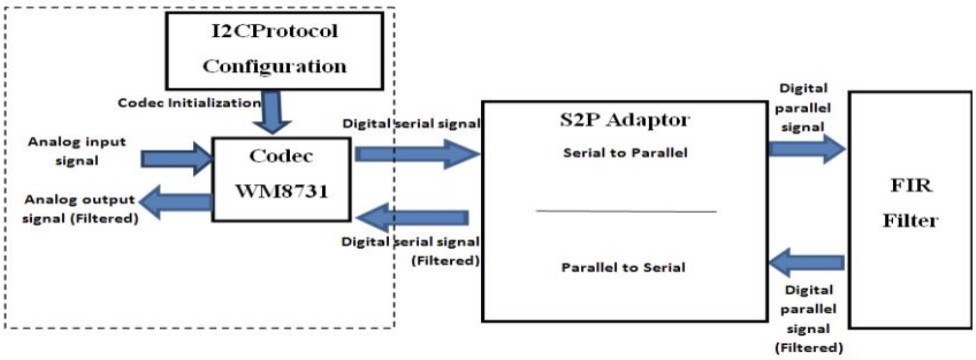


Fig 3.1: Audio Digital Filter System.

3.2 REFINEMENT:

The system is divided into smaller subsystems or modules, and the designs of each are iteratively refined during the refinement phase, which is based on predetermined specifications.

- **Optimized Filter Taps:** While the FIR filter requires eight taps, the number of taps has a direct impact on the filter's efficiency and use of resources. The frequency response of the filter may be enhanced, depending on the application, by adding more taps, although doing so will require more FPGA resources. On the other hand, cutting back on taps can conserve resources without improving performance. As a result, it's critical to tailor the tap count to the requirements of the application.

- **Efficient VHDL Coding:** Digital circuits can be effectively described and simulated using the VHDL language. Adopting best practices, such as minimising the usage of loops, employing concurrent statements for parallel tasks, and effectively managing signal assignments to optimise synthesis results and enhance simulation time, can improve the efficiency of the VHDL code.

- **Clock Management:** Internal processing uses a master clock frequency of 50 MHz Phase-Locked Loops (PLLs) and Clock Management Tiles (CMTs) are two examples of FPGA features that can be used to optimise performance and save power consumption in a system by enabling numerous derived clocks at different frequencies for different areas of the system.

- **Adaptive Sampling Frequency:** While 44.1 kHz is the industry standard for audio on CDs, greater sampling rates may be advantageous for contemporary applications. For high-definition audio streams, implementing an adaptive sampling frequency that can be changed based on the input signal could further improve audio fidelity.

- **Dynamic Configuration:** The primary channel configuration mode is used by the system. The system's flexibility and adaptability can be increased by implementing dynamic configuration modes that can be changed in real-time in accordance with the requirements of the audio signal.

- **Resource Utilisation:** Keep close attention to how the FPGA is using its resources. If there is a lot of extra space on the FPGA, you might want to add more functions, including more sophisticated audio processing. If funds are limited, consider ways to improve the current design.

- **Power Optimisation:** Consider employing power-saving strategies such clock gating or switching to a lower-power state when the audio processing is not required, as FPGAs can consume a lot of power, particularly at high clock frequencies.

- **Enhancement of the Audio Interface:** This might be another area for improvement if the DSP mode parameters of the audio interface can be changed to increase system performance. This could involve modifying the clock polarity on the left and right side or the frame sync signal timing to better accommodate the needs of various audio codecs.

- **Hardware Testing and Simulation**: Thorough simulation and hardware testing should be carried out to make sure the design satisfies the requirements. This might reveal unanticipated problems that need to be fixed to further improve the system.

4. BLOCKS:

a. BLOCK 1: CODEC INITIALIZER

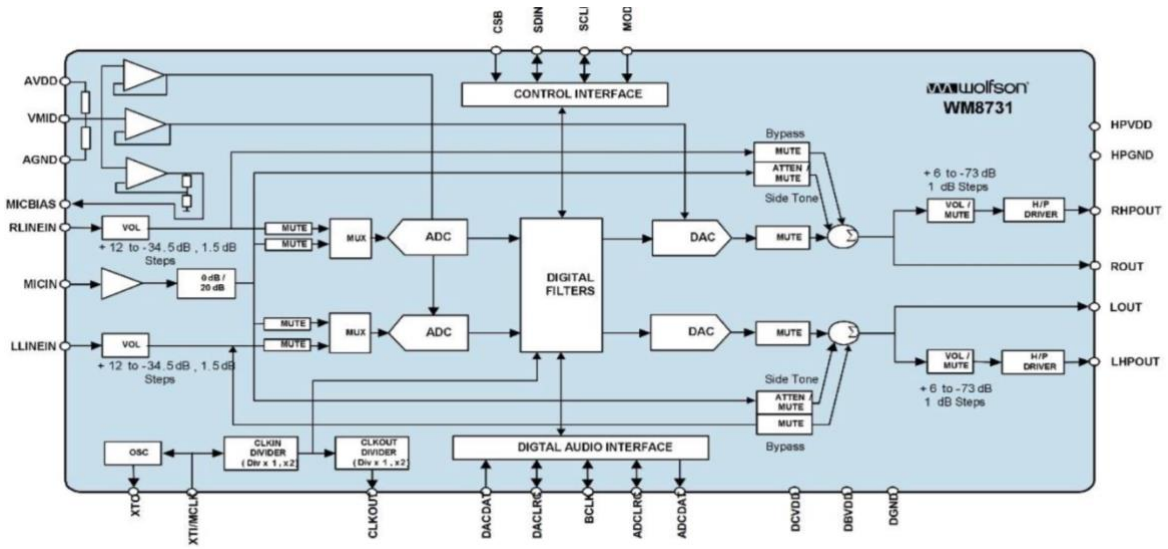


Fig 4.1: The Block Diagram of Codec (WM8731)

The basic implementation requires the codec chip to be configured using the I2C (Inter-Integrated Circuit) protocol, which is a synchronous serial communication protocol widely used for the communication of digital integrated circuits.
A given data stream consists of several 24-bit blocks, each with a specific configuration for the codec chip. Break down the order of each 24-bit block:

1. Chip Address (7 bits): This specifies the address of the codec chip in the I2C network. Each device connected to the I2C bus is assigned a unique address, which enables targeted communication with individual devices.
2. Read/Write Bit (1 bit): This bit indicates whether the next operation is a read or write operation. In your case, it appears that all bits are set to 0, indicating write operations.
3. Acknowledge Wait: Following the address and read/write bit, the sender must wait for an acknowledgment signal from the receiving device to confirm that a valid address has been detected.
4. High Bits of Register (8 bits): These bits represent the high-order bits of the register address in the codec chip where the configuration data will be written.
5. Wait for acknowledgment: After sending the high bit of the register address, the sender waits for an acknowledgment signal from the receiving machine.
6. Low bits of the registers (8 bits): These bits represent the low bits of the register address in the codec chip where the configuration data will be written.
7. Wait for approval: Like pre-approval, this step confirms that the registration address has been received successfully.

REGISTER ADDRESS	REGISTER DATA	DESCRIPTAION
R0 (00h)	000011111	Set the Mute and the Volume of Left Line channel.
R1 (01h)	000110111	Set the Mute and the Volume of Right Line channel.
R2 (02h)	001111001	Set the output volume and control the Left headphone out.
R3 (03h)	000110000	Set the output volume and control the Right headphone out.
R4 (04h)	011010010	Analogue Audio Path Control and set (ADC function).
R5 (05h)	000000001	Digital Audio Path Control
R6 (06h)	001100010	Power Down Control, this register can enable or disable the power down inside the Codec.
R7 (07h)	001000011	Digital Audio Interface Format. The input data indicate Codec's mode and the data input length (16 bits).
R8 (08h)	000100000	Sampling control. For example the sampling frequency is 44.1 kHz.
R9 (09h)	000000001	This register controls the active interface.
R15 (0fh)	000000000	Reset the device.

Fig 4.2 : Register Map

In addition, the I2C protocol includes start and stop states for each 24-bit transfer to ensure proper synchronization between sender and receiver. The purpose of sending this data to the codec chip is to configure its internal registers according to the specified settings. These settings include parameters such as sample rate, audio format, gain levels, and various other configuration options tailored to our application's requirements. By configuring these registers, the codec chip is effectively initialized to perform its designated audio processing tasks accurately, thereby enabling seamless integration within the FPGA-based system.

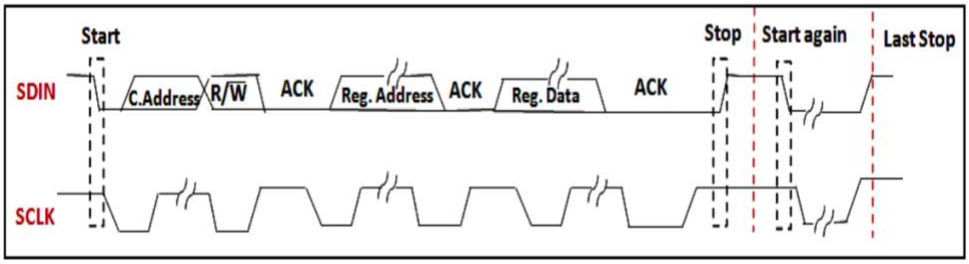


Fig 4.3 : I2C protocol

- **Discussion:**

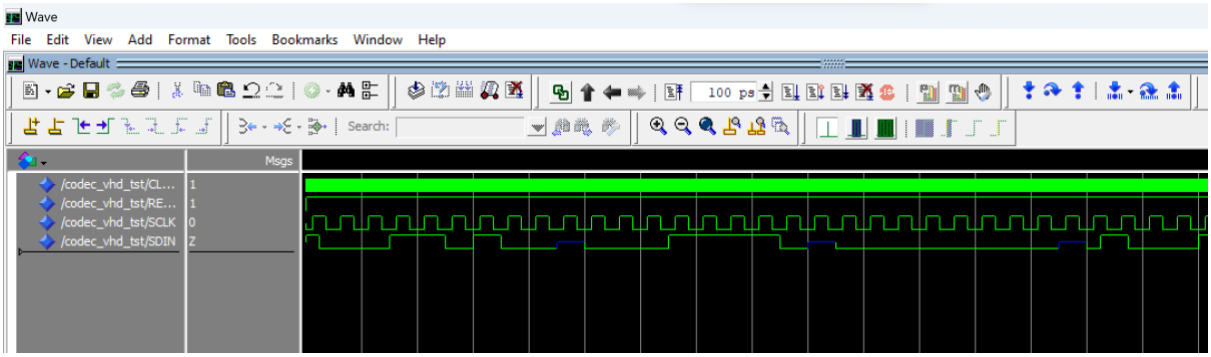


Fig4.4 . codec initializer output

- **Codec output discussion**:

In the Obtained waveform I have noticed that the digital signals displayed in the output waveform depicted in the figure are indicative of the I2C communication protocol that was utilised to initialise the codec. For the codec chip to be configured to operate within the given parameters, the order of processes shown in the waveform is essential such as addressing, Write Operation, Acknowledgment (ACK) signals, Register Addressing, Start and Stop Condition.

Let’s discuss about the parameters:
The I2C protocol makes it easier for a codec to initiate its start-up sequence, which is essential for proper operation. To prevent configuration problems on a bus with several devices, the **7-bit chip address** is essential for identifying the relevant device. When the read/write bit is set to 0, the codec is instructed to accept data for its registers instead of sending data, indicating a write operation. The protocol's handshake includes **acknowledgment signals**, which verify successful data receipt. If these are absent, there may be a communication problem. The codec's features, including power management and audio

signal routing, may be precisely configured thanks to register addressing, which is split into high and low bits. The data packet is framed by the **start and stop conditions**, where the start condition marks the beginning of communication, and the stop condition marks its end.

Knowing about the **Internal Registers** and its importance:

- An audio codec's internal registers are essential to understanding how it functions. They choose crucial audio parameters like bit depth, format, and sample rate that guarantee the codec keeps up with the rest of the audio system. The loudness of the audio stream is influenced by registers that control gain and volume, which is crucial for preserving distortion-free sound quality. The codec's functionality and energy efficiency are affected by the many operating modes, ranging from normal to power-saving, that are set by these registers. In order to comply with particular application requirements, the registers are also in charge of configuring the digital interface between the codec and the FPGA. They also specify the clocking strategy and the audio data format, such as PCM or I2S.

4.2 BLOCK 2: S2P ADAPTOR

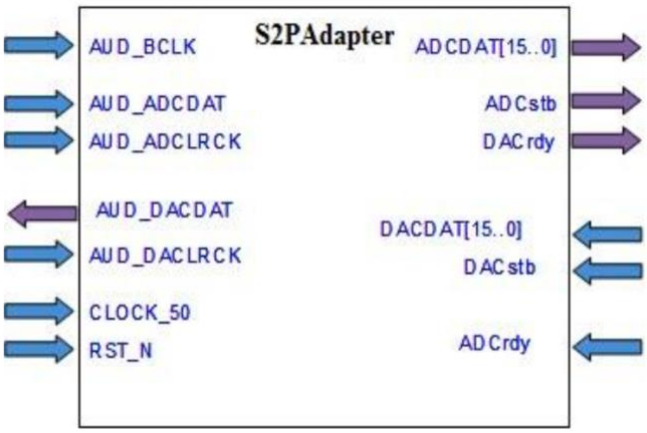


Fig4.5: Block Diagram of S2P adapter

The s2p adapter block acts as an important interface between the codec chip and external systems, facilitating the conversion of serial data to parallel output and vice versa This system has two main components: an input channel and an output channel, each of which handles a specific data transmission task.

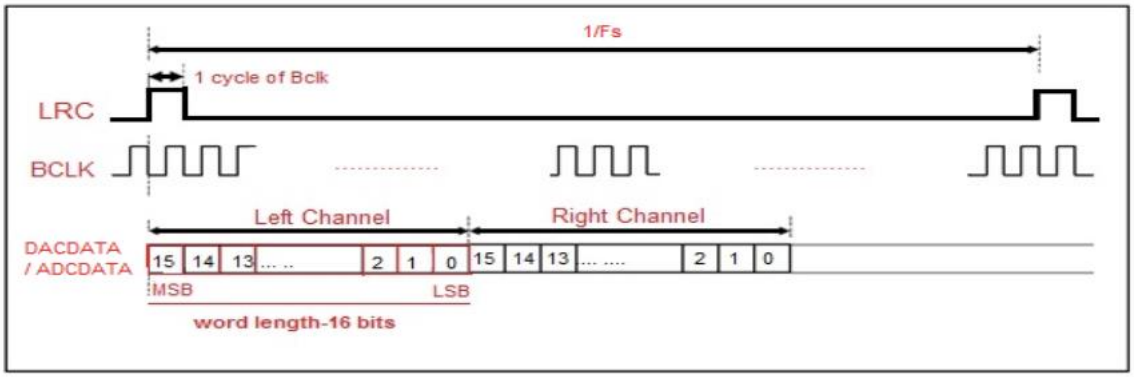


Fig 4.6: Digital Audio Interface

In the input channel, the serial data received from the codec chip is converted into parallel output data. This function is identical to the AUD_BCLK clock provided by the codec chip and ensures that the time is accurately synchronized. AUD_BCLK operates at a frequency of 64 * 44.1 kHz, indicating its use in synchronizing data conversion and codec chip operation. The input channel is based on the LRC (Left/Right Clock) signal generated by the codec chip to initiate data recording. When the LRC signal indicates the presence of input data, the input channel starts working. The strobe signal plays an important role in determining the readiness of the 16-bit register. When the register is filled, the strobe signal goes high for one clock pulse, indicating that the 16-bit data is ready for parallel transfer. This synchronization method ensures proper data usage and prevents data loss or corruption.

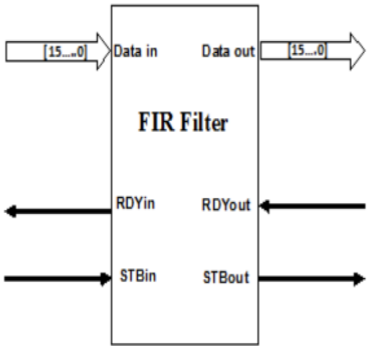


Fig4.7: Parallel Interface

Conversely, in the output channel, the parallel input data is converted into sequential output data for the codec chip. As with the input channel, this conversion pattern is identical to the AUD_BCLK clock provided by the codec chip. The strobe signal on the output channel indicates whether the 16-bit data is ready for serial transmission, and if it is high, the data is stored in the buffer.

Receiving a trigger from the LRC signal indicates that the codec chip is ready to receive output data, the output channel initiates the sequential transmission process and the filled 16-bit data buffer is sent serially in sync with the AUD_BCLK clock. This ensures that data is transmitted in a timely manner and in line with the expectations of the codec chip.

The s2p_adaptor system effectively bridges the gap between the serial data interface of the codec chip and the parallel data processing needs of the external system. By carefully synchronizing the AUD_BCLK clock and using LRC signals for data trigger events, the system ensures smooth data conversion and transmission Furthermore, the strobe signal functions as a key component indicates data readiness, all input and output methods Provides efficient data handling and synchronization.

- **Discussion:**

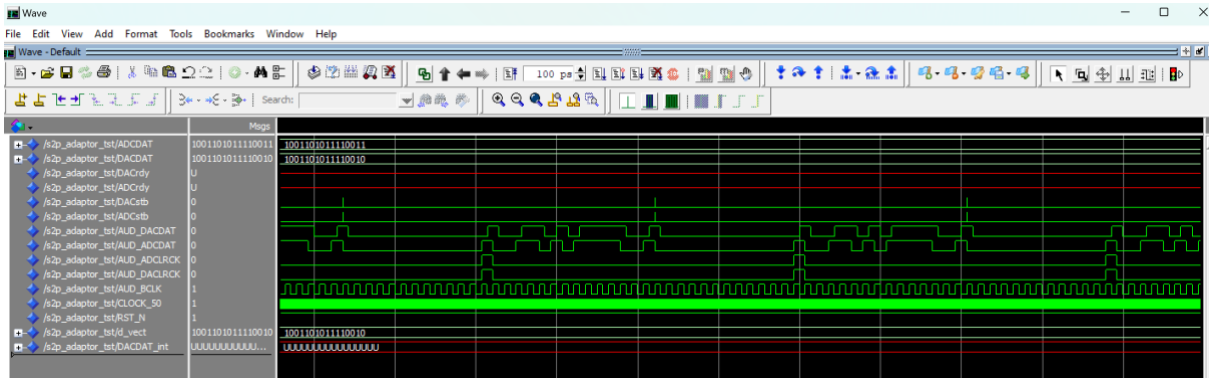


Fig.4.8: s2p adaptor functionality validation

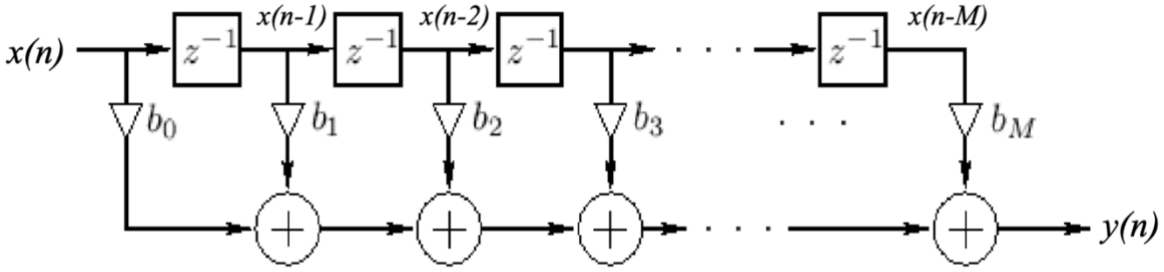
- **S2P Adapter output discussion**:

The waveform verifies that the external parallel input systems and the codec's serial output are correctly interfaced by the s2p adaptor. Data conversion is synchronised with the industry-standard audio sample rate and the codec's clock signal. Parallel data is converted from serial data via the LRC signal, which also activates the input channel. To ensure that all data is captured completely before transfer, a strobe signal is used to signify when the data is ready for output. In a similar manner, the adapter ensures timely delivery for the output by converting parallel data to serial, synchronising with the codec's clock, and initiating transmission as soon as the LRC signal is received. The performance of the adaptor is essential for preserving audio quality and avoiding data corruption.

4.3 BLOCK 3: FIR FILTER

The FIR (Finite Impulse Response) filter VHDL process acts as one of the most important signal processing aspects of your system. It operates on receiving 16-bit parallel input data from the s2p adapter and uses an 8-tap FIR filter function to generate a 16-bit parallel output, which is then sent back to the s2p adapter for conversion to serial data and sent to a codec chip.

The FIR filter uses an 8-tap filter function, meaning that it uses 8 coefficients to perform a weighted average over the input data samples. The assigned coordinates (-1260, 7827, 12471, 16384, 16384, 12471, 7827, -1260) determine the specific weights used for each input sample.



$$y(n) = \sum_{k=0}^{M} x(n-k)b(k)$$

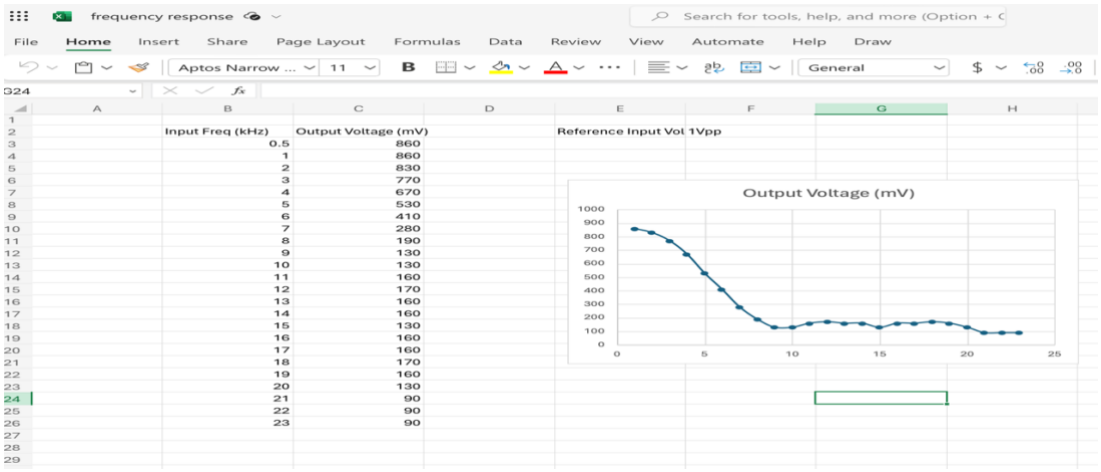
Fig4.9 : FIR Filter (M+1) taps

The FIR filter system receives 16-bit parallel input data from the s2p adapter. The filter is also applied to each input sample, using specified coefficients for the weighted averaging operation. The filter calculates the output value by multiplying each input sample by its corresponding coefficient, integrating the results, and producing the filtered output.

Upon completion of the filter operation, the system provides a 16-bit parallel output. This modified output data is then sent to the s2p adapter for conversion to serial data and sent to the codec chip. The parallel output data matches the input requirements of the s2p adapter, ensuring seamless integration throughout the system.

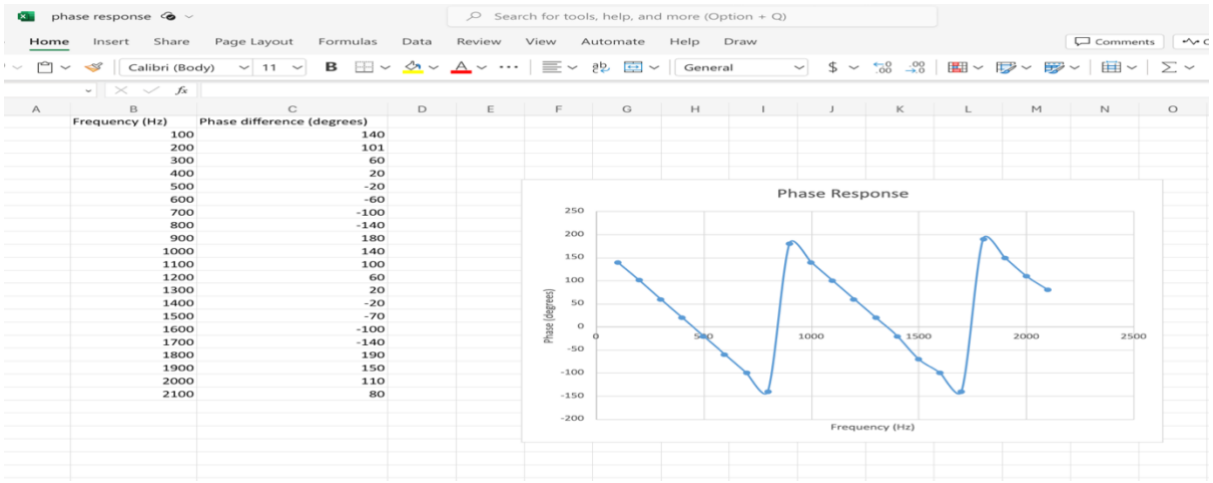
The FIR filter system includes a strobe signal indicating completion of filter operation and readiness of the 16-bit output data to be sent to the s2p adapter When the filter operation is complete, and the output data is available, the strobe signal goes high, which it is an output channel with s2p adapter signals that initiate conversion of the parallel output data to serial format and send it to the codec chip.

The FIR filter plays a critical role in signal processing by applying the specified filter coefficients to the input data, resulting in a filtered output that meets the desired signal. By using the strobe signal, the system ensures proper synchronization between the FIR filter operation and the data transfer process, enabling efficient communication between system components and seamless integration within the overall DSP pipeline.



Input Freq (kHz)	Output Voltage (mV)
0.5	860
1	860
2	830
3	770
4	670
5	530
6	410
7	280
8	190
9	130
10	130
11	160
12	170
13	160
14	160
15	130
16	160
17	160
18	170
19	160
20	130
21	90
22	90
23	90

Fig4.10 : Frequency Response of the FIR filter



Frequency (Hz)	Phase difference (degrees)
100	140
200	101
300	60
400	20
500	-20
600	-60
700	-100
800	-140
900	180
1000	140
1100	100
1200	60
1300	20
1400	-20
1500	-70
1600	-100
1700	-140
1800	190
1900	150
2000	110
2100	80

Fig4.11: Phase Response of the FIR filter

Discussion:

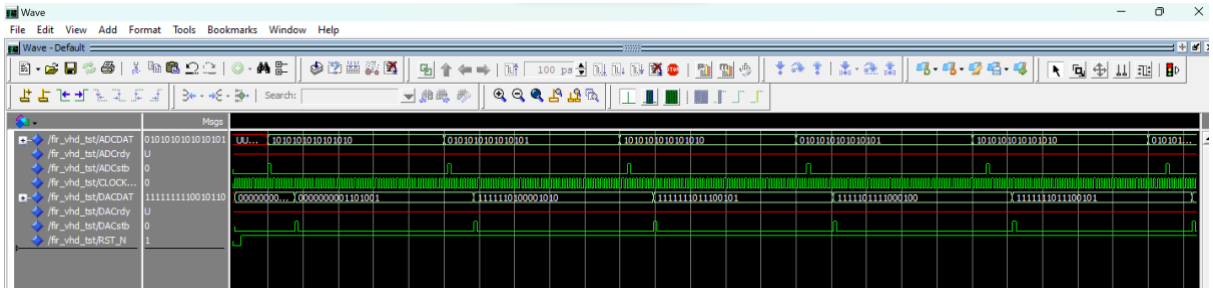


Fig4.12: fir filter validation

- **FIR Filter output discussion**: The frequency properties of the audio signal can be altered with the help of the FIR filter. To fine-tune the signal, it passes 16-bit input data through an 8-tap weighted mechanism. The specific response of the filter is determined by each coefficient that is allocated to a tap. After the modified signal is

output by the filter in parallel form, the codec must convert back to serial format in order to receive a strobe signal indicating that the stage is ready. The FIR filter is crucial for improving the quality of audio since it reduces noise, among other things. The waveform validation clearly illustrates how well it works, showing the effect of the filter and confirming that the signal satisfies requirements for additional usage.

5. SYSTEM OF BLOCKS

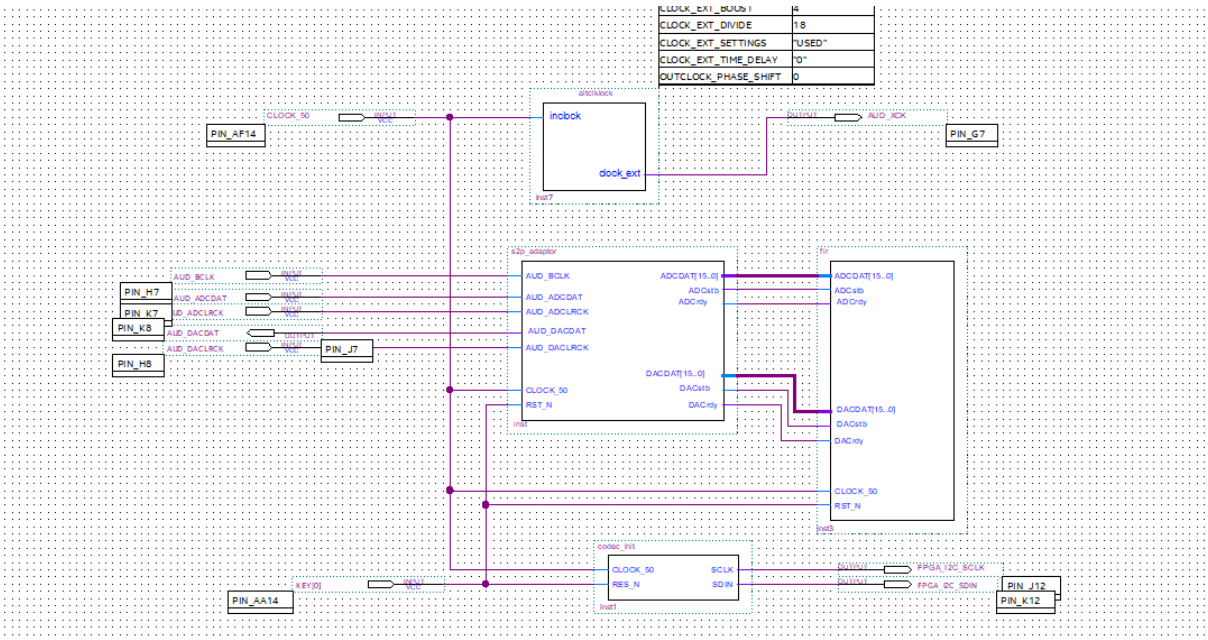


Fig.5: System of block

The overall design of the system, including the codec chip, s2p adapter, and FIR filter, forms a comprehensive signal processing pipeline, and provides efficient data transfer and conversion in the FPGA-based system.

At the heart of the system is the codec chip, which acts as a interface to external audio devices and provides serial input data to the s2p adapter. The s2p adapter acts as an intermediary between the codec chip and the FIR filter, converting the sequence of input data into a parallel format for processing by the FIR filter After the s2p adapter receives the filtered output from the FIR filter, The s2p adapter converts the parallel data to serial in a format to be sent back to the codec chip.

This integrated system ensures seamless communication and data processing between the parties. The codec initializer sets the initial configuration of the codec chip, making it work properly in the system. The s2p adapter makes it easy to switch between serial and parallel data formats, and allows compatibility between codecs, chips and FIR filters. The FIR filter performs signal processing tasks on the input data, enhancing its quality or extracting relevant information.

Overall, this design scheme optimizes audio signal processing in the FPGA environment, enabling versatility and efficiency.

- **Discussion:**

The s2p adapter serves as a bridge and receives serial data from the codec chip, which communicates with external audio components. This adapter transforms the serial data from the codec into a parallel format that the FIR filter may use. Redirected through the S2P adapter, the improved parallel output from the FIR filter (which modifies the audio signal by applying predetermined coefficients) is the end product. After that, the adapter re-serializes the data and sends it to the codec so that it can be output to the external audio devices.

Because of the configuration that the codec initializer has made for the chip, this setup guarantees seamless data handling across all formats and stages. The system's operation depends on each component, which together improves audio quality and permits effective signal processing: the codec, s2p adaptor, and FIR filter. This flexible configuration optimises the environment for manipulating audio signals by utilising the capabilities of the FPGA.

6. PHYSICAL EXPERIMENTS

- **Codec initialization error:**

In the hardware testing phase, the initial process for validation is codec initialization. In this process, an error was encountered where the codec chip was not being initialized.
Source of the error: To confirm the problem was in codec initialization part, I shorted the input and output (AUD_ADCDAT and AUD_DACDAT) and loaded the program to the development board, this confirmed the problem was in codec initialization.

- **Debugging**: To find the problem, I wrote a testbench to display the output waveform, and noticed that the start and stop bits were interchanged. I corrected the program accordingly and this codec initialization was successful.

- **S2p adaptor data loss error:**

During the test cycle, the parallel data output and input of the s2p adaptor were shorted. Since no processing was occurring on the data, the input received from the codec must directly be read as the output, but in my case some parts of the data were clipped off.

- **Debugging**: I ran the testbench on the shorted s2p adaptor, then noticed that strobe signals were triggered five clock cycles earlier than expected and that was causing some parts of data to be clipped off. Once the necessary changes were made to the code and validated the output on the simulator, the hardware was giving the expected output.

7. DISCUSSION OF THE SYSTEM AS A WHOLE

A few observations were noticed personally during real-world testing with the FPGA-based signal processing system, highlighting its intricate workings and essential parts:

One crucial finding was that **codec initialization** is an essential stage. Notably, a start and stop bit inversion problem was found that was impeding effective codec communication. Reversing this inversion was a crucial step that highlighted how important it is to follow protocol specifications to the letter when initialising the codec. The **s2p adapter** turned out to be an essential bridge, guaranteeing data integrity while converting from parallel to serial formats and vice versa. There was a specific problem that caused data loss since the strobe signals were being activated too soon. Solving this problem demonstrated how crucial timing accuracy is to synchronising data transfers, highlighting the significance of the s2p adaptor in the system. Understanding the **FIR filter's** operation revealed a function beyond frequency modification. The audio output was greatly affected by small changes to the coefficients of the FIR filter, which turned out to be a critical component in improving audio quality. This demonstrated that attaining the intended signal processing results depends critically on the filter's design and coefficient choices.

During these real-world trials, the system showed remarkable accuracy in audio data conversion and processing, from the first phases of codec initialization to the fine-grained FIR filter filtering of audio signals. These findings demonstrate a hands-on approach that synchronises theoretical expectations with real-world functionality. The thorough validation procedure and careful design are essential.

CONCLUSION

In conclusion, the development of all systems including codec chip, s2p adapter, FIR filter proved a successful combination of hardware components for high performance audio signal processing in FPGA-based platform and effectiveness of the certification cycle.
Lessons learned from this exercise include the importance of a validation process of adequate specifications, the importance of a clear communication mechanism between system components, and the benefits of having it existing hardware modules and standards will be used for easy integration.
Future improvements of this work include exploring advanced signal processing algorithm techniques, extending resource utilization and efficiency, and extending system capabilities to support additional audio processing functions such as noise reduction, equality, and dynamic range compression In addition, external -Integration with interfaces and communication systems can enable compatibility and performance with external audio devices and systems.

Appendix

The Below figures show the procedure to start the Project

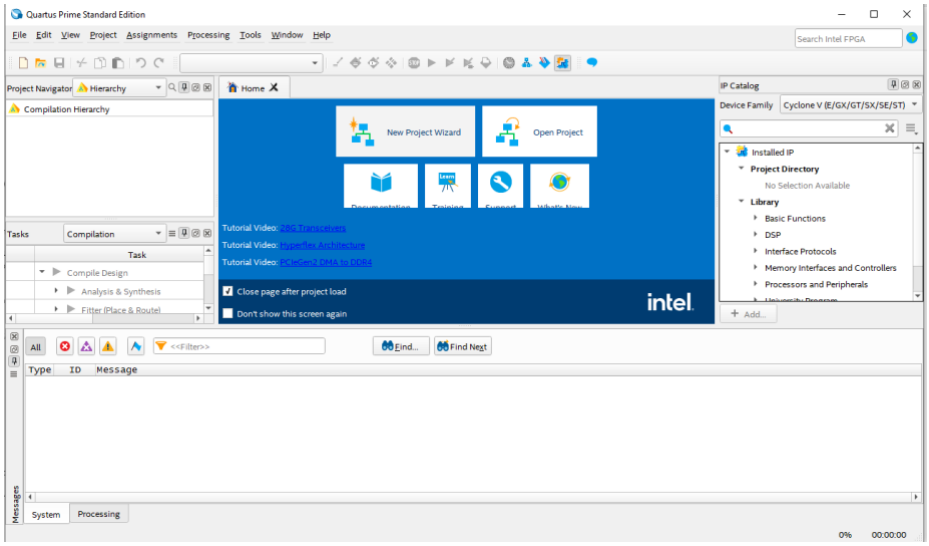


Fig.1: Creating New Project

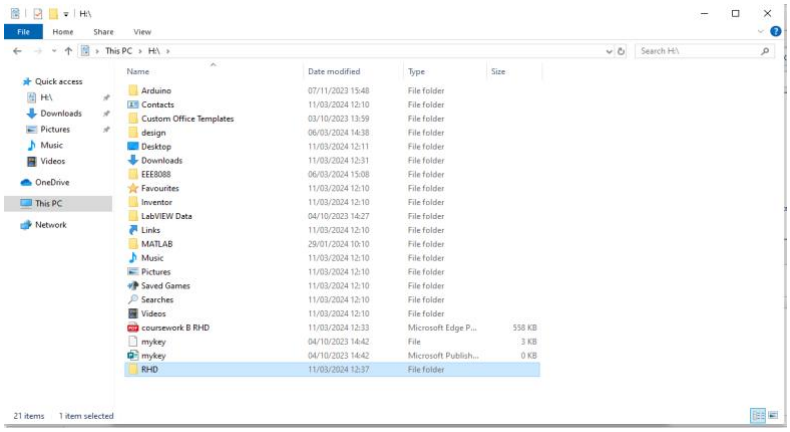


Fig2: Creating new path in Drive H:/

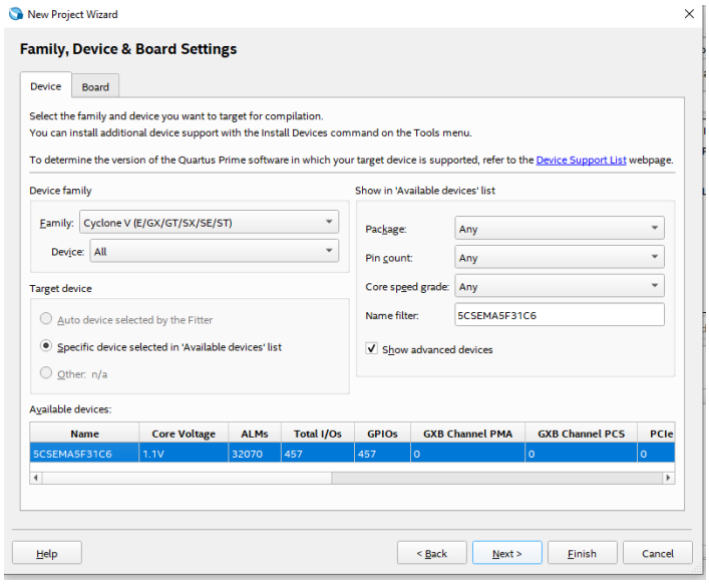


Fig3: Selecting Filter Cyclone V: 5CSEMA5F31C6

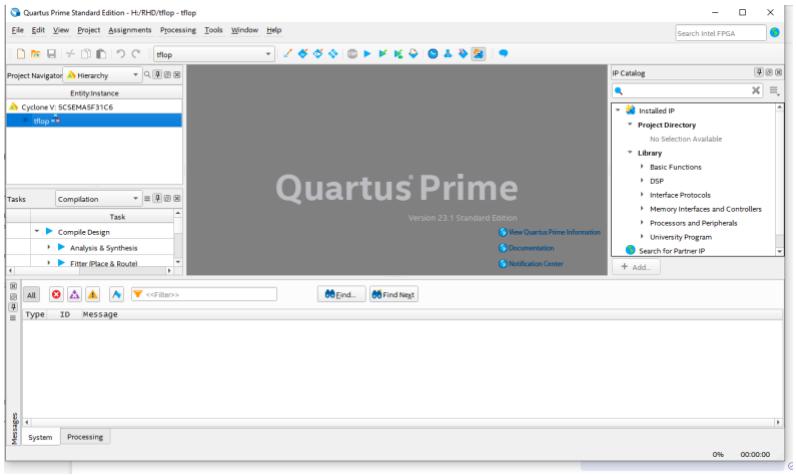


Fig 4: Write VHD code

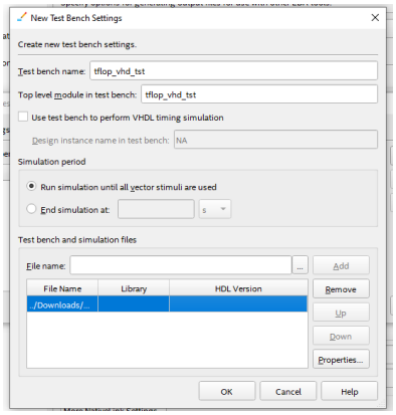


Fig5: Adding Test Bench file

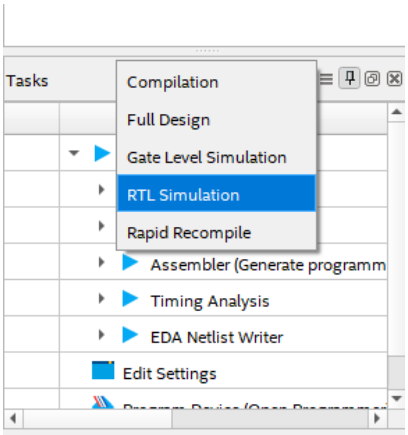


Fig 6 : RTL simulation

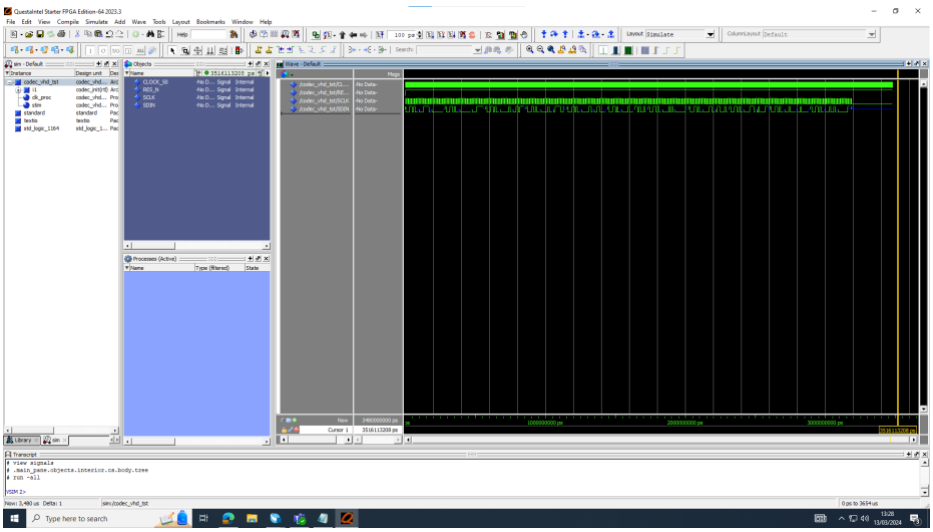


Fig 7 : Simulation obtained after RTL Simulation (The output of Codec)

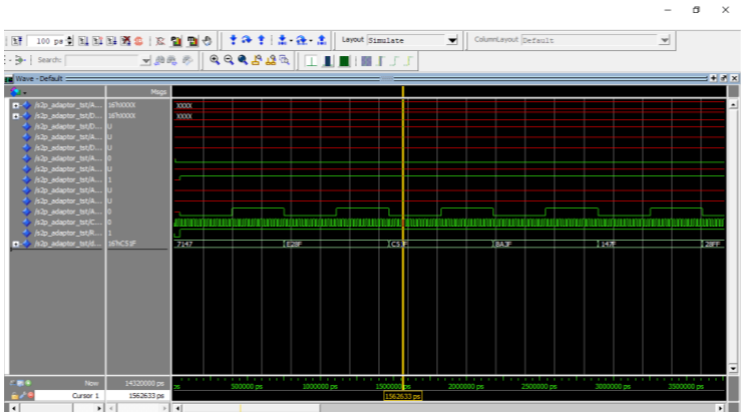


Fig 8 : Input Channel Test Output

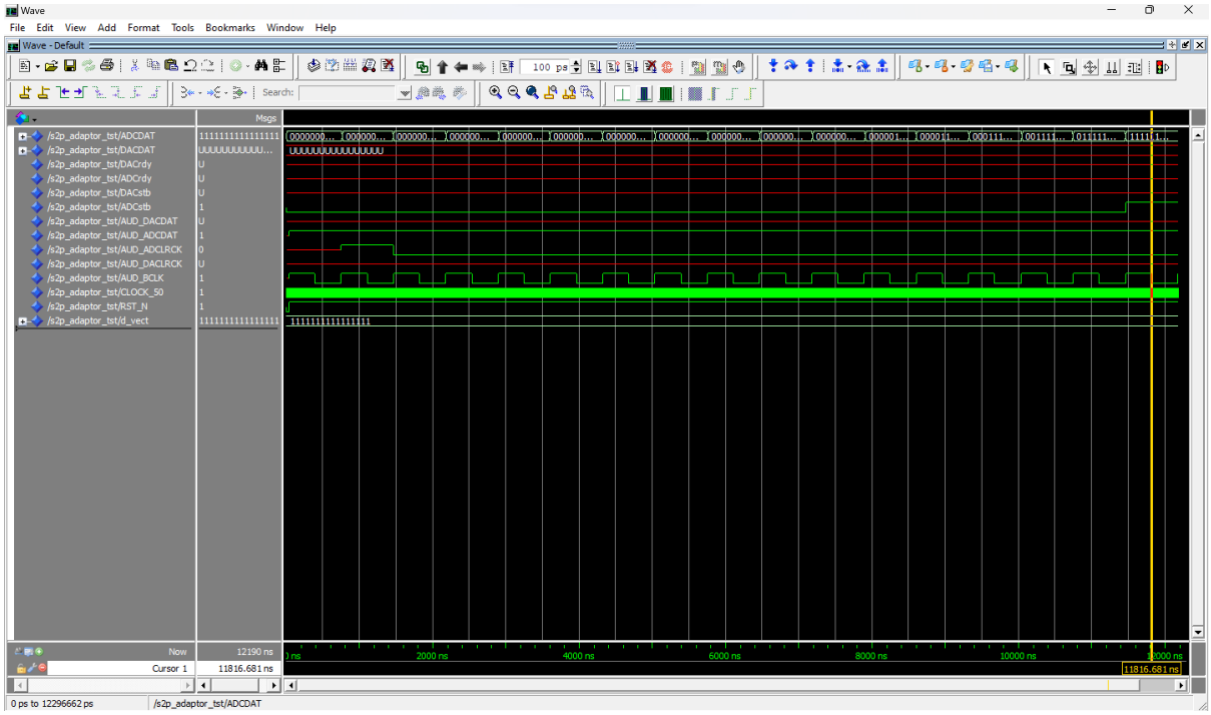


Fig 9 : Output Channel of S2P adapter Test output.

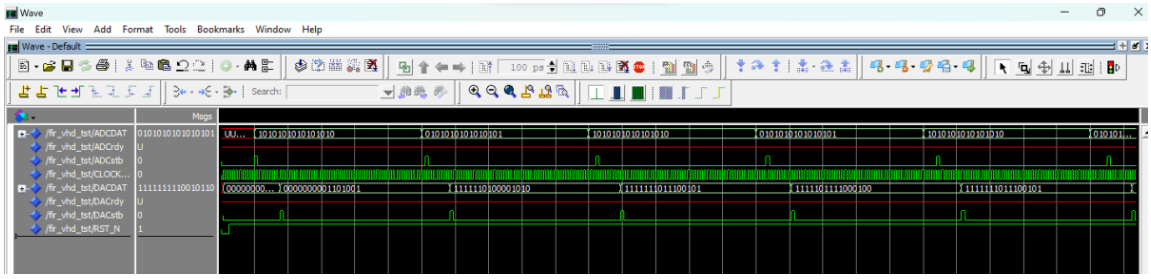


Fig 10: FIR Filter Validation

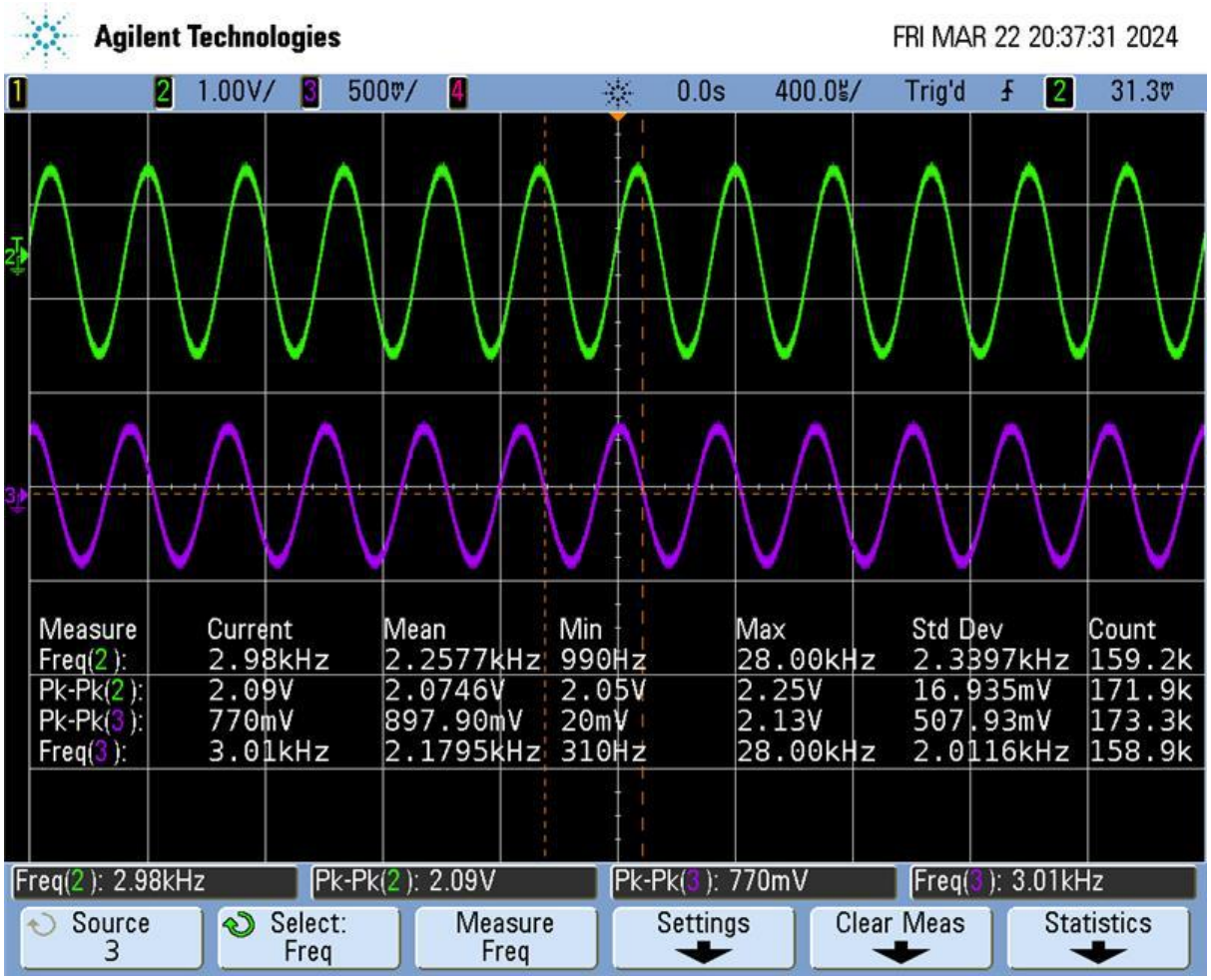


Fig 13 : Input and Output at 2.9kHz

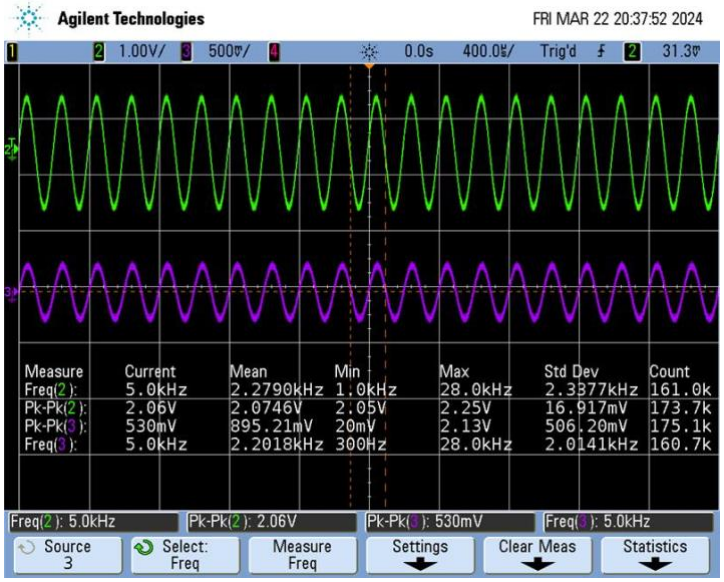


Fig 14 : Input and Output at 5kHz

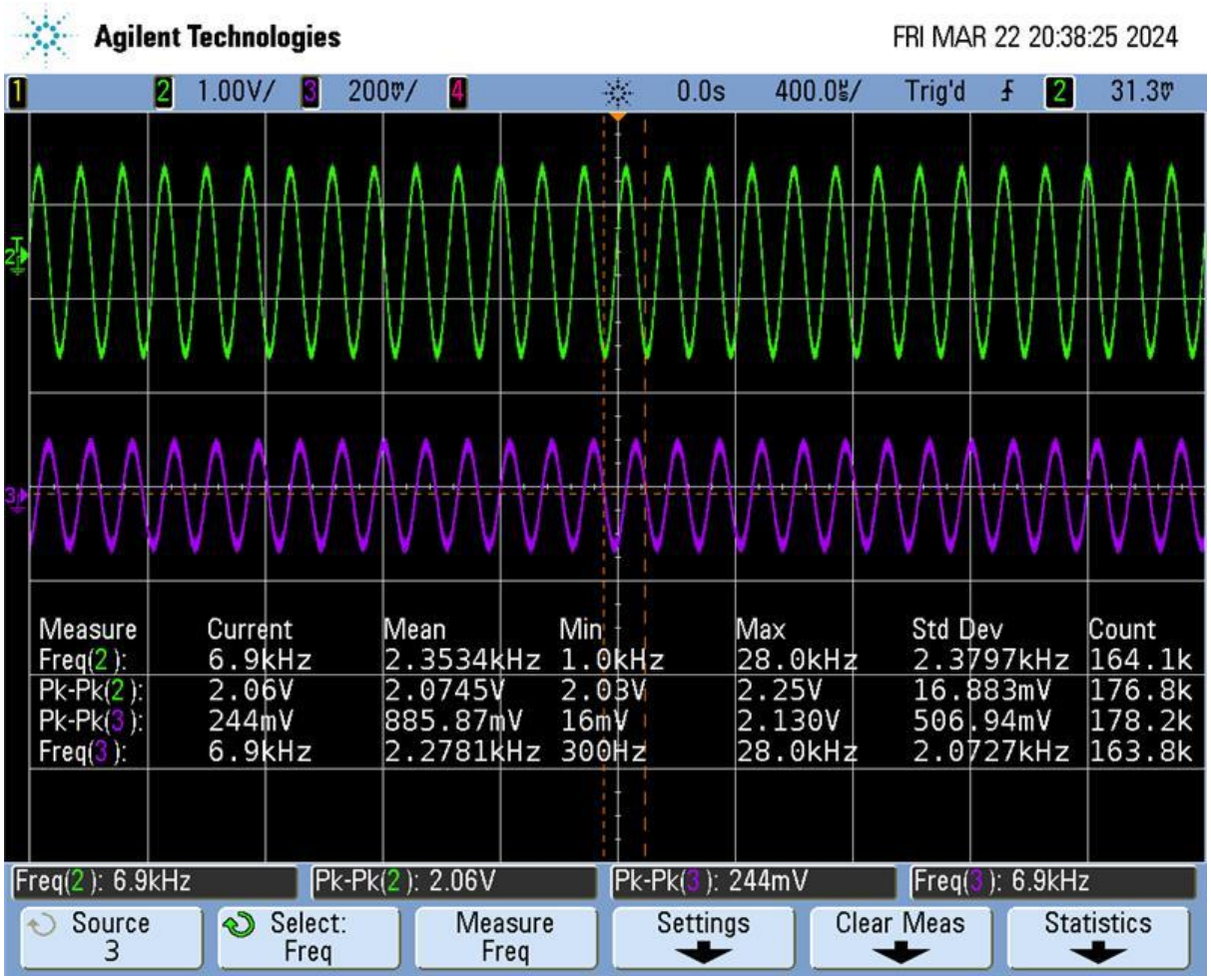


Fig 15: Input and Output at 7kHz.

CODES:

1. Codec VHDL Code:

```
library ieee;
use ieee.std_logic_1164.all;

entity codec_init is
	port
	(
		CLOCK_50	: in  std_logic; -- master clock
		RES_N		: in  std_logic; -- reset, active 0
		SCLK		: out std_logic;	-- serial clock
		SDIN		: out std_logic		-- serial data
	);

end entity;


architecture rtl of codec_init is

constant sdin_load : std_logic_vector (11*24-1 downto 0) :=
b"0011010_0_0001111_000000000"&
b"0011010_0_0000000_000011111"&
b"0011010_0_0000001_000110111"&
b"0011010_0_0000010_001111001"&
b"0011010_0_0000011_000110000"&
b"0011010_0_0000100_011010010"&
b"0011010_0_0000101_000000001"&
b"0011010_0_0000110_001100010"&
b"0011010_0_0000111_001000011"&
b"0011010_0_0001000_000100000"&
b"0011010_0_0001001_000000001";
-- 11 words, the first is reset (R15), the others are registers R0-9.
-- each word is 24 bit constructed as
-- chip address, r/w bit, reg address, reg data
-- these words do not include start, stop and ack bits, see packet format below


-- Packet format:			(bit number)

---- start bit				28
-- 7 bits chip address,
-- 1 r/w bit,
--** ack				19
-- 8 high bits of reg. data,
--** ack				10
-- 8 low bits of reg. data,
--** ack				1
---- stop bit				0
```

```
-- reg. data = 7 bit address + 9 bit config data, 16 bits total,
-- split as 8+8 bits in the packet, MSB go first.


-- declare a shift register
signal sr: std_logic_vector(11*24-1 downto 0);
-- declare an internal signal to be copied into SIDN

-- declare the bit counter;  -- bit counter, runs at 100kHz,
signal bit_cnt: integer range 0 to 28 := 28;
-- bits 28, 19, 10, 1 and 0 are special
-- declare the word counter; -- word counter, runs at about 5kHz
signal word_cnt: integer range 0 to 10 := 10;

-- declare the counter for the bit length; -- frequency divider counter,
-- runs at 50MHz
signal f_div_cnt: integer range 0 to 499 := 499;
signal s_sdin: std_logic;
signal s_sclk: std_logic;


begin

	process (CLOCK_50)
	begin
		if (rising_edge(CLOCK_50)) then
		----------------------------
		-- reset actions
			if (RES_N = '0') then
			-- reset the counters to an appropriate state
					-- load the frequency divider,
					-- 50MHz/500=100kHz bus speed
					-- load the shift register
					-- load the bit counter,
					-- 29 bits in the word protocol
					-- load the word counter, 11 words
			-- reset the outputs to an appropriate state
				f_div_cnt <= 499;
				bit_cnt <= 28;
				word_cnt <= 10;
				sr <= sdin_load;
				s_sclk <= '0';
				s_sdin <= '1';


			elsif (f_div_cnt = 0 and bit_cnt = 0 and word_cnt = 0) then -- deadlock
in the end
```

```
                        -- do nothing, wait for the next reset
                        s_sdin <= 'Z';
                        s_Sclk <= 'Z';

                -- modify reference counters
                -- for frequency divider, bits and words
                        elsif (f_div_cnt = 0) then -- at the end of each bit
                                f_div_cnt <= 499; -- reload the frequency divider counter

                                if (bit_cnt = 0) then -- at the end of each word
                                        bit_cnt <= 28; -- reset the bit counter
                                        word_cnt <= word_cnt-1;       --modify the word counter
                                else      -- the bit is not the end of a word
                                        bit_cnt <= bit_cnt-1;   --modify the bit counter
                                end if;

                        else      -- if not the end of the bit
                                f_div_cnt <= f_div_cnt-1; -- modify the frequency divider
                        end if;

                -- generating SCLK, it is going up and then down inside each bit
                        if (f_div_cnt < 375 and f_div_cnt > 124) then           -- condition when
SCLK goes up
                                s_sclk <= '1';
                        elsif ((f_div_cnt >= 0 and f_div_cnt < 125) or (f_div_cnt > 374 and
f_div_cnt < 500)) then          -- condition when SCLK goes down
                                s_sclk <= '0';
                        end if;

                -- generating serial data output
                        if (f_div_cnt = 250 and bit_cnt = 28) then -- start transition condition
                                s_sdin <= '0';
                        elsif (bit_cnt = 19 or bit_cnt = 10 or bit_cnt = 1) then -- ack bit
condition
                                s_sdin <= 'Z';
                        elsif (bit_cnt = 0 and f_div_cnt=499) then -- stop transition condition
                                s_sdin <= '0';
                        elsif (bit_cnt = 0 and f_div_cnt=250) then -- stop transition condition
                                s_sdin <= '1';
                        elsif(bit_cnt /= 19 and bit_cnt /= 10 and bit_cnt /= 1 and bit_cnt /= 0
and bit_cnt /= 28 and f_div_cnt = 499) then -- condition for the non-special bits
                                s_sdin <= sr(263);
                                sr((263) downto 1) <= sr(262 downto 0); -- shifting
                        end if;

                ----------------------------
                end if;
        end process;
        SDIN <= s_sdin;
        SCLK <= s_sclk;
```

```
        -- forming the output with high impedance states for ack-s
        --SDIN <= 'Z' when (...condition for ack bits...)
        --                              else (sdout);

end rtl;
```

---

2. Codec Test Bench:

```
 LIBRARY ieee;
USE ieee.std_logic_1164.all;

 ENTITY codec_vhd_tst IS
END codec_vhd_tst;
ARCHITECTURE codec_arch OF codec_vhd_tst IS
-- constants
-- signals
SIGNAL CLOCK_50 : STD_LOGIC;
SIGNAL RES_N : STD_LOGIC;
SIGNAL SCLK : STD_LOGIC;
SIGNAL SDIN : STD_LOGIC;
COMPONENT codec_init
    PORT (
    CLOCK_50 : IN STD_LOGIC;
    RES_N : IN STD_LOGIC;
    SCLK : OUT STD_LOGIC;
    SDIN : OUT STD_LOGIC
    );
END COMPONENT;
BEGIN

    i1 : codec_init
        PORT MAP (
        -- list connections between master ports and signals
        CLOCK_50 => CLOCK_50,
        RES_N => RES_N,
        SCLK => SCLK,
        SDIN => SDIN
        );

    clk_proc : PROCESS
            variable i : integer;
        BEGIN -- code that executes only once
            for i in 1 to 12*29*500 loop -- specify here the length of the simulation run
              CLOCK_50 <= '0';
              wait for 10 ns;
              CLOCK_50 <= '1';
              wait for 10 ns;
              end loop;
```

```
            WAIT;
        END PROCESS;

    stim : PROCESS
        BEGIN
        RES_N <= '0';
        wait for 40 ns;
        RES_N <= '1';
        WAIT;
        END PROCESS;
END codec_arch;
```

---

3. S2P Adapter VHDL code:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity s2p_adaptor is
        port(
        --      Core Side - two parallel interfaces for input and output
                ADCDAT:             out     std_logic_vector(15 downto 0);
                DACDAT:             in      std_logic_vector(15 downto 0);
                DACrdy:     out     std_logic;
                ADCrdy:     in      std_logic;
                DACstb:     in      std_logic;
                ADCstb:     out     std_logic;
        --      Audio Side in MASTER mode
                AUD_DACDAT:         out     std_logic; -- serial data out
                AUD_ADCDAT:         in      std_logic; -- serial data in
                AUD_ADCLRCK:        in      std_logic; -- strobe for input
                AUD_DACLRCK:        in      std_logic; -- strobe for output
                AUD_BCLK:   in      std_logic; -- serial interface "clock"
        --      Control Signals
                CLOCK_50:   in      std_logic;
                RST_N:      in      std_logic
        );

end entity;

architecture rtl of s2p_adaptor is
--      Internal Signals
signal old_BCLK: std_logic;
--signal old_dacstb: std_logic;
signal bit_cnt: integer range -1 to 16;
```

```
signal bit_cnt_s: integer range -1 to 16;
--signal DACrdy_int: std_logic;
signal ADCstb_int: std_logic;
signal DACDAT_int: std_logic_vector(15 downto 0);
begin
	process (CLOCK_50)
	--variable enable_o: integer;
	begin
		if (rising_edge(CLOCK_50)) then
		-------------begin sync design----------------
		-- reset actions (synchronous)
			if (RST_N = '0') then
				ADCstb_int <= '0';
				bit_cnt <= -1;
				bit_cnt_s <= -1;
				DACDAT_int <= b"0000000000000000";
				--DACrdy_int <= '0';
			else
				old_BCLK <= AUD_BCLK; -- needed for change detection
on BCLK input
		-- input channel
				--old_dacstb <= DACstb;
				if (old_BCLK='0' and AUD_BCLK='1') then --rising edge
of AUD_BCLK
					if (AUD_ADCLRCK = '1') then -- condition for the
start of the protocol
						bit_cnt <= 14; -- load the bit counter
						ADCDAT(15) <= AUD_ADCDAT; -- read
the first bit og the packet
					elsif (bit_cnt >= 0) then -- condition for the data
bits of the left channel
						ADCDAT(bit_cnt) <= AUD_ADCDAT;--
input one bit
						bit_cnt <= bit_cnt - 1; -- advance the bit
counter
					end if;
					if (bit_cnt = 0) then     -- condition for the strobe
of ADC parallel interface
						ADCstb_int <= '1';
					end if;
				end if;
				if (bit_cnt = -1) then    -- ADCrdy = '1' -- condition to
drop the ADC strobe
					ADCstb_int <= '0';
				end if;

		-- output channel
```

```
                                --if (DACstb = '1') then -- start condition
                                --      DACrdy <= '1';
                                --      bit_cnt_o <= 1;
                                --      AUD_DACDAT <= DACDAT(0);
                                --elsif (old_BCLK='0' and AUD_BCLK='1'and bit_cnt_o <
16) then -- each following falling edge
                                --      AUD_DACDAT <= DACDAT(bit_cnt_o); -- produce
DAC serial data bit
                                --      bit_cnt_o <= bit_cnt_o + 1;
                                --end if;
                                --if (...) then -- condition for loading DAC parallel
register
                                --      ...;
                                --end if;
                                if (DACstb = '1') then -- start condition
                                        --enable_o <= 1;
                                        --bit_cnt_s <= 0;
                                        DACDAT_int <= DACDAT;
                                end if;
                                if (old_BCLK='1' and AUD_BCLK='0') then
                                        if (AUD_DACLRCK = '1') then -- to begin serial
transfer
                                                bit_cnt_s <= 14;
                                                AUD_DACDAT <= DACDAT_int(15);
                                        elsif (bit_cnt_s >= 0) then
                                                AUD_DACDAT <= DACDAT_int(bit_cnt_s);
                                                bit_cnt_s <= bit_cnt_s - 1;
                                        end if;
                                        --if (bit_cnt_s = 0) then
                                                --DACrdy_int <= '0';
                                        --      enable_o <= 0;
                                        --end if;
                                end if;

                        end if;
                ----------end sync design----------
                end if;
        end process;
        ADCstb <= ADCstb_int;
end rtl;
```

4. **S2P Adapter Test Bench:**

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY s2p_adaptor_tst IS
END s2p_adaptor_tst;

ARCHITECTURE s2p_adaptor_arch OF s2p_adaptor_tst IS

-- Signals
SIGNAL ADCDAT, DACDAT: STD_LOGIC_VECTOR(15 downto 0);
SIGNAL DACrdy, ADCrdy, DACstb, ADCstb: STD_LOGIC;
SIGNAL AUD_DACDAT, AUD_ADCDAT, AUD_ADCLRCK, AUD_DACLRCK, AUD_BCLK:
STD_LOGIC;
SIGNAL CLOCK_50, RST_N: STD_LOGIC;
SIGNAL d_vect: std_logic_vector(15 downto 0) := "1001101011110010";
--signal lrc_cnt: integer := 1;
--signal b_cnt: integer := -1;
--signal clk_cnt: integer := 0;
signal DACDAT_int: std_logic_vector(15 downto 0);
--signal DACstb_int: std_logic;

COMPONENT s2p_adaptor
  PORT (
    ADCDAT: OUT STD_LOGIC_VECTOR(15 downto 0);
    DACDAT: IN STD_LOGIC_VECTOR(15 downto 0);
    DACrdy: OUT STD_LOGIC;
    ADCrdy: IN STD_LOGIC;
    DACstb: IN STD_LOGIC;
    ADCstb: OUT STD_LOGIC;
    AUD_DACDAT: OUT STD_LOGIC;
    AUD_ADCDAT: IN STD_LOGIC;
    AUD_ADCLRCK: IN STD_LOGIC;
    AUD_DACLRCK: IN STD_LOGIC;
    AUD_BCLK: IN STD_LOGIC;
    CLOCK_50: IN STD_LOGIC;
    RST_N: IN STD_LOGIC
  );
END COMPONENT;

BEGIN
  i1: s2p_adaptor
    PORT MAP (
      ADCDAT => ADCDAT,
      DACDAT => DACDAT,
      DACrdy => DACrdy,
      ADCrdy => ADCrdy,
      DACstb => DACstb,
      ADCstb => ADCstb,
      AUD_DACDAT => AUD_DACDAT,
```

```
      AUD_ADCDAT => AUD_ADCDAT,
      AUD_ADCLRCK => AUD_ADCLRCK,
      AUD_DACLRCK => AUD_DACLRCK,
      AUD_BCLK => AUD_BCLK,
      CLOCK_50 => CLOCK_50,
      RST_N => RST_N
    );

 clk_proc: PROCESS  -- master clock (50MHz) and reset
  BEGIN
    for i in 1 to 5600 loop
      CLOCK_50 <= '0';
      wait for 10 ns;
      CLOCK_50 <= '1';
      wait for 10 ns;
      if (i = 1) then
        RST_N <= '0';
      elsif (i = 3) then
       RST_N <= '1';
      end if;
    end loop;
    WAIT;
  END PROCESS;

clk_proc_t: PROCESS  -- AUD_BCLK
  BEGIN
  --variable j: integer;
    for i in 1 to 160 loop
      AUD_BCLK <= '0';
      wait for 357 ns;
      AUD_BCLK <= '1';
      wait for 357 ns;
    end loop;
  WAIT;
  END PROCESS;

lrck_sgnl: PROCESS  -- LRC signals
  BEGIN
  AUD_ADCLRCK <= '0';
  AUD_DACLRCK <= '0';
    wait until rising_edge(AUD_BCLK);
    wait until rising_edge(AUD_BCLK);
    for i in 1 to 10 loop
      wait until rising_edge(AUD_BCLK);
      AUD_ADCLRCK <= '1';
      AUD_DACLRCK <= '1';
      wait until rising_edge(AUD_BCLK);
      AUD_ADCLRCK <= '0';
      AUD_DACLRCK <= '0';
```

```
            for j in 1 to 25 loop
             wait until rising_edge(AUD_BCLK);
            end loop;
          end loop;
        WAIT;
        END PROCESS;



 input_data: PROCESS
  BEGIN
    DACstb <= '0';
    for i in 1 to 10 loop
      wait until rising_edge(AUD_ADCLRCK);
      for j in 0 to 15 loop
        AUD_ADCDAT <= d_vect(15-j);
       DACDAT(15-j) <= d_vect(15-j);
       wait until rising_edge(AUD_BCLK);
       if (j = 14) then
         wait until rising_edge(CLOCK_50);
         DACstb <= '1';
         wait until rising_edge(CLOCK_50);
          DACstb <= '0';
       end if;
      end loop;
    end loop;
  WAIT;
  END PROCESS;

  --stim: PROCESS
  --BEGIN
  --  DACDAT_int <= ADCDAT;
  --  DACstb_int <= ADCstb;
  --  wait;
  --END PROCESS;
  --DACDAT <= DACDAT_int;
END s2p_adaptor_arch;
```

---

5. **FIR Filter VHDL Code:**

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


entity fir is
port(

-- Input and output

        ADCDAT: in std_logic_vector(15 downto 0);
        DACDAT: out std_logic_vector(15 downto 0);
        DACrdy: in std_logic;
        ADCrdy: out std_logic;
        DACstb: out std_logic;
        ADCstb: in std_logic;

-- Control Signals
        CLOCK_50: in std_logic;
        RST_N: in std_logic
        );
        end entity;


architecture rtl of fir is

-- Internal Signals

signal old_ADCstb: std_logic;
signal tap_cnt: integer range 7 downto -1;
type tap_shift is array (7 downto 0) of signed (15 downto 0);
signal x: tap_shift;
signal accumulator: signed (34 downto 0);
signal DACstb_int: std_logic;

-- Coefficients for the Filter
constant b: tap_shift := (
to_signed(integer(-1260),16),
to_signed(integer(7827),16),
to_signed(integer(12471),16),
to_signed(integer(16384),16),
to_signed(integer(16384),16),
to_signed(integer(12471),16),
to_signed(integer(7827),16),
```

```
to_signed(integer(-1260),16));

begin

process (CLOCK_50)
	variable i: integer;
	variable lame:integer;
	begin

		if (rising_edge(CLOCK_50)) then -- for each rising edge of clock 50MHz
			if (RST_N = '0') then -- Reset operation
				tap_cnt <= -1;
				DACDAT <= b"0000000000000000";
				--ADCrdy <= '1';
				DACstb_int <= '0';
				--x <= (others => (others => 0));
				for i in 0 to 7 loop
					x(i) <= b"0000000000000000";
				end loop;

			else
				old_ADCstb <= ADCstb;
				if (ADCstb = '1') then -- Start of input
					--ADCrdy <= '0';
					x(7) <= signed(ADCDAT); -- Input data register is
moved to 7th element of array
					x(6 downto 0) <= x(7 downto 1); -- shifting the
array of registers
					tap_cnt <= 6;
					--DACstb <= '0';
					--lame:= 0;
					accumulator <= resize(b(7) * signed
(ADCDAT),35); -- would not take in 32 bits instead of 35bits
				elsif (tap_cnt >= 0) then -- Time multiplexing the
multiplier
					accumulator <= signed(accumulator + (
b(tap_cnt)*x(tap_cnt)));
					tap_cnt <= tap_cnt - 1;
					--if (tap_cnt = 6) then
					--	accumulator <= signed(accumulator + (
b(tap_cnt)*x(tap_cnt)));
					--	tap_cnt <= tap_cnt - 1;
					--elsif (tap_cnt = 5) then
					--	accumulator <= signed(accumulator + (
b(tap_cnt)*x(tap_cnt)));
					--	tap_cnt <= tap_cnt - 1;
					--elsif (tap_cnt = 4) then
```

```
                                        --      accumulator <= signed(accumulator + (
b(tap_cnt)*x(tap_cnt)));
                                        --      tap_cnt <= tap_cnt - 1;
                                        --elsif (tap_cnt = 3) then
                                        --      accumulator <= signed(accumulator + (
b(tap_cnt)*x(tap_cnt)));
                                        --      tap_cnt <= tap_cnt - 1;
                                        --elsif (tap_cnt = 2) then
                                        --      accumulator <= signed(accumulator + (
b(tap_cnt)*x(tap_cnt)));
                                        --      tap_cnt <= tap_cnt - 1;
                                        --elsif (tap_cnt = 1) then
                                        --      accumulator <= signed(accumulator + (
b(tap_cnt)*x(tap_cnt)));
                                        --      tap_cnt <= tap_cnt - 1;
                                        if (tap_cnt = 0) then
                                                --accumulator <= signed(accumulator + (
b(tap_cnt)*x(tap_cnt)));
                                                DACstb_int <= '1';
                                                --lame := 1;
                                                --ADCrdy <= '1';
                                                DACDAT <=
std_logic_vector(signed(accumulator(34) & accumulator(32 downto 18))); -- type
conversion to std_logic_vector
                                                --tap_cnt <= -1;
                                        end if;
                                end if;

                                if (tap_cnt = -1)then
                                        DACstb_int <= '0';
                                        --tap_cnt <= 0;
                                end if;
                        end if;
                end if;

        end process;
        DACstb <= DACstb_int;
end rtl;
```

6. **FIR Filter Test bench :**

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY fir_vhd_tst IS
END fir_vhd_tst;

ARCHITECTURE fir_arch OF fir_vhd_tst IS

-- constants
-- signals

SIGNAL ADCDAT : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL ADCrdy : STD_LOGIC;
SIGNAL ADCstb : STD_LOGIC;
SIGNAL CLOCK_50 : STD_LOGIC;
SIGNAL DACDAT : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL DACrdy : STD_LOGIC;
SIGNAL DACstb : STD_LOGIC;
SIGNAL RST_N : STD_LOGIC;
COMPONENT fir

PORT (

ADCDAT : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
ADCrdy : OUT STD_LOGIC;
ADCstb : IN STD_LOGIC;
CLOCK_50 : IN STD_LOGIC;
DACDAT : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
DACrdy : IN STD_LOGIC;
DACstb : OUT STD_LOGIC;
RST_N : IN STD_LOGIC
);
END COMPONENT;
BEGIN

i1 : fir
PORT MAP (

-- list connections between master ports and signals
ADCDAT => ADCDAT,
ADCrdy => ADCrdy,
ADCstb => ADCstb,
CLOCK_50 => CLOCK_50,
DACDAT => DACDAT,
DACrdy => DACrdy,
DACstb => DACstb,
RST_N => RST_N
);
```

```
        -- Clock 50 MHz
        clock : PROCESS
        variable i : integer;
        BEGIN
        for i in 1 to 160000 loop
        CLOCK_50 <= '0';
        wait for 10 ns;
        CLOCK_50 <= '1';
        wait for 10 ns;
        end loop;

        WAIT;
        END PROCESS;
        --Reset goes high once and then remain low
        stim : PROCESS
        BEGIN
        RST_N <= '0';
        wait for 45 ns;
        RST_N <= '1';



WAIT; -- do not repeat once finished
END PROCESS;
-- ADCDAT from s2p is manually provided
ADCdata: PROCESS
variable p: integer;
BEGIN
wait for 200 ns;
for p in 1 to 8 loop
ADCDAT <= b"1010101010101010";
wait for 1000 ns;
ADCDAT <= b"0101010101010101";
wait for 1000 ns;
end loop;
END PROCESS;

ADCstrobe: PROCESS
variable o: integer;
BEGIN
ADCstb <= '0';
wait for 200 ns;
for o in 1 to 16 loop
ADCstb <= '1';
wait for 20 ns;
ADCstb <= '0';
wait for 1000 ns;
end loop;
```

```
WAIT;
END PROCESS;
END fir_arch;
```