UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

DAVI NACHTIGALL LAZZAROTTO

DISPOSITIVO DE ÁUDIO MULTI-CANAL EM FPGA

Porto Alegre
2019

DAVI NACHTIGALL LAZZAROTTO

DISPOSITIVO DE ÁUDIO MULTI-CANAL EM FPGA

Projeto de Diplomação apresentado ao Departamento de Engenharia Elétrica da Universidade Federal do Rio Grande do Sul como parte dos requisitos para a obtenção do título de Engenheiro Eletricista.

ORIENTADOR: Prof. Dr. Altamiro A. Susin

Porto Alegre
2019

DAVI NACHTIGALL LAZZAROTTO

DISPOSITIVO DE ÁUDIO MULTI-CANAL EM FPGA

Este Projeto foi julgado adequado para a obtenção dos créditos da Disciplina Projeto de Diplomação do Departamento de Engenharia Elétrica e aprovado em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____
Prof. Dr. Altamiro A. Susin, UFRGS
Doutor pelo Institut National Polytechnique de Grenoble, França

Banca Examinadora:

Prof. Dr. Altamiro Amadeu Susin, UFRGS
Doutor pelo Institut National Polytechnique de Grenoble, França

Prof. Dr. Ronaldo Husemann, UFRGS
Doutor pela Universidade Federal do Rio Grande do Sul - Porto Alegre, Brasil

Dr. Gustavo Ilha, CEITEC
Doutor pela Universidade Federal do Rio Grande do Sul - Porto Alegre, Brasil

Porto Alegre, dezembro de 2019.

RESUMO

Algoritmos de localização de emissores de som são utilizados em diversas aplicações, tais como *smart room*, video conferências, aplicações militares e dispositivos médicos de ultrassom. Estes algoritmos necessitam da aquisição de som em múltiplos pontos simultaneamente, o que é feito com matrizes de microfones. Uma possibilidade nestas matrizes é a utilização de microfones MEMS (*Microelectrical Mechanical Systems*) com saída digital PDM (*Pulse Density Modulation*).

O dispositivo desenvolvido neste projeto e apresentado no presente documento é uma interface para uma matriz de seis microfones PDM baseado em FPGA com conectividade USB. Este dispositivo transforma o sinal PDM proveniente dos microfones em PCM (*Pulse-Coded Modulation*), por ser o formato de áudio mais utilizado atualmente e por ser mais facilmente manipulável. O sistema foi projetado para enviar os sinais PCM através da interface USB de forma sincronizada.

Para a implementação do projeto, foi utilizado um kit FPGA Xilinx XUPV5. O kit contém uma FPGA Virtex-5, que foi utilizada para a transcodificação PDM-PCM do sinal. Já o protocolo USB foi implementado pelo controlador USB Cypress *Ez-Host*, também presente no kit. A comunicação entre a FPGA e o controlador USB foi feita utilizando o protocolo HPI (*Host Port Interface*).

Como resultado, observou-se, através do analisador lógico TLA6402, que a transcodificação permitiu a obtenção dos sinais PCM relativos aos seis microfones. Foram também realizados testes com a interface AC97 do kit Xilinx XUPV5, a qual transformou o sinal PCM em analógico. Este sinal analógico foi direcionado a um alto-falante que permitiu escutar a saída dos microfones. Notou-se que o sinal escutado era condizente com o esperado, mas com a adição de ruído. Foi realizada também a transmissão USB dos sinais de áudio, que permitiu recuperá-los de forma sincronizada em um *host* USB, porém apenas com dois canais.

Palavras-chave: Audio multi-canal, matriz de microfones, PDM, USB.

ABSTRACT

Sound source localization algorithms are used in many applications, such as smart room, video conferencing, military applications and medical ultrasound devices. These algorithms require sound acquisition at multiple points simultaneously, which is done with microphone arrays. One type of microphone used in these arrays are the Microelectrical Mechanical Systems (MEMS) with Pulse Density Modulation (PDM) digital output.

The device developed in this project and presented in this document is a FPGA-based interface for an array of six PDM microphones with USB connectivity. This device transforms the incoming PDM signal to PCM (Pulse-Coded Modulation), as it is the most widely used audio format and is more easily manipulated. The system is designed to send the PCM signals through the USB interface synchronously.

For the project implementation, an FPGA Xilinx XUPV5 kit was used. The kit contains a Virtex-5 FPGA, which was used for the PDM-PCM transcoding. The USB protocol was implemented by the Cypress Ez-Host USB controller, also available in the kit. Communication between the FPGA and the USB controller was done using the HPI (Host Port Interface) protocol.

As a result, it was observed through the TLA6402 logic analyzer that the transcoding allowed obtaining the PCM signals for the six microphones. Tests were also performed with the AC97 interface of the Xilinx XUPV5 kit, which transforms the PCM signal into analog. This analog signal was directed to a speaker that allowed to hear the microphone output. It was noted that the audio signal was consistent with the expected, but with the addition of noise. The USB transmission of the audio signals was also performed, which allowed them to be retrieved synchronously at a USB host, but only with two channels.

Keywords: multi-channel audio, microphone array, PDM, USB.

LISTA DE ILUSTRAÇÕES

Figura 1: Amostragem de um sinal com frequência acima da frequência de Nyquist. 22
Figura 2: Função de transferência de bloco quantizador. 23
Figura 3: Representação da modulação PCM de um sinal. 24
Figura 4: Modulação PDM de uma senoide. 25
Figura 5: Sinal modulado PDM no domínio frequência. 25
Figura 6: Diagrama de blocos de um modulador Sigma-Delta de primeira ordem. 26
Figura 7: Diagrama de blocos de um filtro FIR. 27
Figura 8: Diagrama de blocos de um integrador. 28
Figura 9: Diagrama de blocos de um filtro *comb*. 29
Figura 10: Diagrama de blocos de um filtro CIC. 29
Figura 11: Resposta em frequência de um filtro CIC. 31
Figura 12: Representação gráfica de um terminal de entrada. 35
Figura 13: Representação gráfica de um terminal de saída. 35
Figura 14: Representação gráfica de uma *feature unit*. 36
Figura 15: Placa de aquisição de áudio utilizada no projeto 38
Figura 16: Diagrama do ciclo do microfone PDM. 43
Figura 17: Diagrama de blocos do filtro CIC 46
Figura 18: Resposta em frequência do filtro *halfband* decimador representado em ponto flutuante. 48
Figura 19: Resposta em frequência do filtro *halfband* decimador representado com 8 bits. 49
Figura 20: Resposta em frequência do filtro *halfband* decimador representado com 12 bits. 49
Figura 21: Resposta em frequência do filtro *halfband* decimador representado com 16 bits. 50
Figura 22: Resposta em frequência do filtro *halfband* decimador representado com 20 bits. 50
Figura 23: Resposta em frequência do filtro *halfband* decimador representado com 24 bits. 51
Figura 24: Resposta ao impulso do filtro *halfband* decimador. 52
Figura 25: Topologia do modulador sigma-delta simulado. 55
Figura 26: Sinal de saída do modulador sigma delta simulado com $k = 1$. 56
Figura 27: Diagrama de blocos do controlador USB Cypress *Ez-Host*. 57
Figura 28: Modos de operação do Cypress *Ez-Host* definidos pelos pinos de *bootstrap*. 58

Figura 29: Representação da arquitetura em camadas do *firmware* do Cypress *Ez-Host*. 59
Figura 30: Diagrama de blocos do Cypress *Ez-Host* com barramento HPI. 60
Figura 31: Registradores dedicados ao protocolo HPI. 60
Figura 32: Registrador HPI STATUS. 61
Figura 33: Transação HPI de escrita de dado em memória. 62
Figura 34: Organização hierárquica dos descritores USB do dispositivo. 63
Figura 35: Representação gráfica da topologia da função áudio do dispositivo USB. 68
Figura 36: Conexão entre o controlador USB Cypress Ex-Host e a memória EEPROM no kit FPGA Xilinx XUPV5. 75
Figura 37: Organização das amostras de áudio na USB *frame*. 80
Figura 38: Máquina de estados implementada na interface HPI. 82
Figura 39: Representação do sinal modulado PDM. 85
Figura 40: Espectro Frequencial do sinal modulado PDM em baixas frequências. 86
Figura 41: Espectro Frequencial do sinal modulado PDM em altas frequências. . 87
Figura 42: Representação do sinal de saída do filtro CIC. 87
Figura 43: Espectro frequencial do sinal de saída do filtro CIC. 88
Figura 44: Sinal de saída do quinto integrador do filtro CIC. 88
Figura 45: Sinal de saída de cada filtro do transcodificador PDM-PCM. 89
Figura 46: Espectro frequencial da saída de cada filtro do transcodificador PDM-PCM. 90
Figura 47: Simulação VHDL do bloco de geração de *clock*. 90
Figura 48: Simulação VHDL do transcodificador PDM-PCM. 90
Figura 49: Setup de teste com analisador lógico. 91
Figura 50: Sinal de áudio gerado pelo sistema correspondente a quatro estalos de dedo. 92
Figura 51: Sinal de áudio gerado pelo sistema correspondente a voz humana. . . 92
Figura 52: Sinal de áudio gerado pelo sistema correspondente a uma senoide de 1kHz. 93
Figura 53: Setup de testes com alto-falante. 94
Figura 54: Simulação VHDL de uma operação de escrita em HPI. 95
Figura 55: Simulação VHDL de seis operações de escrita HPI. 95
Figura 56: Simulação VHDL de diversos ciclos de escrita HPI. 95
Figura 57: Reconhecimento do dispositivo USB por *host* Windows. 96
Figura 58: Reconhecimento do dispositivo USB como grupo de microfones. . . 97
Figura 59: Sinal de onda quadrada de dispositivo USB recebido no Audacity. . . 98
Figura 60: Sinal de áudio de dois microfones recebido via USB no Audacity. . . 99

LISTA DE TABELAS

Tabela 1: Tipos de transações USB. 33
Tabela 2: Tabela comparativa entre modos de funcionamento do microfone WM7236E. 39
Tabela 3: Valores dos tempos do ciclo do microfone PDM. 43
Tabela 4: Performance do filtro *halfband* decimador com diferentes resoluções em bits. 51
Tabela 5: Descritor de dispositivo utilizado no dispositivo do projeto. 65
Tabela 6: Constantes dos tipos de descritores 65
Tabela 7: Descritor de configuração utilizado no dispositivo do projeto. 66
Tabela 8: Descritor padrão da interface *AudioControl* utilizado no dispositivo do projeto. 67
Tabela 9: Constantes do tipo *Audio Interface Subclass*. 67
Tabela 10: Descritor específico de classe da interface *AudioControl* utilizado no dispositivo do projeto. 67
Tabela 11: Constantes do tipo "*Audio Class-Specific AC Interface Descriptor Subtypes*". 68
Tabela 12: Descritor do terminal de entrada utilizado no dispositivo do projeto. . 69
Tabela 13: Descritor de *feature unit* utilizado no dispositivo do projeto. 70
Tabela 14: Descritor do terminal de saída utilizado no dispositivo do projeto. . . 71
Tabela 15: Descritor padrão da interface *AudioStreaming* de *alternate setting* número zero utilizado no dispositivo do projeto. 71
Tabela 16: Descritor padrão da interface *AudioStreaming* de *alternate setting* número um utilizado no dispositivo do projeto. 72
Tabela 17: Descritor específico de classe da interface *AudioStreaming* utilizado no dispositivo do projeto. 72
Tabela 18: Descritor de tipo de formato utilizado no dispositivo do projeto. . . . 73
Tabela 19: Descritor padrão do *endpoint* utilizado no dispositivo do projeto. . . . 73
Tabela 20: Descritor específico de classe do *endpoint* utilizado no dispositivo do projeto. 74
Tabela 21: Sinais de controle HPI para cada estado da máquina de estados. . . . 83

LISTA DE ABREVIATURAS

ADPCM	Backward-Adaptative Differential Pulse-Code Modulation
BIOS	Basic Input/Output System
CIC	Cascaded Integrator-Comb
CD	Compact Disk
CRC	Cyclic Redundancy Check
DAC	Digital/Analog Converter
DCM	Digital Clock Manager
DE	Design Example
DMA	Direct Memory Access
DPCM	Differential Pulse-Code Modulation
DSD	Direct Stream Digital
DSP	Digital Signal Processor
EEPROM	Electrically Erasable Programmable Read-Only Memory
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
HID	Human Interface Device
HPI	Host Port Interface
HSS	High-Speed Serial
I^2S	Inter-IC Sound
I^2C	Inter-Integrated Circuit
LaPSI	Laboratório de Processamento de Sinal e Imagem
LCP	Link Command Protocol
LPCM	Linear Pulse-Code Modulation
LSB	Least Significant Bit
MEMS	Microelectronic Mechanical System
MIDI	Musical Instrument Digital Interface

MLP	Meridian Lossless Packing
MSB	Most Significant Bit
PC	Personal Computer
PCM	Pulse-Code Modulation
PDM	Pulse Density Modulation
PID	Packet ID
RAM	Random Access Memory
ROM	Read-Only Memory
SE	Simple Example
SNR	Signal-to-Noise Ratio
SOF	Start of Frame
SPI	Serial Peripheral Interface
UFRGS	Universidade Federal do Rio Grande do Sul
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language
ZOH	Zero-Order Hold

SUMÁRIO

1 INTRODUÇÃO 17

2 REVISÃO BIBLIOGRÁFICA 21
2.1 Sinais digitais 21
2.1.1 Decimação e Aliasing 21
2.1.2 PCM 22
2.1.3 PDM 24
2.1.4 Moduladores Sigma-Delta 26
2.1.5 Projeto de filtros FIR 27
2.1.6 Filtro CIC 28
2.1.7 Características frequenciais do filtro CIC 30
2.1.8 Design de filtros CIC 30
2.2 USB 31

3 SOLUÇÃO PROPOSTA E MATERIAIS 37
3.1 Visão geral 37
3.2 Módulo de transcodificação PDM-PCM 40
3.2.1 Gerador de *clock* 41
3.2.2 Sampler PDM 42
3.2.3 Arquitetura do transcodificador PDM-PCM 44
3.2.4 Projeto do filtro CIC 44
3.2.5 Projeto do filtro *halfband* decimador 47
3.2.6 Simulação em MATLAB 54
3.3 Módulo de comunicação 56
3.3.1 Controlador USB Cypress *Ez-Host* 57
3.3.2 Protocolo HPI 58
3.3.3 Projeto dos descritores USB 61
3.3.4 *Firmware* do controlador USB 75
3.3.5 Interface HPI em FPGA 80

4 RESULTADOS 85
4.1 Transcodificador PDM-PCM 85
4.1.1 Resultados da simulação em MATLAB 85
4.1.2 Resultados da simulação VHDL 89
4.1.3 Resultados da implementação 91
4.2 Sistema de comunicação 93
4.2.1 Resultados da interface HPI 93

4.2.2 Resultados da enumeração USB 95
4.3 Resultados do sistema completo 96

5 CONCLUSÕES 101

REFERÊNCIAS 103

ANEXO A COEFICIENTES DO FILTRO *HALFBAND* DECIMADOR. . . 107

1 INTRODUÇÃO

Os sistemas de aquisição de áudio estão presentes no dia a dia, em comunicações por celular, vídeos, rádio ou música. A necessidade de captar o som em um ponto para reproduzi-lo em outro momento ou outro lugar demanda a utilização de tais sistemas. As diferentes particularidades de cada aplicação levam a diversas formas de projetar e desenvolver dispositivos de aquisição e processamento de áudio.

Microfones são dispositivos que traduzem um sinal de pressão captado no ar em sinal elétrico [Lewis (2013)]. Um tipo especial de microfone chama-se MEMS (*MicroElectroMechanical Systems*), que utilizam transdutores em silício, os quais podem se basear nos princípios piezoelétrico, piezoresistivo e capacitivo [Ganji and Majlis (2004)]. Geralmente, tais dispositivos contêm pre-amplificadores, condicionadores de sinal e conversores analógico-digitais em um mesmo chip [Todorović et al. (2017)]. Microfones também podem ser classificados quanto ao seu formato do sinal de saída, que pode ser tanto analógico quanto digital.

Nos microfones analógicos, o sinal de saída é uma grandeza elétrica proporcional à pressão medida. Microfones digitais podem ainda ser classificados quanto ao formato do sinal de saída. Um formato de áudio largamente utilizado em microfones digitais é o PDM (*Pulse Density Modulation*) [Lewis (2013)]. Este formato representa um sinal analógico como uma densidade de valores 1 ou 0 no domínio digital. Os sinais PDM são geralmente amostrados a uma frequência próxima de 3 MHz, muito acima da banda base do sinal de áudio [Lewis (2013)]. Outro formato de sinal de saída de microfones digitais é o I^2S, que envia o sinal de áudio em sua banda base de frequência.

Microfones PDM são amplamente adotados em aparelhos celulares e outros dispositivos portáteis [Lewis (2012)]. O sinal PDM é gerado pelo microfone através de um modulador do tipo sigma-delta [Hegde (2010)], o qual é formado por um circuito interno com integradores e subtratores, contido no encapsulamento do próprio microfone. Em um sistema com microfones PDM, o sinal de saída do microfone é geralmente processado por um codec ou um DSP [Lewis (2013)], que implementa filtros de decimação para reduzir a frequência de amostragem para a banda base.

A evolução da tecnologia de microfones digitais é interessante para as novas aplicações que têm surgido. Nos anos recentes, pesquisadores e indústrias têm se interessado por sistemas de aquisição de áudio com múltiplos microfones para aplicações baseadas em voz humana. Exemplos de tecnologias que utilizam matrizes de microfones são algoritmos de localização de emissores de som e algoritmos de aprimoramento de sinais de fala [Reitbauer et al. (2012)].

Tais algoritmos podem ser aplicados em diferentes contextos. Aplicações do tipo smart room podem utilizar sistemas de detecção da posição de um emissor de som para automatizar os alto-falantes da sala [Gareta (2007)]. Outros tipos de sistemas podem mo-

ver automaticamente a direção de uma câmera para focar na pessoa que está falando [Mei et al. (2006)]. A localização de emissores de som encontra ainda encontra utilidade em aplicações militares, dispositivos médicos de ultrassom, monitoramento de residências, entre outros [Todorović et al. (2017)].

Para tais sistemas, a utilização de microfones PDM do tipo MEMS apresenta vantagens. O pequeno tamanho do transdutor e a simplicidade do modulador sigma-delta permite que esses dispositivos tenham encapsulamentos pequenos, frequentemente com dimensões de 3 mm x 4 mm x 1 mm [Lewis (2013)]. Isso permite que vários microfones sejam colocados em uma área pequena, o que é interessante em muitas aplicações com matrizes de microfones [Reitbauer et al. (2012)]. Além disso, o fato de o encapsulamento conter o transdutor e o modulador sigma-delta permite que a interface seja puramente digital[Todorović et al. (2017)].

No entanto, microfones PDM também apresentam desafios para essas aplicações. A maioria dos codecs utilizados como interface para tais microfones possui apenas dois canais de entrada de áudio PDM, não sendo suficientes para aplicações com matrizes de microfones. Já a utilização de DSP's pode não ser adequada para aplicações onde a sincronia entre os canais seja importante, uma vez que utiliza cálculos que são executadas de forma sequencial. Dessa forma, os sinais dos diferentes canais não são processados ao mesmo tempo, gerando um atraso entre eles e perdendo a sincronia do áudio.

Uma alternativa a esses métodos é a utilização de FPGAs (*Field Programmable Gate Array*) como interface com os microfones PDM [Todorović et al. (2017)]. Esses dispositivos permitem a criação de uma arquitetura de *hardware* feita sob medida para a aplicação, permitindo um maior número de canais e com processamento paralelo dos sinais, garantindo a sincronia entre os canais de áudio.

Outra característica importante em sistemas de processamento de áudio é o modo como o sinal é enviado para o processador. Para as aplicações que utilizam matrizes de microfone, esse processamento é frequentemente realizado por processadores de PCs (*Personal Computer*). Geralmente, deseja-se que o sinal de áudio recebido esteja em formato PCM (*Pulse Code Modulation*), por ser mais de mais fácil manipulação [Kite (2012)]. No entanto, o protocolo de comunicação utilizado para enviar esse sinal pode variar.

Um PC possui diferentes interfaces de comunicação que possibilitam o envio desses sinais. Dispositivo com matrizes de microfones com comunicação ethernet já foram desenvolvidos [Reitbauer et al. (2012)], assim como matrizes de microfone com conectividade USB [Todorović et al. (2017)].

O presente projeto tem por objetivo desenvolver um dispositivo similar aos citados, o qual consiste em uma interface para uma matriz de seis microfones PDM com conectividade USB, para ser utilizado em projetos de pesquisa de localização de emissores de som. Esse dispositivo deve implementar os filtros necessários para recuperar o sinal de áudio em formato PCM dos diferentes canais da matriz de microfones sem que haja perda de sincronia. O dispositivo também implementar uma interface USB para se comportar como um dispositivo multi-canal de áudio, de forma que um *software* de edição de áudio com suporte a esse tipo de dispositivo possa receber o sinal.

Para a realização do projeto, foi utilizada um placa de aquisição de áudio com seis microfones PDM. A interface com os microfones foi feita por uma FPGA Virtex-5 contida em um kit Xilinx XUPV5. No decorrer do projeto, foi desenvolvido o código em VHDL que descreve a funcionalidade do *hardware* contido na FPGA.

A implementação do protocolo USB foi feita pelo controlador USB Cypress *Ez-Host*,

o qual está também presente no kit Xilinx XUPV5. Durante o projeto, foi desenvolvido o *firmware* deste controlador em linguagem C. Visto que o dispositivo desenvolvido no projeto tem as funções de realizar a interface com os microfones PDM e enviar o sinal de áudio via USB, o trabalho realizado foi dividido em dois módulos.

O primeiro módulo consiste no *hardware* responsável por receber o sinal PDM diretamente dos microfones e transformá-lo em PCM. Uma imposição do projeto é que essa transcodificação não induza a perda da sincronia entre os sinais dos diferentes canais de entrada. O *hardware* desenvolvido neste primeiro módulo do projeto é totalmente contido na FPGA. Além desse código, foi realizada uma simulação em MATLAB dos filtros necessários para obter o sinal PCM. Essa simulação foi utilizada para verificar se um sinal de áudio captado pelo microfone seria corretamente reconstruído pelos filtros projetados. A geração deste sinal PDM captado pelo microfone foi também simulada.

O segundo módulo do trabalho corresponde ao sistema responsável pela comunicação, o qual tem o objetivo de implementar o protocolo USB para envio dos dados PCM gerados. Esse módulo utiliza um controlador USB para implementação do protocolo, mas consiste também no *hardware* necessário para o envio do sinal PCM da FPGA para o controlador USB. Esse envio é feito através do protocolo HPI (*Host Port Interface*), o qual é específico ao controlador USB Cypress *Ez-Host* e fornece acesso direto à sua memória RAM. Para isso, foi desenvolvida em VHDL uma interface que escreve continuamente os sinais PCM dos seis canais na memória do controlador USB através do protocolo HPI.

A transcodificação PDM-PCM foi verificada pela análise do sinal gerado através de uma analisador lógico TLA 6402. Com a FPGA conectada ao analisador lógico, o código VHDL foi configurado para enviar os bits do sinal PCM para as saídas conectadas ao analisador. Desse modo, o sinal foi adquirido e desenhado na tela do analisador, mostrando a forma de onda do sinal PCM gerado.

Outra forma utilizada para validar a transcodificação foi utilizando o codec AC97 SoundMax AD1981 presente na placa Xilinx XUPV5. Este codec apresenta interfaces de saída e entrada de áudio analógicas, e foi utilizado para gerar um sinal analógico a partir do sinal PCM. Esse sinal é direcionado para um conector de áudio, ao qual foi conectado um alto-falante. Dessa maneira, foi possível escutar o áudio PCM produzido e verificar se ele correspondia ao áudio captado na entrada do microfone.

Já o módulo de comunicação foi testado através da ferramenta de *software* USBLyzer executada em um PC com sistema operacional Windows. Essa ferramenta permitiu observar a comunicação no barramento USB no momento em que o dispositivo foi conectado ao PC. Foi também utilizado o *software* de edição de áudio Audacity para verificar se o sinal de áudio dos seis canais foi recebido pelo PC.

2 REVISÃO BIBLIOGRÁFICA

2.1 Sinais digitais

Sinais digitais são sequências de números, cada um com um índice inteiro que representa um instante de tempo de forma descontínua. Eles podem ser puramente sintéticos ou podem ser uma representação de sinais analógicos [Tenoudji (2012)].

Uma classe de sinal analógico que pode ser representada por sinais digitais são os sinais de áudio. Existem diversos formatos de digitalização de sinais de áudio, sendo o LPCM *Linear Pulse-Code Modulation*) o formato mais utilizado [Hoffmann (2002)]. No entanto, existem outros sistemas de representação como DPCM (*Differential PCM*), ADPCM (*Backward-adaptative differential PCM*), MLP (*Meridian Lossless Packing*), DSD (*Direct Stream Digital*) [Hoffmann (2002)], PDM (*Pulse Density Modulation*) [Kite (2012)], entre outros. Nessa seção serão abordados em mais detalhe os formatos PDM e LPCM.

2.1.1 Decimação e Aliasing

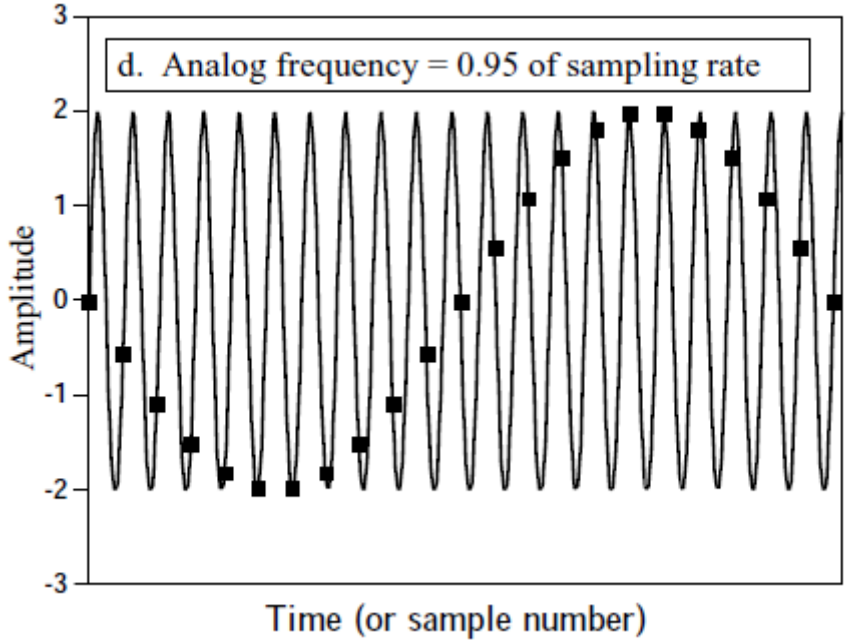
A representação em intervalos de tempo descontínuos (discretos) intrínseca aos sinais digitais apresenta limitações. Se o sinal é amostrado com uma frequência de amostragem f_s, então ele deve ter sua banda de frequência limitada no intervalo $[-f_s/2, f_s/2]$. Este limite máximo correspondente à metade da frequência de amostragem é chamado de frequência de Nyquist [Smith (1999)].

Caso essa regra não seja obedecida, ocorre o fenômeno de *aliasing*, que consiste no rebatimento de frequências fora do intervalo $[-f_s/2, f_s/2]$ para dentro desse intervalo. A Figura 1 apresenta esse processo, onde um sinal com frequência de $0,95f_s$ é percebido com um sinal de frequência de $0,05f_s$ após a amostragem.

O fenômeno de *aliasing* pode também ocorrer no processo de decimação. A decimação é a redução da frequência de amostragem de um sinal discreto, e é realizada descartando-se uma a cada. O fator de decimação é definido como a razão entre a frequência de amostragem do sinal original e a frequência de amostragem do sinal decimado.

Para isso, considera-se um sinal discreto de frequência de amostragem f_{s1} que passa pelo processo de decimação com fator de decimação D para gerar um sinal com frequência de amostragem $f_{s2} = f_{s1}/D$. Sabendo que o sinal original tem componentes frequenciais apenas até $f_{s1}/2$, o *aliasing* ocorre para todas as componentes frequenciais no sinal original entre $f_{s2}/2$ e $f_{s1}/2$.

Figura 1: Amostragem de um sinal com frequência acima da frequência de Nyquist.



Fonte: Smith (1999)

2.1.2 PCM

PCM é um sistema de representação de um sinal amostrado com uma série de amostras de múltiplos bits [Kite (2012)]. A conversão de um sinal analógico em PCM consiste em duas etapas: amostragem e quantização. A amostragem é o processo em que a variável independente de um sinal, isso é, o tempo, é transformado de contínuo para discreto [Smith (1999)]. A quantização consiste em atribuir um valor digital a uma tensão analógica [Hoffmann (2002)]. A forma de quantização mais utilizada é a linear, em que existe um intervalo fixo entre um nível de quantização e o seguinte [Hoffmann (2002)]. Sinais PCM gerados por esse método de quantização são chamados LPCM (*Linear pulse-code modulation*).

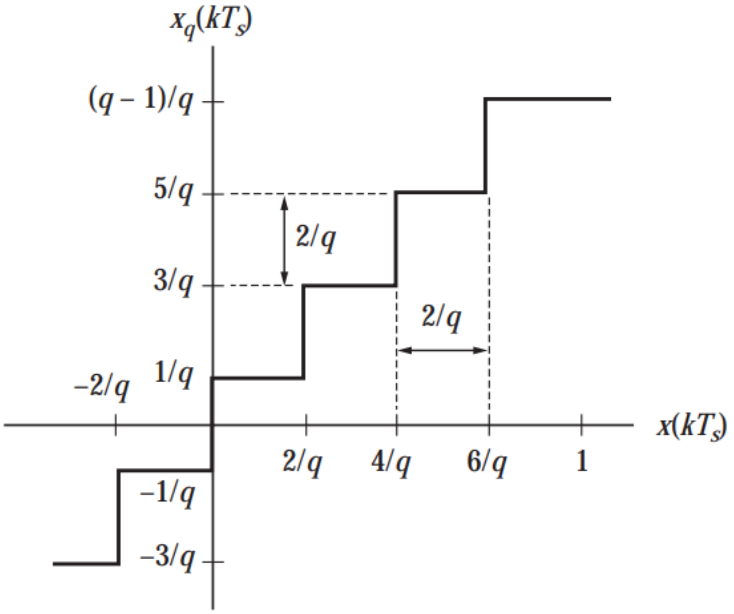
Na Figura 2, esse processo de quantização linear é demonstrado graficamente. A escala total do sinal analógico foi definida como o intervalo $[-1, 1]$ e dividida em q intervalos de quantização. Desse modo, o intervalo de quantização tem largura de $2/q$. Na Figura 2, o sinal amostrado corresponde a $x(kT_s)$ e o sinal quantizado a $x_q(kT_s)$ [Carlson (2002)].

Supondo um sinal analógico $x(t)$, a Figura 3 mostra o resultado de sua amostragem $x(kT_s)$ de sua quantização $x_q(t)$.

Uma vez que cada valor quantizado é representado por um número binário, o número de intervalos de quantização é uma potência de 2. Considerando um sinal PCM com n bits, o número de intervalos de quantização q é igual a 2^n.

O processo de quantização incute um erro no sinal PCM, chamado erro de quantização. Esse erro corresponde à diferença entre os sinais $x_q(t)$ e $x(t)$, representado pela equação 1.

Figura 2: Função de transferência de bloco quantizador.



Fonte: Carlson (2002)

$$\epsilon_k = x_q(kT_s) - x(kT_s) \tag{1}$$

Na maior parte dos casos, não há correlação entre ϵ_k e $x(t)$ [Carlson (2002)]. O arredondamento realizado durante a quantização garante que $|\epsilon_k| \leq 1/q$. Assume-se que esse erro tem valor médio zero e densidade de probabilidade uniforme no intervalo $[1/q, 1/q]$ [Carlson (2002)]. Portanto, a potência do sinal de erro de quantização é dado pela equação 2.

$$\sigma_k{}^2 = \epsilon_k{}^2 = \frac{1}{2q}\int_{-1/q}^{1/q} \epsilon^2 d\epsilon = \frac{1}{3q^2} \tag{2}$$

Isso mostra que o erro diminui quadraticamente quando o número de intervalos de quantização aumenta. Para um sinal com potência S_x, é possível calcular a relação sinal-ruído de quantização a partir da equação 3.

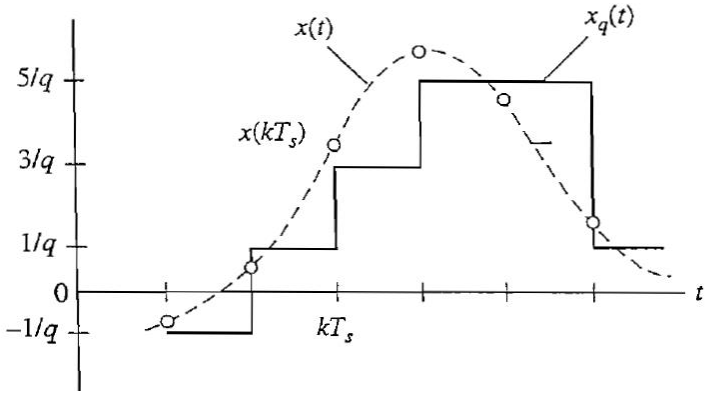
$$\left(\frac{S}{N}\right)_q = \frac{S_x}{\sigma_q{}^2} = 3q^2 S_x \tag{3}$$

Sabendo que o número de intervalos de quantização de um sinal PCM é dado por $q = 2^n$, essa relação em decibeis pode ser dada pela equação 4. Para o seu cálculo, considera-se que a amplitude do sinal tem valor unitário, ou seja, $S_x = 1$. Isso significa que o sinal considerado ocupa toda a escala do sinal analógico de entrada.

$$SNR_q = 10\log_{10}(3 \cdot 2^{2n} S_x) = 6.0n + 4.8\text{dB} \tag{4}$$

Em casos onde o sinal tem amplitude mais baixa, essa relação pode diminuir, sendo necessária uma quantidade maior de bits para descrever um sinal com um ruído de quantização aceitável.

Figura 3: Representação da modulação PCM de um sinal.



Fonte: Carlson (2002)

2.1.3 PDM

A modulação por densidade de pulsos ou PDM (*Pulse Density Modulation*) é uma forma de representação de sinais com apenas 1 bit de informação. Para ter uma representação adequada, utiliza técnicas como sobreamostragem e *noise shaping* [Kite (2012)].

O *Noise shaping* em sinais PDM é a propriedade de reduzir o ruído de quantização nas baixas frequências e aumentar em altas frequências [Loibl et al. (2016)]. Para este método surtir efeito, é necessário aumentar a largura de banda do sinal, para que o ruído possa ser mais intenso em frequências mais altas que as frequências audíveis [Kite (2012)]. Isso é feito através da sobreamostragem, que consiste na utilização de uma frequência de amostragem muito maior que o dobro da largura de banda do sinal de áudio. Em microfones PDM, por exemplo, a frequência de amostragem é tipicamente próxima a 3 MHz [Kite (2012)].

No caso de sinais modulados PDM, o *noise shaping* é aplicado de forma a transferir o ruído para altas frequências, acima da largura de banda do sinal. Dessa forma, ele pode ser removido por filtros passa-baixa, aumentando assim a SNR do sinal.

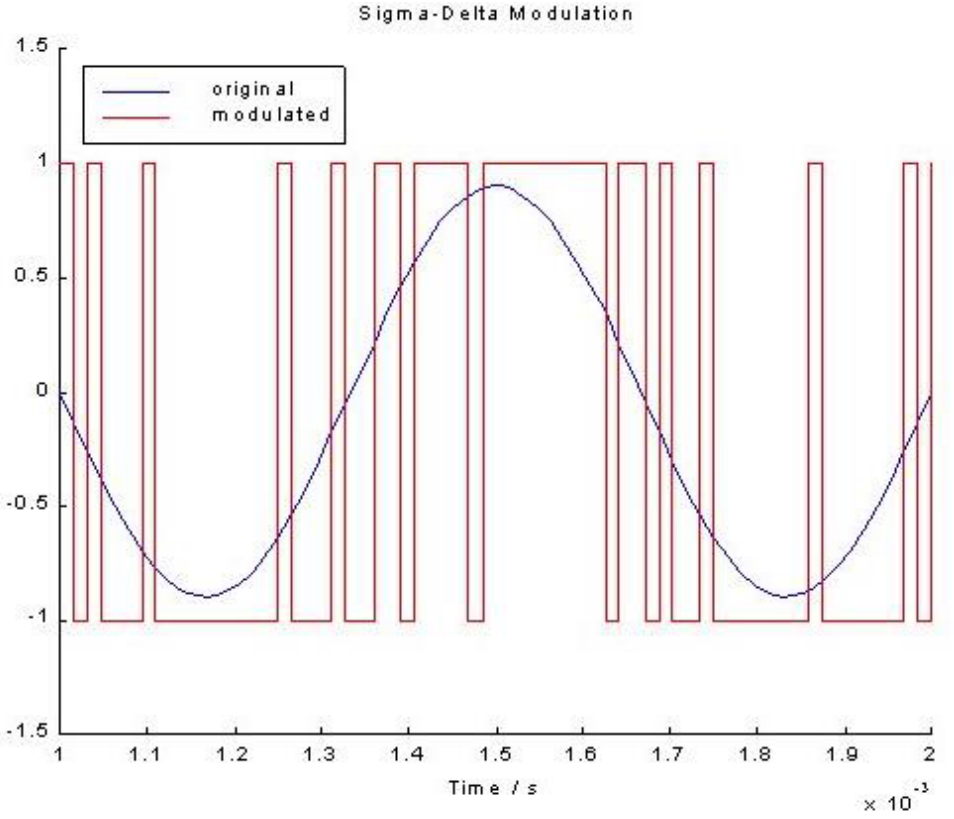
O princípio do sinal PDM está em representar um sinal contínuo por uma densidade de valores ‘1’ ou de valores ‘0’. Tendo uma frequência de amostragem fixa, o sinal PDM possui mais valores ‘1’ do que ‘0’ quando o sinal modulado é positivo, e mais valores ‘0’ do que ‘1’ quando o sinal modulado é negativo. Quanto mais positivo o sinal modulado, maior a densidade relativa de valores ‘1’, e vice-versa.

O sinal PDM deve sempre estar acompanhado do seu respectivo *clock*, que determina os momentos em que ele é amostrado. Um exemplo de sinal PDM com seu respectivo sinal modulado pode ser visto na Figura 4.

Na Figura 4, observa-se a modulação PDM de uma senoide. Nas cristas da senoide, há uma grande densidade de valores ‘1’, ao passo que nos vales da senoide, há uma grande densidade de valores ‘0’.

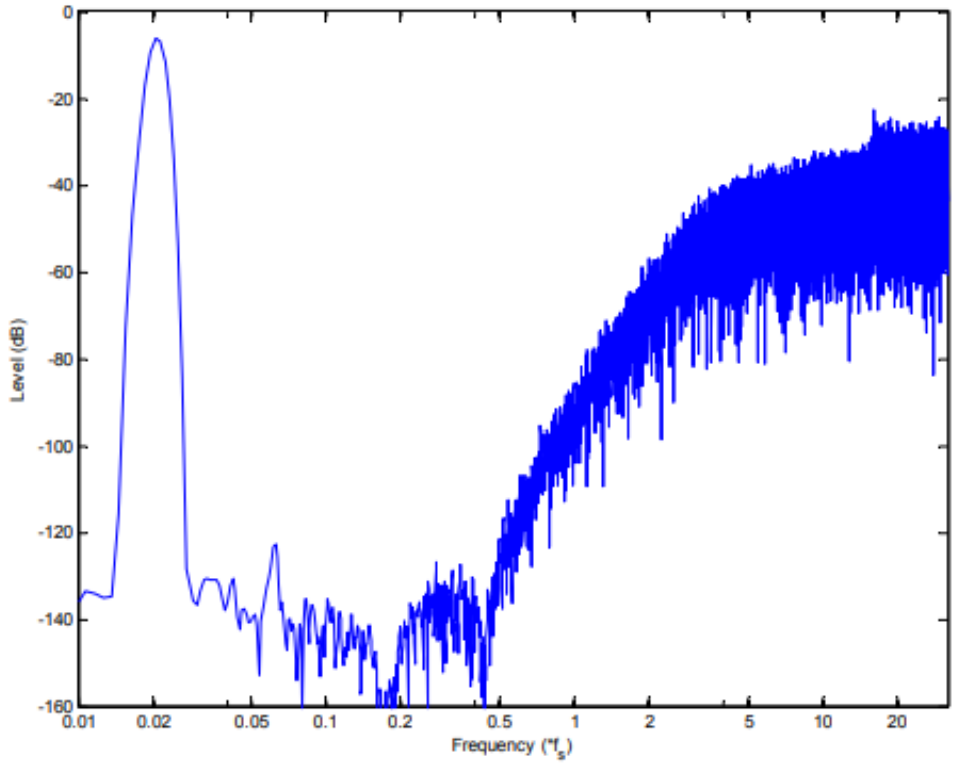
Na Figura 5, outro exemplo mostra o comportamento no domínio frequência de um sinal PDM.

Figura 4: Modulação PDM de uma senoide.



Fonte: Rosti (2004)

Figura 5: Sinal modulado PDM no domínio frequência.



Fonte: Kite (2012)

Nesse exemplo, o sinal modulado é uma senoide de frequência 50 vezes menor que a frequência de amostragem de saída. A banda de passagem do sinal vai até $0.5f_s$. A banda acima disso é gerada pela sobreamostragem.

Observa-se que o sinal modulado aparece como um pico na banda base, com amplitude muito superior ao ruído na mesma banda. O *noise shaping* do sinal PDM faz a maior parte de ruído se localizar acima de $0.5f_s$. Dessa forma, o sinal original pode ser obtido com baixa distorção a partir da filtragem do sinal PDM.

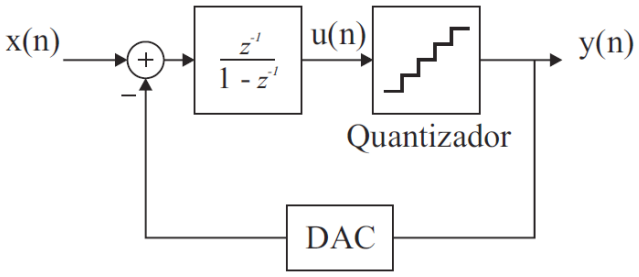
2.1.4 Moduladores Sigma-Delta

Moduladores sigma-delta são utilizados no processo de conversão analógico digital para gerar sinais modulados PDM. Para isso, esse modulador se baseia em um bloco integrador, um quantizador e um laço de realimentação [Aguirre (2014)]. Embora possam existir moduladores sigma-delta com quantizadores de diferentes quantidades de bits, para a geração de sinais PDM semelhantes ao apresentado na seção anterior é necessário que o quantizador tenha apenas 1 bit.

A topologia de um modulador sigma-delta de primeira ordem pode ser visto na Figura 6. Observa-se que a topologia é composta por um laço principal e um laço de realimentação. O laço principal possui um integrador e um quantizador. O sinal de saída do quantizador, o qual está no domínio digital, passa por um conversor digital-analógico. Um bloco subtrator faz a diferença entre o sinal de entrada e a saída do conversor D/A, definindo o laço de realimentação negativa.

Esta topologia de modulador contém somente um integrador. No entanto, outras topologias podem conter mais. A ordem do modulador é definida pelo número de integradores que ele contém, o que também define a sua complexidade e resposta em frequência.

Figura 6: Diagrama de blocos de um modulador Sigma-Delta de primeira ordem.



Fonte: Aguirre (2014)

O laço de realimentação calcula a diferença entre o sinal de entrada e o sinal PDM de saída. Essa diferença é então integrada, enquanto que o quantizador gera o valor de saída comparando o sinal de diferença integrado com ’0’.

Se o sinal de saída for maior que o sinal de entrada, o erro será negativo, e o sinal u(n) terá uma derivada negativa até o momento em que cruzar o eixo horizontal. Nesse momento, o sinal de saída passará a ser maior que o sinal de entrada, e o sinal u(n) terá uma derivada positiva. Dessa forma, o sinal de saída estará sempre oscilando entre ‘0’ e ‘1’. No entanto, para sinais positivos, a saída passará mais tempo no valor ‘1’, e para sinais negativos a saída passará mais tempo no valor ‘0’. Tal comportamento corresponde ao esperado de um sinal modulado PDM.

2.1.5 Projeto de filtros FIR

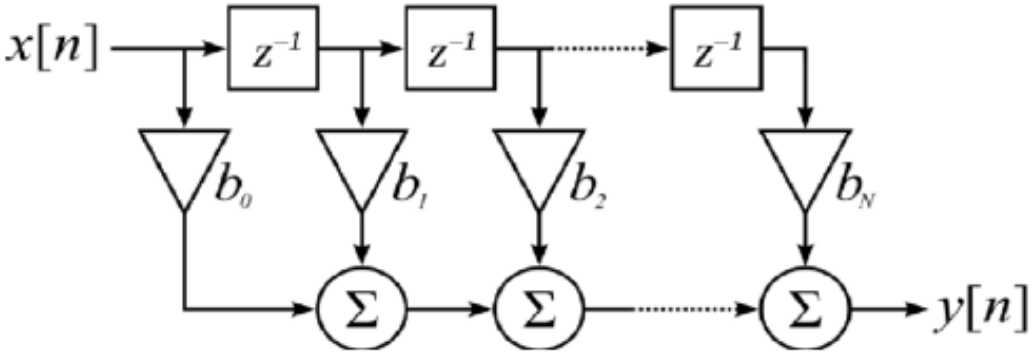
Para o processamento de sinais digitais, um tipo de filtro particularmente útil é o filtros FIR (*Finite Impulse Response*). Filtros FIR recebem esse nome porque a sua resposta a uma entrada do tipo impulso unitário tem um número finito de coeficientes não-nulos.

Um filtro FIR pode ser descrito inteiramente pela sua resposta ao impulso. A saída do filtro pode ser obtida pela convolução da entrada pela resposta ao impulso [Smith (1999)]. Portanto, o cálculo do sinal de saída pode ser dado pela equação 5.

$$y[n] = \sum_{i=0}^{m} a_i x[n-i] = a_0 x[n] + a_1 x[n-1] + a_2 x[n-2] + ... + a_m x[n-m] \quad (5)$$

A implementação de um filtro FIR genérico pode ser feita pela utilização de blocos multiplicadores com os coeficientes a_i, multiplicando o sinal de entrada x_n atrasado de um número i de amostras. Os blocos de atraso utilizados nessa arquitetura são comumente chamados de *taps*. A Figura 7 mostra o diagrama de blocos de uma implementação genérica do filtro FIR, utilizando blocos de atraso, multiplicação e soma.

Figura 7: Diagrama de blocos de um filtro FIR.



Fonte: Kumar (2016)

Um dos métodos utilizados para calcular os coeficientes de um filtro FIR partindo de uma resposta em frequência desejada é o método de janelamento. O primeiro passo deste método é escolher a resposta em frequência ideal desejada. Para um filtro passa-baixa, esta resposta é um filtro com ganho unitário na banda de passagem, nas frequências abaixo de f_c, e atenuação total na banda de rejeição, nas frequências acima de f_c [Smith (1999)].

O segundo passo é calcular a transformada de Fourier inversa da função gerada. No caso do filtro passa-baixas ideal, este resultado é uma função do tipo $sinc(x)$ [Smith (1999)]. Idealmente, os coeficientes seriam iguais aos valores desta função calculados em intervalos igualmente espaçados. No entanto, a sua implementação não é possível, por duas razões. Primeiramente, apresentaria coeficientes não nulos de índice negativo. Além disso, esta função se estenderia ao infinito, resultando em infinitos coeficientes necessários para sua implementação [Smith (1999)].

A solução é passar esta função por dois processos. O primeiro é realizar o truncamento desta função com $M + 1$ coeficientes de forma simétrica em torno do lóbulo principal, sendo M um número par. O segundo processo consiste em deslocar todos os coeficientes para a direita de $M/2$ índices. O resultado é uma resposta ao impulso com coeficientes de índices variando de 0 a M [Smith (1999)].

Este filtro, embora implementável, apresenta resposta em frequência diferente da resposta em frequência ideal desejada, com *ripple* excessivo na banda de passagem [Smith (1999)]. Isso é causado pelo truncamento abrupto realizado nos coeficientes. A solução para isso é multiplicar todos os coeficientes por uma janela mais suave, o que melhora a resposta em frequência. Existem diferentes tipos de janelas que podem ser aplicadas nesta situação. Exemplos são a janela de Blackman, Hamming [Smith (1999)] ou Kaiser.

2.1.6 Filtro CIC

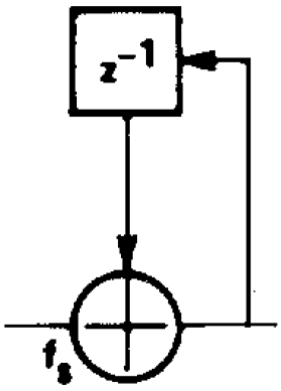
O filtro CIC (*Cascaded Integrator-Comb*) é um filtro digital do tipo FIR capaz de realizar a interpolação ou decimação de um sinal digital [B. Hogenauer (1981)].

Uma característica importante dessa classe de filtros é a sua eficiência em *hardware*, não necessitando de multiplicadores e somente de somadores e registradores, sendo estes de tamanho limitados se comparados a filtros FIR tradicionais. A sua arquitetura é composta de uma seção de integração e outra de filtros *comb*, intercaladas por um bloco de decimação ou de interpolação.

O número de integradores é sempre igual ao número de filtros *comb*, e depende da ordem N do filtro implementado, que por sua vez depende da resposta em frequência desejada para o filtro.

A implementação de um bloco integrador é feita somente com um somador e um bloco de atraso, e tem como função incrementar à saída do bloco o valor da entrada. O diagrama de blocos da estrutura de um integrador pode ser vista na Figura 8.

Figura 8: Diagrama de blocos de um integrador.



Fonte: B. Hogenauer (1981)

A equação de recorrência do integrador é dada pela equação 6.

$$y[n] = y[n-1] + x[n] \tag{6}$$

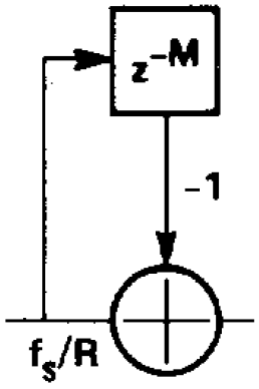
Logo, a sua função de transferência é dada pela equação 7.

$$H_I(z) = \frac{1}{1 - z^{-1}} \tag{7}$$

O bloco *comb* tem a função de subtrair o sinal de entrada no tempo presente do seu valor anterior. O diagrama de blocos de sua estrutura é apresentado na Figura 9.

O atraso M apresentado é um parâmetro do filtro que é geralmente estabelecido como 1 ou 2. A sua equação de recorrência é dada pela equação 8.

Figura 9: Diagrama de blocos de um filtro *comb*.



Fonte: B. Hogenauer (1981)

$$y[n] = x[n] - x[n - M] \tag{8}$$

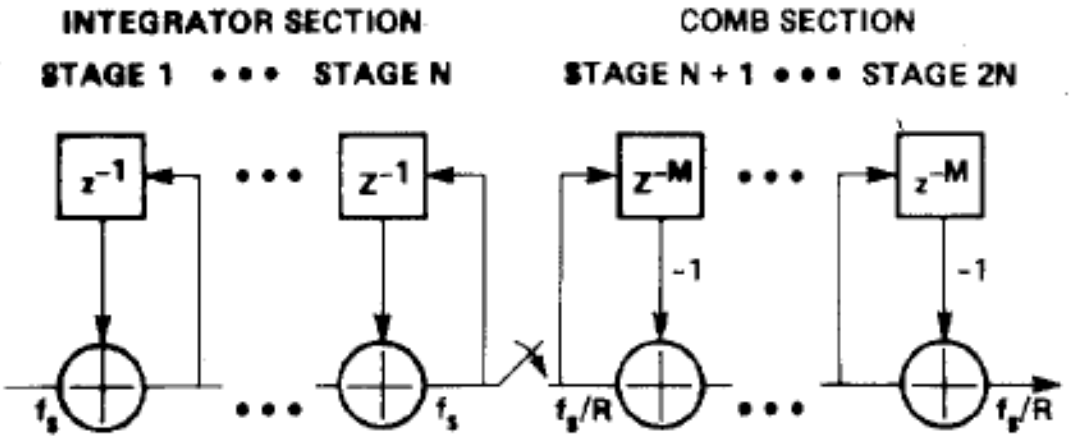
Logo, a sua função de transferência é dada pela equação 9.

$$H_C(z) = 1 - z^{RM} \tag{9}$$

Além desses blocos, o filtro CIC também é composto por um bloco decimador ou interpolador, dependendo do tipo do filtro. A sua função é reduzir ou aumentar a frequência de amostragem, respectivamente. O bloco decimador, que será utilizado nesse projeto, realiza essa redução da frequência de amostragem selecionando uma a cada R amostras.

Portanto, o diagrama de blocos completo do filtro CIC decimador é dado apresentado na Figura 10.

Figura 10: Diagrama de blocos de um filtro CIC.



Fonte: B. Hogenauer (1981)

Observa-se que esse filtro apresenta apenas três parâmetros construtivos: o fator de decimação R, a ordem N, e o atraso da seção *comb* M.

A partir das funções de transferências já estabelecidas para os diferentes blocos do filtro, a função de transferência do filtro completo, considerando a frequência de amostragem de entrada do filtro CIC decimador, é dada pela equação 10.

$$H_{CIC}(z) = H_I(z)^N H_C(z)^N = \frac{(1 - z^{RM})^N}{(1 - z^{-1})^N} = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^N \tag{10}$$

É importante salientar que, como os blocos integradores tem coeficiente de realimentação unitário e estão em malha aberta, há *overflow* nos integradores no filtro CIC decimador. No entanto, isso não é um problema para o funcionamento do filtro se duas condições forem respeitadas:

- os integradores devem ser implementados utilizando representação em complemento de 2, de modo que haja *wrap-around* [Smith (2007)] entre o maior e o menor número da representação;
- a largura de bits do sinal utilizado deve ser capaz de comportar o valor máximo do sinal da saída [B. Hogenauer (1981)].

2.1.7 Características frequenciais do filtro CIC

Filtros CIC têm característica de filtro passa-baixa. A sua função de transferência no domínio frequência pode ser obtida a partir da substituição dada na equação 11.

$$z = e^{\frac{j2\pi f}{R}} \tag{11}$$

Com isso, obtém-se a expressão dada na equação 12 para a resposta em frequência da potência do filtro CIC, sendo f a razão entre a frequência analisada e a frequência de amostragem de saída do filtro CIC decimador.

$$P(f) = \left[\frac{sin(\pi M f)}{sin(\frac{\pi f}{R})}\right]^{2N} \tag{12}$$

Observa-se que essa função de transferência apresenta zeros quando f é igual a um múltiplo de $1/M$, por sua característica senoidal. Como a frequência de amostragem de entrada é maior que a de saída, existem múltiplos zeros que dependem do valor do fator de decimação R.

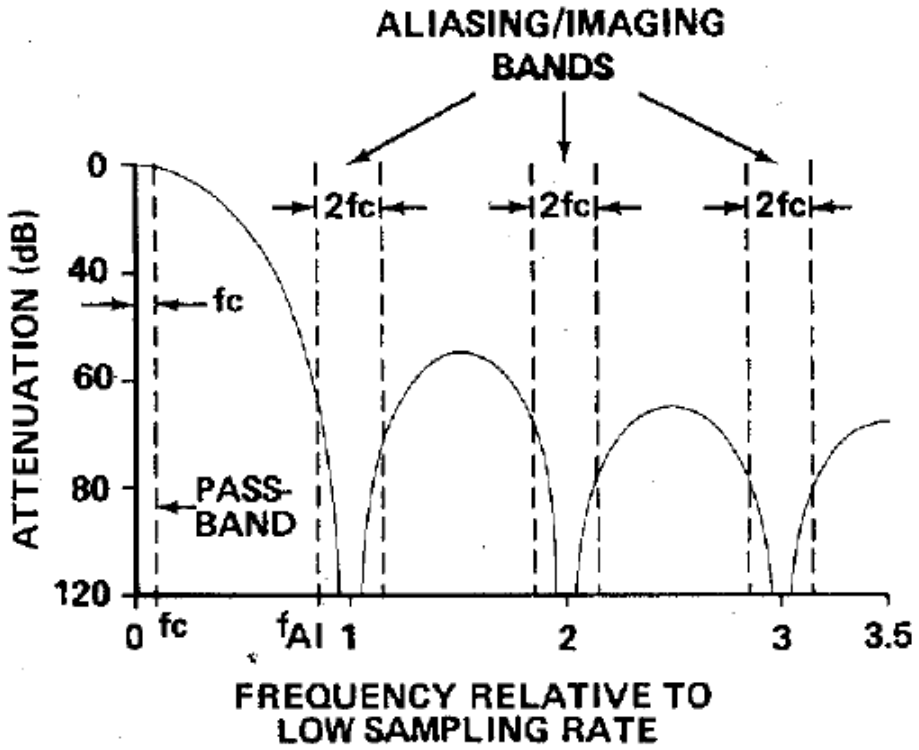
Além disso, por causa da diferença entre a frequência de Nyquist da entrada e da saída do filtro, o sinal existente em algumas bandas de frequência é rebatido à banda de interesse. Essas regiões rebatidas se encontram no entorno dos zeros, e o valor da atenuação nesse ponto depende dos parâmetros do filtro e da largura da faixa de interesse. Portanto, no momento do projeto de um filtro CIC, devem ser levadas em conta essa característica de *aliasing* do filtro CIC.

Um gráfico apresentando a resposta do filtro pode ser visto na Figura 11.

2.1.8 Design de filtros CIC

O projeto de filtros CIC consiste na determinação de seus três parâmetros (R, M e N) e do número de bits utilizado em cada estágio. Enquanto que a determinação dos parâmetros influenciam principalmente nas características frequenciais do filtro, a largura em bits determina a amplitude máxima do sinal de saída e garante a estabilidade do filtro.

Figura 11: Resposta em frequência de um filtro CIC.



Fonte: B. Hogenauer (1981)

Para obter a quantidade de bits das etapas do filtro, é necessário garantir que não haja perda de informação devido ao overflow dos registradores. A partir das funções de transferência combinada de cada etapa (integradores e filtros *comb*), é possível calcular a máxima magnitude de saída relativa ao pior caso possível de sinal de entrada para a maior amplitude de entrada [B. Hogenauer (1981)].

Utilizando essa definição, o artigo de Hogenauer apresenta uma equação para calcular a quantidade necessária de bits, dada na equação 13.

$$B_{max} = [N \log_2(RM) + B_{in} - 1] \tag{13}$$

Se essa quantidade de bits é maior que a largura desejada para o sinal de saída, é possível realizar arredondamento em algumas etapas do filtro, desde que respeitada a quantidade calculada na saída dos integradores.

Uma vez expostas as bases teóricas sobre sinais digitais que serão utilizadas durante este projeto, a próxima seção detalhará o protocolo de comunicação escolhido para o envio de dados no dispositivo projetado, o USB.

2.2 USB

Universal Serial Bus, ou USB, é um protocolo de comunicação que foi desenvolvido tendo em vista objetivos como facilidade de uso, suporte para transmissão em tempo real de áudio e vídeo e criação de soluções de baixo custo com taxa de transferência de até 480 Mbps para o USB 2.0 [Compaq et al. (2000)]. Sendo do tipo *plug-and-play*, é compatível com diversos dispositivos, tais como *flash drives*, microfones, câmeras, impressoras,

teclados, *mouses* e diversos outros.

USB 2.0 suporta as velocidades *high speed* (480Mbps), *full speed* (12Mbps) e *low speed* (1.5Mbps). A versão mais recente do protocolo, USB 3.0, suporta ainda a velocidade *SuperSpeed* de 4,8 Gbps. No entanto, a largura de banda deve ser compartilhada por transferências de status, controle, verificação de erros e dados. Logo, a taxa máxima para transferência de dados de uma aplicação é mais baixa, sendo de 53MB/s para *high speed*, 1.2MB/s para *full speed* e 800B/s para *low speed* [Axelson (2009)].

Os agentes de um barramento de comunicação USB são o *host* e o dispositivo. Um sistema USB contém um único *host*, que controla o tráfego de dados no barramento, sendo responsável por iniciar transferências às quais os dispositivo respondem. Um dispositivo pode ter uma ou mais funções, que são enquadradas conforme a sua classe. Uma função USB se refere a uma capacidade de um determinado dispositivo. O dispositivo também possui um endereço definido pelo *host*, que é utilizado em todas as comunicações.

Um *host* USB geralmente tem um controlador responsável por controlar o tráfego no barramento. Além disso, possui diversos *drivers*, que são responsáveis por fazer a interface entre o sistema operacional do *host* e uma função específica do dispositivo. O *host* deve ser capaz de detectar diferentes dispositivos (tanto durante a inicialização quanto quando um novo dispositivo é conectado), gerenciar o fluxo de dados (dividir o tempo disponível entre os diferentes dispositivos no mesmo barramento), checar erros na comunicação e gerenciar a corrente de alimentação.

Já o dispositivo deve primordialmente ser capaz de responder a todas as requisições do *host*, que são identificadas pelo endereço do dispositivo no barramento. Ele também deve, assim como o *host*, realizar a checagem de erros da comunicação.

Toda comunicação USB se enquadra em um dos quatro tipos de transferências: *control*, *bulk* , *interrupt* e *isochronous*, sendo cada uma adaptada para diferentes casos de comunicação.

Transferências do tipo *control* são obrigatórias em qualquer comunicação USB, uma vez que é através delas que o *host* configura o device no momento em que ele é conectado ao barramento USB. Além disso, são as únicas que tem funções definidas nas especificações USB, as quais são chamadas de *requests*. Através dessas transferências, o *host* pode ler informações sobre um dispositivo e configurá-lo.

Transferências do tipo *bulk* são feitas para aplicações em que há necessidade de maior volume de dados, mas a taxa de transmissão não é crítica. Em um barramento ocupado, transferências *bulk* devem esperar e podem ser lentas. No entanto, em um barramento ocioso, são as transferências mais rápidas. Exemplos de dispositivos que as utilizam são *flash drives*, impressoras e *scanners*.

Transferências do tipo *interrupt* são utilizadas por dispositivos para os quais não são desejáveis atrasos na comunicação. Tais dispositivos necessitam de baixa latência, mas geralmente com baixa quantidade de dados. São utilizadas por dispositivo HID (*Human Interface Devices*) tais como *mouses* e teclados.

Transferências do tipo *isochronous* têm taxa de envio garantida pelo barramento, e são utilizadas por dispositivos que visam transmissão em tempo real. No entanto, não existe checagem de erros para esse tipo de transferência. Dessa forma, não há retransmissão de dados em caso de falha de transmissão, e erros ocasionais são mais propensos a acontecer. Essas transferências são utilizados em dispositivos que requerem uma quantidade de largura de banda fixa e suporta erros de transmissão, como é o caso de dispositivos de áudio e vídeo.

Todas as transferências USB são compostas por uma ou mais transações, que por sua

vez podem ter três tipos: IN, OUT ou *Setup*. Transações do tipo OUT indicam o envio de dados do *host* para o dispositivo, e transações do tipo IN indicam o sentido contrário. Transações do tipo *Setup* indicam que uma transferência do tipo *control* está começando. A Tabela 1 mostra informações sobre os diferentes tipos de transação.

Tabela 1: Tipos de transações USB.

Transaction Type	Source of Data	Types of Transfers that Use the Transaction Type	Contents
IN	device	all	data or status information
OUT	host	all	data or status information
Setup	host	control	a request

Fonte: Axelson (2009)

Transações USB, por sua vez, são subdivididas em pacotes. Uma transação é formada por um pacote *token*, um pacote de dados e um pacote de *handshake*. Além de conter um PID (Packet ID), pacotes podem conter dados, endereços e checagem de erros (CRC).

A comunicação USB é dividida em *frames*, que determinam um espaço de tempo constante de 1 ms para ditar o taxa das transações. Para sinalizar um novo *frame*, o *host* USB envia periodicamente um pacote do tipo SOF (Start of Frame).

Os dispositivos USB são organizados conforme uma estrutura hierárquica com três níveis de organização: configuração, interface e *endpoint*. Uma configuração pode conter múltiplas interfaces, que por sua vez pode conter múltiplos *endpoints*.

Cada configuração determina um conjunto de funcionalidades que um dispositivo pode desempenhar. Um dispositivo pode ter somente uma configuração selecionada por vez. Essa seleção é realizada pelo *host*. Já uma interface estabelece uma única função do dispositivo. O propósito dos diferentes *endpoints* contidos em uma interface são determinados pela classe do dispositivo.

Interfaces também suportam o uso de *alternate settings*, que são configurações alternativas que uma interface pode assumir. Os *endpoints* associados a uma interface devem ser definidos para cada *alternate setting*, de modo que diferentes *alternate settings* podem possuir diferentes números de *endpoints*. Para uma interface com diferentes *alternate settings*, o *host* é responsável por selecionar qual conjunto de propriedades é utilizado em um determinado momento.

O *endpoint* é um endereço interno presente no dispositivo que armazena as informações para serem transmitidas ou que foram recebidas. Cada *endpoint* é definido por um conjunto de propriedades, tais como um endereço que o indentifica, uma direção de transmissão, uma quantidade máxima de bytes por transação, além do tipo de transferência que ele suporta. A direção pode ser IN (do dispositivo para o *host*) ou OUT (do *host* para o dispositivo).

O *endpoint* de endereço zero é reservado para mensagens do tipo *control*. Esse *endpoint* é utilizado para todas as comunicações USB realizadas nos primeiros momentos em que o dispositivo é detectado pelo *host*, em que ele fornece a descrição sobre todas as configurações, interfaces e *endpoints* que ele contém. Os outros *endpoints* podem ser definidos para qualquer outro tipo de transferência.

A enumeração é o processo que ocorre quando um dispositivo é conectado ao *host*. Durante esse processo, o *host* envia uma série de transferências do tipo *control* solicitando informações sobre o dispositivo, que ele utiliza para definir quais *drivers* são utilizado para a comunicação com aquele dispositivo. Parte da enumeração consiste no envio dos descritores do dispositivo.

Os descritores de um dispositivo USB são estruturas de dados que o descrevem para um *host*. Os descritores são organizados de forma hierárquica, e podem ser descritores de dispositivo, configuração, interface ou *endpoint*.

Um *host* faz o pedido de um descritor do dispositivo através de uma *request* do tipo *GetDescriptor*. No processo de enumeração, o descritor do tipo dispositivo é o primeiro descritor que um *host* recebe. Ele contém informações básicas sobre o dispositivo, tais como a sua classe, a sua identificação e o seu número de configurações. Em seguida, o *host* requisita os outros descritores.

Dispositivos USB podem ser divididos em classes. Exemplos de classes são áudio, vídeo ou impressoras. A classe do dispositivo determina a funcionalidade que ele desempenha, e é especificada nos descritores que o dispositivo envia para o *host*. Ela pode ser definida tanto nos descritores de dispositivo quanto nos descritores de interface, dependendo da classe. Neste projeto, foi interessante um estudo mais aprofundado da classe áudio, a qual foi utilizada no dispositivo desenvolvido.

A classe é áudio é sempre especificada nos descritores de interface. Além disso, cada interface de classe áudio deve pertencer a uma de três subclasses: *AudioControl*, *AudioStreaming* e *MidiStreaming*. A primeira é utilizada para conter o *endpoint* do tipo *control* do dispositivo, e é por onde o *host* pode modificar as suas propriedades de áudio. Interfaces de subclasse *AudioStreaming* carregam os streams de áudio do dispositivo. Já interfaces de subclasse *MidiStreaming* transportam dados MIDI.

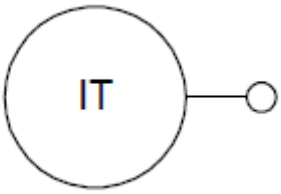
Uma função áudio em um dispositivo é composta por uma interface *AudioControl* e o número de interfaces *AudioStreaming* e *MidiStreaming* que forem necessárias. Cada função expõe para o *host* propriedades físicas do dispositivo que podem ser manipuladas. Cada função deve ser descrita por uma interconexão de entidades, que se dividem em terminais e *units*. A especificação USB descreve sete tipos de terminal e *units* que podem ser utilizados para descrever uma função áudio USB:

- *Input terminal* (terminal de entrada)
- *Output terminal* (terminal de saída)
- *Mixer Unit*
- *Selector Unit*
- *Feature Unit*
- *Processing Unit*
- *Extension Unit*

Cada entrada e saída de uma entidade corresponde a um pino. Este pino não é um componente físico, mas sim um símbolo utilizado para retratar as entradas e saídas. Uma entidade pode ter múltiplos pinos de entrada, mas apenas um pino de saída. Um pino representa um *cluster* de áudio, por onde podem passar múltiplos canais, desde que sejam síncronos. Um exemplo de *cluster* de áudio é um sinal estéreo, que contém dois canais.

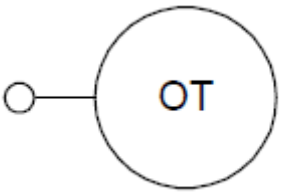
Terminais indicam os pontos de entrada e saída de uma stream de áudio no dispositivo. Na topologia de uma função áudio USB, um terminal de entrada se comporta como uma fonte, ao passo que um terminal de saída se comporta como um sumidouro. Portanto, um terminal de entrada tem apenas um pino de saída, ao passo que um terminal de saída tem apenas um pino de entrada. As sua representações gráficas podem ser vistas nas Figuras 12 e 13.

Figura 12: Representação gráfica de um terminal de entrada.



Fonte: Ashour et al. (1998b)

Figura 13: Representação gráfica de um terminal de saída.

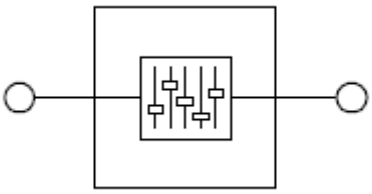


Fonte: Ashour et al. (1998b)

Diferentemente dos terminais, *units* têm sempre entrada e saída, e representam controles e transformações nos canais de áudio que são implementados no dispositivo e podem ser manipulados pelo *host*. Um exemplo é a *feature unit*, que possibilita ao *host* um controle de propriedades básicas do áudio. A representação gráfica de uma *feature unit* pode ser vista na Figura 14. O conjunto de características manipuláveis em uma *feature unit* é listado abaixo [Ashour et al. (1998b)]:

- Volume
- *Mute*
- Controle de tom (baixos, médios, agudos)
- Equalizador gráfico
- Controle automático de ganho
- *Delay*
- *Bass boost*
- *Loudness*

Figura 14: Representação gráfica de uma *feature unit*.



Fonte: Ashour et al. (1998b)

Sinais de áudio em sistemas USB podem ter diferentes formatos. Esses formatos são agrupados em três tipos:

- Tipo I: sinais de áudio que são construídos amostra por amostra, não comprimindo informações de múltiplas amostras em outros blocos de informação. Um exemplo desse tipo de formato é o PCM.
- Tipo II: sinais de áudio que não preservam a noção de canais físicos separados durante a transmissão. Tipicamente, compreende todos os formatos que não são do tipo PCM, e que juntam informações de diversos canais para a transmissão, de modo que possam ser reconstruídos posteriormente.
- Tipo III: sinais de áudio que não se encaixam nas descrições dos tipos anteriores. Um ou mais sinais de áudio PCM são convertidos em "amostras pseudo-estéreo"e transmitidos, para serem recuperados no *host*.

Sinais do tipo I de múltiplos canais seguem um esquema de ordenamento de transmissão. Primeiramente, as amostras para todos os canais em um determinado instante de tempo t_0 são transmitidas, seguidas pelas amostras para todos os canais para o instante t_1, assim por diante. A ordem dos canais deve ser mantida para todos os instantes de tempo.

3 SOLUÇÃO PROPOSTA E MATERIAIS

3.1 Visão geral

Conforme exposto na introdução do presente relatório, durante o presente projeto, foi desenvolvido um sistema de aquisição de áudio de múltiplos canais. Este desenvolvimento se deu no Laboratório de Processamentos de Sinais e Imagem (LaPSI) da Universidade Federal do Rio Grande do Sul (UFRGS).

O sistema foi concebido para ter as seguintes atribuições: aquisição de áudio de múltiplos canais e envio do áudio de todos os canais de forma sincronizada para um processador. Para a aquisição do sinal de áudio, optou-se por utilizar uma placa com seis microfones PDM.

A utilização de microfones PDM apresenta diversas vantagens. Comparado a microfones analógicos, ele possibilita uma interface simplificada com maior robustez a interferência de ruídos [Cirrus (2015b)]. Já se comparado com outras codificações digitais, por exemplo o PCM, ele apresenta a vantagem de utilizar apenas um bit e ter uma implementação mais simples [Kite (2012)].

No entanto, a maioria dos sistemas utiliza o formato PCM para representar sinais de áudio [Hoffmann (2002)], por ser de mais fácil manipulação e permitir operações como mixagem, filtragem e equalização [Kite (2012)]. Entre as variedades de sinal PCM, a mais utilizada é o LPCM (*Linear PCM*), onde os intervalos de quantização são igualmente espaçados [Hoffmann (2002)].

Por essa razão, optou-se por projetar o dispositivo descrito no presente relatório para realizar a transcodificação dos sinais de áudio PDM proveniente da placa de aquisição de áudio para sinais LPCM. Essa transcodificação é realizada no dispositivo para os sinais de cada um dos microfones. Nesse relatório, o formato de saída do áudio é referido somente como PCM e não como LPCM, de forma que permaneça subentendido que o sinal de saída do sistema apresenta intervalo de quantização constante (linear).

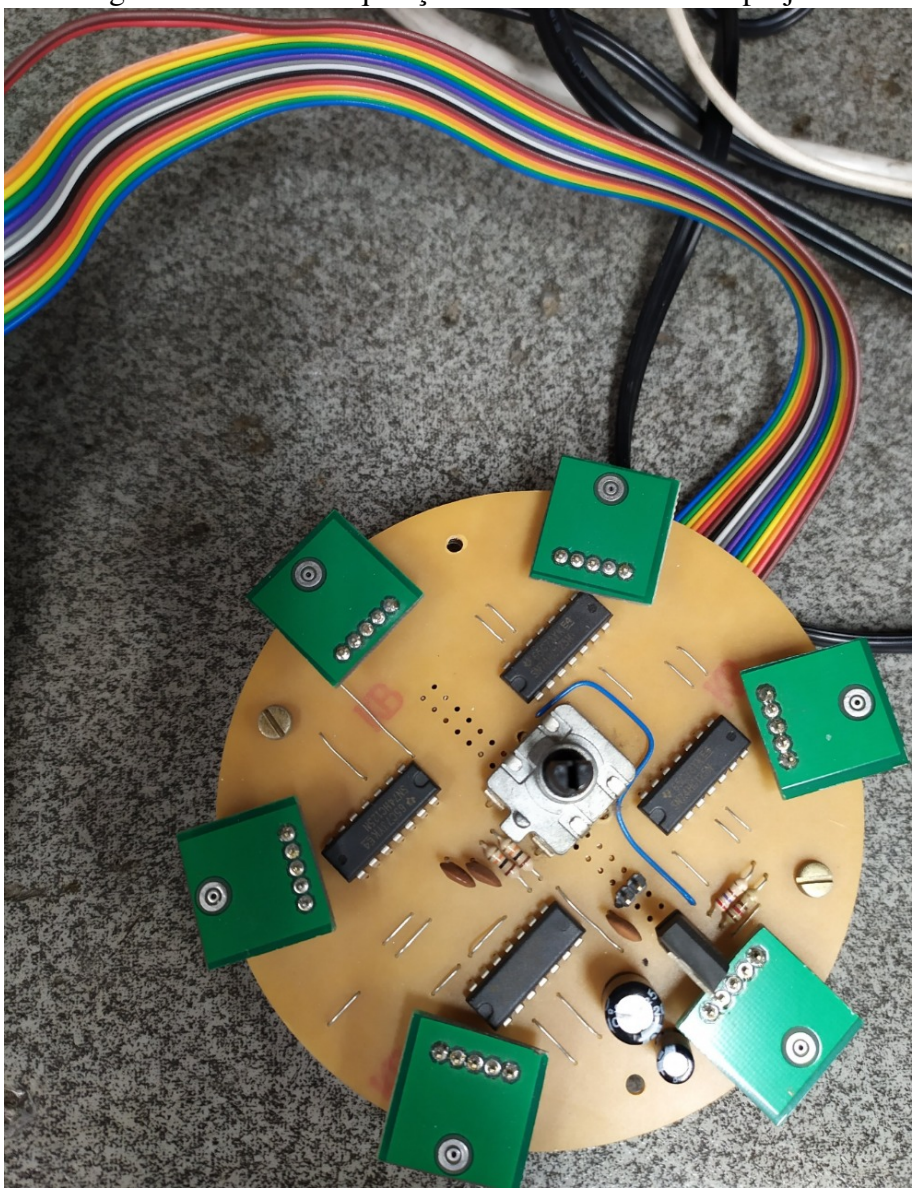
O envio dos sinais PCM de áudio é feito via protocolo USB. Esse protocolo foi escolhido por ser a interface de comunicação mais utilizada em computadores atualmente [Axelson (2009)]. Além disso, apresenta a vantagem de definir diversos tipos de dispositivos em suas especificações, incluindo dispositivos de áudio.

Para a interface com o sistema de aquisição de áudio, optou-se utilizar uma FPGA. Além de ter múltiplas entradas e saídas que podem ser configuradas livremente conforme a aplicação, apresenta também a vantagem de possibilitar processamento paralelo, o que é útil nesse projeto uma vez que são utilizados seis microfones. Além disso, o fato de definir livremente a arquitetura de *hardware* para a aplicação possibilita obter o circuito mais adaptado e eficiente para o presente projeto de forma específica.

A aquisição de áudio do sistema é realizada por seis microfones PDM de modelo

WM7236E. Tanto a escolha pelo tipo de microfone, quanto pelo modelo ou pela sua quantidade não foram feitas nesse projeto, mas sim em projetos anteriores realizados no LaPSI. Em projetos passados, já havia sido desenvolvida uma placa contendo seis microfones desse modelo e dispostos no perímetro de um círculo igualmente espaçados. Uma foto dessa placa pode ser vista na Figura 15.

Figura 15: Placa de aquisição de áudio utilizada no projeto



Fonte: Autor.

Uma vez que essa placa não foi desenvolvida no presente projeto, não é o objetivo do presente relatório justificar as escolhas realizadas no desenvolvimento da placa de aquisição de áudio. Ao invés disso, os componentes da placa serão tratados como pressupostos do projeto. O presente relatório se limita a descrever a sua topologia nos aspectos que são relevantes para o presente projeto, bem como explicar o seu modo de funcionamento.

A placa de microfones é constituída por seis microfones PDM de modelo WM7236E, regulador de tensão e conversores de nível. Além disso, possui um *encoder* em seu centro,

que no entanto não é utilizado no escopo do presente projeto. Ela se conecta ao exterior por um barramento de 14 fios paralelos, além de outros dois fios para a alimentação.

Cada microfone PDM da placa apresenta cinco pinos [Cirrus (2016)]:

- CLK - entrada de *clock* do microfone.
- LRSEL - seleciona o canal direito ou esquerdo.
- GND - referência de tensão.
- VDD - alimentação.
- DAT - saída de dados do microfone.

A alimentação do microfone pode ser feita na faixa de 1,62V a 2V [Cirrus (2016)]. Para que a placa possa ser diretamente alimentada por uma fonte externa com tensões mais elevadas (por exemplo 3,3V ou 5V), ela possui um regulador de tensão para gerar uma tensão de 1,8V a partir da alimentação externa, a qual e compatível com as especificações dos microfones PDM.

A frequência do sinal de entrada de *clock* do microfone define a sua zona de operação. Quando o microfone recebe um sinal de 2,4MHz a 4,9MHz, ele opera em modo *Hi-Fi Record*, em que ele apresenta melhores performances em termos largura de banda e SNR.

Já com uma frequência na faixa entre 300kHz e 800kHz, o microfone opera em modo *Voice*, onde o consumo de corrente cai significativamente e há perda na performance do microfone em termos de largura de banda e SNR. Uma comparação entre os diferentes modos pode ser visto na Tabela 2.

Tabela 2: Tabela comparativa entre modos de funcionamento do microfone WM7236E.

Modos	Corrente	Largura de banda	SNR
Hi-Fi Record	980 μA	20Hz a 22kHz	63dB
Voice	280 μA	200Hz a 8kHz	60dB

Fonte: [Cirrus (2015a)].

A saída de dados do microfone é o pino por onde é enviado o sinal áudio em modulação PDM. Já o pino LRSEL pode ser utilizado para seleção do canal L ou R em um sistema estéreo. Como essa funcionalidade do microfone não é explorada no projeto, na placa ele é conectado diretamente à alimentação (VDD).

A placa se conecta ao exterior por um barramento de quatorze fios paralelos, além de mais dois fios par alimentação. Entre esses quatorze fios, doze são conectados aos microfones. Para cada microfone, existe um fio conectado à sua entrada de *clock* e um fio conectado à sua saída de dados. Os outros dois fios do barramento são conectados ao *encoder* presente na placa, o qual não é de interesse do presente projeto.

Para o funcionamento dessa placa de aquisição de áudio, é necessário portanto uma alimentação externa, assim como o envio de seis sinais de *clock* na frequência desejada para o sinal PDM. Os sinais de saída dos microfones podem ser recuperados pelo barramento de fios paralelos.

Os fios para conexão externa da placa de aquisição de áudio são do tipo fêmea, podendo ser conectado a uma barra de pinos do tipo macho. Nesse projeto, eles foram conectados à barra de pinos do kit FPGA Xilinx XUPV5.

O kit FPGA escolhido apresenta uma FPGA Virtex-5 e diversos outros componentes, alguns deles sendo também utilizados no presente projeto. Esse kit foi utilizado por apresentar uma FPGA com características que foram julgadas suficientes para realizar a transcodificação dos sinais dos microfones. Além disso, ela contém um controlador USB Cypress *Ez-Host*, que implementa o protocolo USB em *hardware*. Uma vez que o protocolo USB foi escolhido para o envio dos dados, o fato de haver um controlador USB no kit remove a necessidade de implementar o protocolo na FPGA ou de adquirir outro controlador USB. Outra razão para a escolha do kit é que a placa de aquisição de áudio já havia sido previamente projetada para se conectar diretamente à barra de pinos existente no kit Xilinx XUPV5.

Conforme mencionado anteriormente, o sistema embarcado no kit Xilinx XUPV5 desenvolve as funções de transformar os sinais de áudio PDM dos microfones em PCM e enviá-los via USB. O sistema apresenta, portanto, dois módulos com objetivos diferentes: o módulo de transcodificação PDM-PCM e o módulo de comunicação.

O módulo de transcodificação PDM-PCM tem as seguintes funções: realizar a amostragem do sinal dos microfones PDM, gerar o *clock* para os microfones e realizar a transcodificação de PDM para PCM de cada um dos sinais. Esse módulo é embarcado inteiramente na FPGA, sendo descrito em VHDL. Cada uma das funções é realizada por um componente diferente no código VHDL. No caso dos componentes de amostragem e transcodificação, existem seis instâncias diferentes, uma vez que é necessário uma instância para cada microfone.

O módulo de comunicação tem como objetivo implementar o protocolo USB para envio das informações. O controlador USB Cypress *Ez-Host* presente no kit Xilinx XUPV5 é o responsável por implementar o protocolo. O controlador USB foi programado, através de um *firmware* em linguagem C, para se comportar como um dispositivo USB de classe áudio com transferências do tipo *isochronous*.

O envio dos sinais de áudio PCM dos microfones da FPGA para o controlador USB é feito através da interface HPI (*Host Port Interface*) do controlador USB. Essa interface utiliza um protocolo definido pelo próprio controlador USB [Cypress (2003)]. Embora o controlador USB apresente outras interfaces de comunicação, escolheu-se a interface HPI por apresentar um barramento paralelo de 16 bits, de modo a permitir maior velocidade de comunicação.

Para implementar o protocolo HPI na FPGA, o sistema apresenta um componente específico para esse propósito em VHDL. Os sinais de saída do módulo de transcodificação PDM-PCM, os quais são os sinais de áudio dos seis microfones em formato PCM, passam por esse componenente de interface HPI, que envia os sinais para o controlador USB. O módulo de comunicação, portanto, é formado pelo componente de interface HPI contido na FPGA e pelo controlador USB Cypress *Ez-Host*.

Nas próximas seções, os dois módulos são apresentados detalhadamente. São apresentados os diagramas de blocos de cada módulo, assim como a sua implementação em VHDL e em código C.

3.2 Módulo de transcodificação PDM-PCM

Conforme mencionado anteriormente, o módulo de transcodificação PDM-PCM apresenta três componentes VHDL distintos, cada um com uma função diferente:

- *pdm_clk_generator*: gera o *clock* para a entrada dos microfones PDM a partir de um *clock* de 100MHz.

- *sampler*: realiza a amostragem do sinal PDM proveniente de um microfone.
- *pdm_pcm_converter*: realiza a transcodificação do sinal PDM proveniente de um microfone para PCM.

3.2.1 Gerador de *clock*

O componente *pdm_clk_generator* tem por objetivo gerar o sinal de *clock* de entrada dos microfones. Uma vez que o consumo de corrente dos microfones não é um parâmetro que se busca otimizar no presente projeto, optou-se por utilizar os microfones no modo *Hi-Fi Record*, para obter a melhor qualidade de áudio possível. Logo, a frequência do sinal de *clock* gerado por esse componente está na faixa entre 2,4MHz e 4,9MHz.

A frequência escolhida para o *clock* foi de 3,84MHz. Essa escolha levou em conta a frequência de amostragem desejada para o sinal de saída e a capacidade de dividir a frequência de *clock* de 100MHz que é gerada no kit FPGA.

Para o sinal PCM de saída do módulo de transcodificação PDM-PCM, existem algumas frequências de amostragem que são utilizadas como padrão. As frequências de 44,1 kHz e 48 kHz são padrões para CD e áudio profissional, respectivamente, e permitem a representação de todo o espectro audível (20 Hz - 20 kHz). Outras frequências de amostragem mais baixas podem ser utilizadas para representar a voz humana, como 16 kHz ou 8 kHz [Self (2010)].

No presente projeto, optou-se por gerar sinais PCM amostrados a 48 kHz. O padrão de CD (44,1 kHz) não foi escolhido por causa da maior dificuldade de se obter esta frequência a partir de um *clock* de 100 MHz. A divisão da frequência do sinal de *clock* é feita normalmente por operações que dividem por números inteiros. No entanto, é possível, a partir de blocos de *hardware* com PLL (*Phase Locked Loop*), realizar divisões por razões de números inteiros.

Para se obter a frequência de 44,1 kHz a partir de 100 MHz, seria necessário efetuar uma divisão pelo fator $1000000/441$. Já para alcançar a frequência de 48 kHz, o fator de divisão é de $6250/3$. Uma vez que o denominador do fator de divisão de frequência para 48 kHz é muito menor, ela foi escolhida como saída do sistema.

Já as frequências de 16 kHz e 8 kHz foram preteridas por não representar toda o espectro de frequência audível. No entanto, em uma linha possível de trabalhos futuros, é possível implementar essas frequências de amostragem de saída. Para as frequências de 16 kHz e 8 kHz, essa implementação pode ser realizada com filtros digitais decimadores que reduzam a frequência de amostragem de 48 kHz por um fator de 3 e 6, respectivamente.

O processo para se chegar da frequência de 100 MHz para 48 kHz é constituído de duas etapas. A primeira é a geração do *clock* enviado para os microfones PDM, que utiliza o *clock* de 100 MHz. Essa operação de divisão pode ser feita por um número inteiro ou por uma razão de números inteiros. A segunda etapa consiste na redução da frequência de amostragem do sinal PDM para a frequência de amostragem do sinal PCM. Essa redução é feita pelos filtros decimadores do componente *pdm_pcm_converter*, e é feita por uma operação de divisão por um número inteiro.

Essas limitações não permitem muitas escolhas possíveis para a frequência utilizada para o sinal PDM. O valor de 3,84 MHz foi escolhido por cumprir os requisitos apresentados. Primeiramente, ele pode ser obtido a partir do *clock* de 100 MHz através de uma divisão pelo fator $625/24$, o que é feito através do componente *pdm_clk_generator*. Em segundo lugar, ele pode gerar a frequência do sinal PCM de 48 kHz através de uma divisão por 80, o que é feito no componente *pdm_pcm_converter*.

O componente *pdm_clk_generator* foi implementado em dois estágios, cada um dividindo a frequência do *clock* de entrada de 100 MHz por um fator. O primeiro estágio é formado por um bloco DCM (*Digital Clock Manager*), enquanto que o segundo é implementado utilizando um contador em código VHDL.

O DCM na FPGA Virtex-5 é um bloco de *hardware* que oferece diferentes características de manipulação de sinal de *clock*. Entre essas características está a síntese de frequências. Um dos modos de realizar essa operação é através da saída CLKFX do bloco. É possível configurar, através de código VHDL, um multiplicador inteiro M e um divisor inteiro D para compor o fator M/D pelo qual a frequência de entrada é multiplicada para gerar a frequência de saída. O valor de M pode ser configurado como qualquer valor entre 2 e 33, enquanto que o valor de D deve estar na faixa entre 1 e 32 [Xilinx (2012)].

Os valores escolhidos para M e D foram 24 e 25, respectivamente. No código VHDL, isso foi estabelecido através dos parâmetros CLKFX_DIVIDE e CLKFX_MULTIPLY no *generic map* de um componente DCM_BASE. Já no *port map* do componente, foi conectado na porta CLKIN o sinal de entrada do componente *pdm_clk_generator*, isso é, um sinal de *clock* de 100 MHz. Na porta CLKFX foi conectado o sinal de saída de *clock* do DCM, o qual é a entrada da etapa seguinte. Além disso, foi conectada a saída CLK0 na entrada CLKFB. Esta última conexão foi feita para não existir diferença de fase entre o *clock* de entrada e o *clock* de saída [Xilinx (2012)].

Desse modo, a frequência do sinal de *clock* de saída do DCM é de 100 MHz multiplicado por $24/25$, o que equivale a 96 MHz. A segunda etapa então divide essa frequência por 25 para alcançar a frequência desejada de 3,84 MHz.

Para isso, foi implementado um método de divisão de *clock* baseado em um contador. No código VHDL do componente *pdm_clk_generator*, é declarado o sinal *counter* com 8 bits. A cada borda positiva do sinal de saída do DCM, esse contador é incrementado em um processo VHDL, até um valor máximo de 24 (00011000 em binário). Quando ele atinge esse valor máximo, ele retoma o valor 0, para ser incrementado de novo, sucessivamente.

Esse sinal *counter*, por sua vez, controla o sinal de *clock* de saída. Quando *counter* é menor que 12 (00001100 em binário), o sinal de *clock* de saída é igualado a 0. Caso contrário, ele é igualado a 1. Dessa maneira, obtém-se um sinal de *clock* de saída com frequência 25 vezes menor que o sinal proveniente do DCM, o que corresponde a 3,84 MHz.

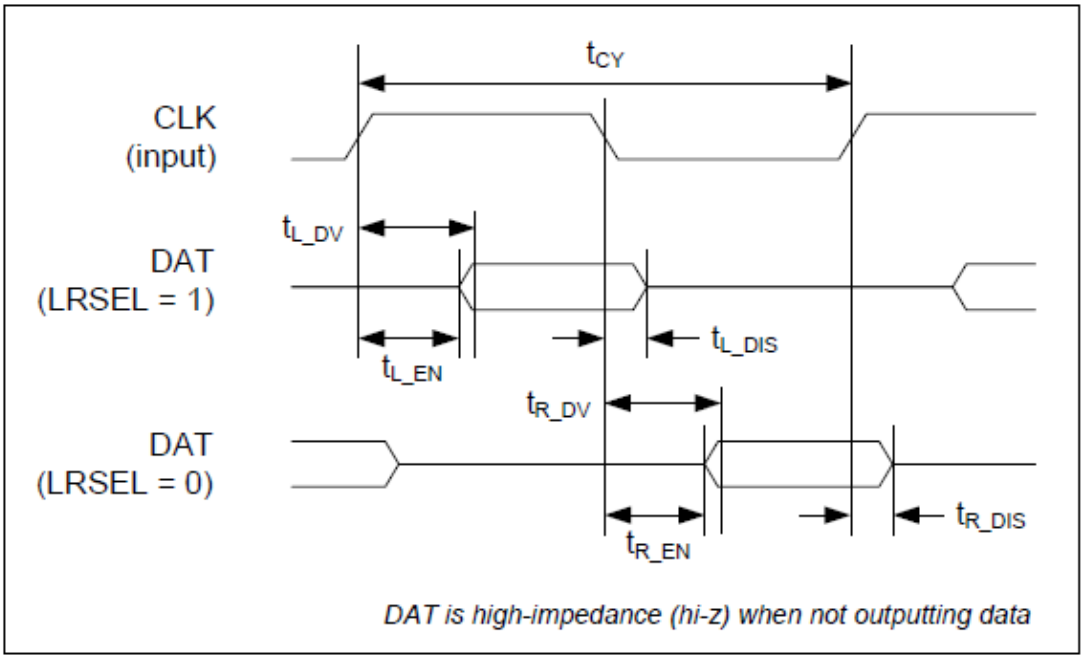
Uma vez implementado o componente *pdm_clk_generator*, ele foi simulado através da ferramenta ISE Simulator, antes de ser implementado e testado na FPGA com o resto do sistema. Os resultados dessa simulação são apresentados no capítulo seguinte.

3.2.2 Sampler PDM

Outro componente VHDL do módulo de transcodificação PDM-PCM é o *sampler*. Esse componente recebe diretamente o sinal de saída dos microfones PDM e amostra eles uma vez a cada ciclo de 3,84 MHz. Essa amostragem é necessária porque o microfone tem um ciclo de tempo específico em que ele disponibiliza o valor do sinal PDM em seu pino DAT. No resto do tempo, o valor do pino é mantido em alta impedância. O diagrama que representa os ciclos do microfone é apresentado na Figura 16.

Uma vez que o pino LRSEL é conectado à alimentação na placa de aquisição de áudio (LRSEL = 1), interessa-se aos valores dos tempos t_{L_EN}, t_{L_DV} e t_{L_DIS}. O primeiro define o momento onde a saída é habilitada pelo microfone. O segundo define o momento onde a saída habilitada pelo microfone é válida. Já o terceiro define o momento onde a

Figura 16: Diagrama do ciclo do microfone PDM.



Fonte: Cirrus (2016).

saída é desabilitada [Cirrus (2016)]. Esses valores são apresentados na Tabela 3.

Tabela 3: Valores dos tempos do ciclo do microfone PDM.

	Min	Typ	Max	Unit
t_{L_EN}	24			ns
t_{L_DV}	30		80	ns
t_{L_DIS}			20	ns

Fonte: [Cirrus (2016)].

A amostragem do sinal é feita a partir do momento que o sinal no pino DAT do microfone é válido. Isso acontece no máximo 80 ns após oa borda de subida do *clock* de entrada. Esse sinal fica habilitado até 20 ns após a borda de descida do sinal de *clock*. Como a frequência do *clock* é 3,84 MHz, o seu período é de 260,417 ns. Ou seja, a borda de descida acontece aproximadamente 130 ns após a borda de subida. Portanto, o intervalo onde o dado permanece disponível na saída do microfone vai de 80 ns até 150 ns depois da borda de subida.

No projeto do componente *sampler*, optou-se por realizar a amostragem do sinal 120 ns depois da borda de subida do *clock*. O componente recebe como entrada o sinal PDM, o sinal de *clock* referente ao sinal PDM e o *clock* geral do sistema de 100 MHz. O sinal de saída do componente é o sinal PDM amostrado.

O código VHDL do componente utiliza um contador de 7 bits que é incrementado a cada ciclo de *clock* do sistema e é zerado quando o *clock* do sinal PDM vai a 0. A amostragem acontece quando o contador chega ao valor 12 (0001100 em binário). Sabendo que o período do clock de 100 MHz é 10 ns, a amostragem acontece aproximadamente 120 ns depois da borda de subida do *clock* PDM. O valor desse tempo não é exato uma

vez que a borda de subida do *clock* geral do sistema e a borda de subida do *clock* PDM não acontecem ao mesmo tempo. No entanto, é garantido que a amostragem acontece no período permitido, isso é, entre 80 ns e 130 ns após a borda de subida do *clock* PDM.

3.2.3 Arquitetura do transcodificador PDM-PCM

Com os dois componentes do módulo de transcodificação PDM-PCM, é possível obter o sinal PDM dos microfones corretamente amostrado. O terceiro componente, o *pdm_pcm_converter*, é o mais complexo do módulo. Para transformar o formato do sinal de áudio de PDM para PCM, ele é constituído por uma sequência de filtros de decimação implementados em VHDL.

O sinal de áudio em formato PDM pode ser visto apenas como um sinal de áudio sobreamostrado de 1 bit [Kite (2012)]. Observando-se dessa maneira, um transcodificador PDM-PCM tem a função de reduzir a frequência de amostragem do sinal e aumentar a quantidade de bits, além de filtrar o ruído em alta frequência. Logo, ele deve agir como um filtro decimador, o qual tem a função essencial de reduzir a frequência de amsotragem e manter o nível de *aliasing* na banda passante abaixo de um limite prescrito [B. Hogenauer (1981)].

Para calcular a quantidade de bits efetiva que se pode recuperar no sinal PCM na saída do componente, utiliza-se a SNR do sinal PDM do microfone. Para o modo *Hi-Fi Record*, a SNR é de 63 dB [Cirrus (2016)]. Utilizando a equação 4, calcula-se a quantidade de bits do sinal PCM que tenha a mesma SNR. Esse cálculo é dado na equação 14.

$$N = \frac{SNR - 4,8}{6,0} = 9,7 \tag{14}$$

Logo, a quantidade mínima de bits requerida no sinal PCM é 10. No entanto, optou-se por utilizar um sinal PCM de 16 bits, uma vez que é o padrão para sinais de áudio de CD [Kite (2012)][Hoffmann (2002)].

A arquitetura escolhida para o componente *pdm_pcm_converter* é formada por três filtros de decimação em cascata. O primeiro, que recebe diretamente o sinal PDM, é um filtro CIC (*Cascaded Integrator Comb*). Os dois filtros seguintes são filtros do tipo *halfband* decimadores. Essa arquitetura é amplamente adotada em aplicações similares [Hegde (2010)], e foi escolhida por aliar performance a simplicidade de implementação.

3.2.4 Projeto do filtro CIC

O filtro CIC foi utilizado no presente projeto por ser um método eficiente, em termos de *hardware*, de implementar um filtro de decimação. Ele não requer multiplicadores e utiliza uma capacidade de armazenamento pequena [B. Hogenauer (1981)]. No entanto, a largura da banda passante e a resposta em frequência do filtro fora da banda passante são limitadas. Por isso, são utilizados filtros FIR convencionais na região de baixa frequência de amostragem para modelar melhor a resposta em frequência. Dessa maneira, o filtro CIC é utilizado em altas frequências de amostragem, onde a economia em *hardware* é crítica, e um filtro convencional é utilizado em baixas frequências de amostragem, onde o número de multiplicações por segundo é baixa [B. Hogenauer (1981)].

Conforme explicado no capítulo de revisão bibliográfica, o projeto do filtro CIC se resume a três parâmetros: o fator de decimação R, o atraso dos blocos *comb* M e a ordem do filtro N. Além disso, para garantir o funcionamento do filtro, é necessário calcular a quantidade miníma de bits a ser utilizada no sinal de saída dos integradores. No entanto, esse número de bits não afeta a resposta em frequência do filtro.

O fator de decimação R é definido pelo fator de decimação desejado para todo o componente *pdm_pcm_converter* e pela arquitetura utilizada. O fator de decimação total do componente, conforme já explicitado, é igual a 80. A arquitetura em cascata do componente resulta num fator de decimação total igual a multiplicação entre os fatores de decimação de cada um dos blocos. Uma vez que os filtros *halfband* decimadores tem fator de decimação igual a 2, tem-se que o fator de decimação do filtro CIC é dado pela equação 15.

$$R_{CIC} = \frac{80}{R_{HB1} \times R_{HB2}} = \frac{80}{2 \times 2} = 20 \tag{15}$$

A ordem N do filtro CIC é definida pela ordem do modulador sigma-delta contido no microfone PDM. Uma vez que o modulador do microfone é de quarta ordem, o filtro CIC foi projetado com quinta ordem [Hegde (2010)].

Já o atraso diferencial M do filtro CIC é geralmente definido como $M = 1$ ou $M = 2$ [B. Hogenauer (1981)]. Uma vez que o aumento de M provoca um aumento na atenuação na banda de passagem do filtro, optou-se por utilizar $M = 1$.

Por fim, o número de bits na saída dos integradores é calculado de acordo com a equação 13. Com os valores de R e M já definidos, o valor de B_{in}, ou seja, o número de bits do sinal de entrada, foi definido como 1, uma vez que o filtro CIC recebe como entrada o sinal PDM. Desse modo, o valor obtido para o número de bits necessários na saída dos integradores é dado pela equação 16.

$$B_{max} = [N \log_2(RM) + B_{in} - 1] = 5 \log_2(20) + 1 - 1 = 21,6 \tag{16}$$

Com o valor calculado, obtém-se que o número mínimo de bits necessário são 22 bits. Portanto, escolheu-se utilizar 24 bits (3 bytes, para um byte de 8 bits) na saída de todos os integradores do filtro CIC. Essa quantidade foi também utilizado na saída dos blocos *comb*. Uma vez que o sinal de saída do componente tem 16 bits, um truncamento é realizado após o último bloco *comb*. Dessa forma, o sinal de entrada dos filtros *halfband* decimadores já tem 16 bits.

Em código VHDL, o filtro CIC foi representado pelo componente *cic_filter*. Dentro dele, outros três componentes foram utilizados: integrador, decimador e *comb*. Tanto o integrador como o *comb* recebem como entrada um sinal de 24 bits e um sinal de *clock*, e tem como saída um sinal de 24 bits.

O integrador se baseia em um processo VHDL que contém uma variável *sum* definida como tipo *signed* com 24 bits. A cada borda positiva do *clock* de entrada, essa variável soma o valor do sinal de entrada ao seu próprio valor. O sinal de saída assume o valor da variável *sum*.

A soma feita dentro do processo VHDL utiliza a biblioteca IEEE.NUMERIC_STD e é implementada utilizando complemento de 2. Quando o resultado da soma ultrapassa o valor máximo que pode ser representado para o número de bits da variável, ocorre o *wrap-around*. Essa propriedade considera uma transição entre o número mais positivo representável e o número mais negativo representável como a soma de uma unidade. Em outras palavras, a soma continua a contar a partir do menor número da escala. Essa propriedade satisfaz a condição imposta para o funcionamento do filtro CIC, conforme explicado no capítulo de revisão bibliográfica.

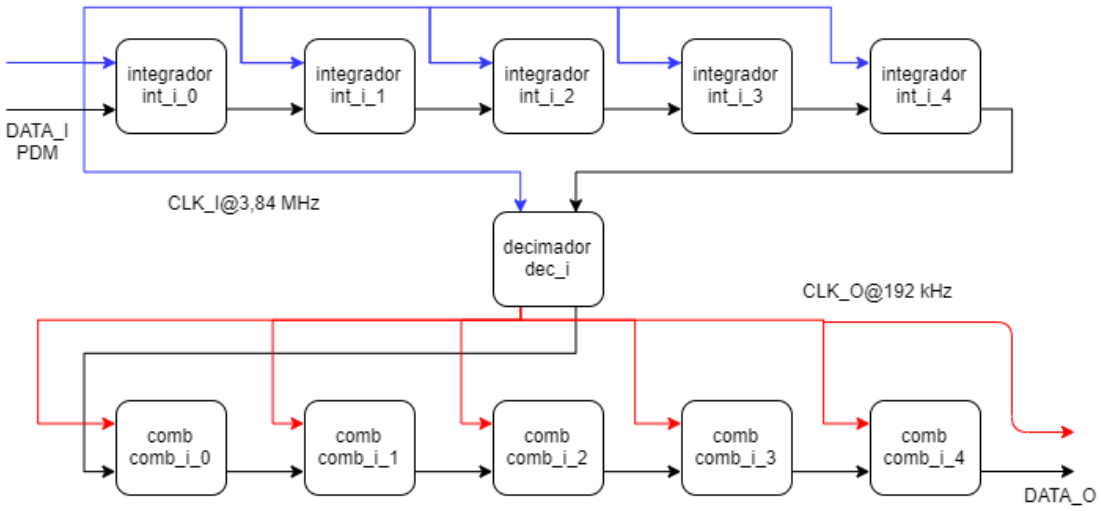
No componente *comb*, existe também um processo VHDL com uma variável *previous* do tipo *signed* e com 24 bits. A cada borda positiva do *clock* de entrada, o sinal de saída

assume o valor da subtração entre o sinal de entrada e a variável *previous*, que em seguida assume o valor do sinal de entrada que fica registrado para o ciclo seguinte.

O componente decimador tem como função reduzir a frequência de amostragem do sinal por 20. Ele recebe como entrada um sinal de *clock* e um sinal de dados, que corresponde à saída do último integrador. Ele tem como saída um sinal de *clock* e um sinal de dados, que corresponde à entrada do primeiro *comb*. Ele implementa um processo VHDL que incrementa um sinal *counter* de 8 bits a cada borda positiva do sinal de *clock* de entrada. Quando *counter* atinge o valor 9 (00001001 em binário), o sinal de *clock* de saída é igualado a 1, e o sinal de saída é igualado ao de entrada. Já quando *counter* atinge o valor 19 (00010011 em binário), o sinal de *clock* de saída é igualado a 0, e o sinal *counter* é igualado a 0 também, para que possa ser incrementado no próximo ciclo. O resultado é um sinal de *clock* de saída com frequência 20 vezes menor que o de entrada, e o sinal de dados de saída correspondendo ao de entrada a cada borda de subida do *clock* de saída.

O *cic_filter* possui cinco integradores, cinco *combs* e um decimador. O modo como eles são conectados em seu código VHDL pode ser visto na Figura 17. Os caminhos em azul representam o *clock* de entrada de alta frequência (3,84 MHz). Os caminhos em vermelho representam o *clock* depois de passar pelo decimador, com frequência 20 vezes menor. Os outros caminhos representam o sinal de dados à medida que ele passa pelos componentes do filtro CIC.

Figura 17: Diagrama de blocos do filtro CIC



Fonte: Autor.

Além dos componentes da Figura 17, o *cic_filter* desempenha outras funções. Primeiramente, é necessário fazer a adaptação do sinal de entrada para o primeiro integrador, uma vez que o sinal de entrada do integrador tem 24 bits e o sinal de entrada do *cic_filter* tem apenas 1. Além disso, o sinal PDM de entrada, por variar entre 0 e 1, tem um nível DC intrínseco ao próprio formato PDM. Para suprimi-lo, considera-se que o sinal de entrada 0 corresponde a -1. Dessa forma, o sinal de entrada do primeiro integrador é um sinal de 24 bits que pode assumir os valores -1 e 1. Isso é feito por um processo VHDL no *cic_filter*, que atribui o valor -1 para o sinal de entrada do primeiro integrador quando o sinal de entrada do *cic_filter* tem valor 0, e atribui o valor 1 quando o sinal de entrada tem valor 1.

Além disso, o sinal de saída do último *comb* é truncado antes da saída do filtro CIC. Esse truncamento consiste em selecionar apenas os 16 bits MSB do sinal de 24 bits original. O sinal gerado por esse truncamento é a saída do filtro CIC, e é acompanhado com o seu sinal de *clock* com frequência igual a sua frequência de amostragem.

3.2.5 Projeto do filtro *halfband* decimador

Além do filtro CIC, o outro filtro utilizado no *pdm_pcm_converter* foi o *halfband* decimador. Esse filtro tem como objetivo dividir a frequência de amostragem do sinal de entrada pela metade, mas sem induzir *aliasing* no sinal de saída. Ele é formado por um filtro FIR (*Finite Impulse Response*) seguido de um bloco decimador, que seleciona uma a cada duas amostras do sinal de entrada.

Considerando um sinal de entrada de frequência de amostragem f_s, sabe-se pelo teorema de Nyquist que ele representa frequências de até $f_s/2$. Se deseja-se reduzir a sua frequência de amostragem pela metade, o sinal resultante passará a representar frequências de até $f_s/4$. Logo, o filtro FIR do *halfband* decimador tem o objetivo de atenuar ao máximo o sinal de entrada no intervalo entre $f_s/4$ e $f_s/2$ e ter resposta em frequência com mínimo *ripple* até $f_s/4$. Outra exigência para um filtro *halfband* decimador é que o fim da banda passante e o início da banda de rejeição sejam equidistantes de $f_s/4$ [Hegde (2010)].

Mesmo que existam dois filtros *halfband* decimadores no *pdm_pcm_converter*, foi necessário calcular os coeficientes do filtro apenas uma vez. Embora as frequências absoltuas dos dois filtros sejam diferentes, a resposta em frequência relativa à frequência de amostragem do sinal de entrada é a mesma. Logo, os mesmos coeficientes podem ser utilizados para os dois filtros FIR.

Os coeficientes do filtro foram projetados utilizando a ferramenta de *software* MATLAB, com as funções *fir1* e *kaiserord*. A função *fir1* calcula os coeficientes do filtro FIR utilizando o método de janelamento. A função recebe como parâmetro a ordem do filtro, a frequência de corte, o tipo de filtro (passa-baixa, passa-alta ou passa-faixa) e os coeficientes da janela utilizada.

Já a função *kaiserord* calcula os parâmetros necessários para a função *fir1* utilizando uma janela do tipo Kaiser. A função *kaiserord* recebe como parâmetros os vetores *f* e *a*, que indicam a resposta em frequência desejada para o filtro projetado. O terceiro parâmetro é o vetor *dev*, que indica o *ripple* desejado na banda passante e a atenuação na banda de rejeição. Como resultado, a *kaiserord* calcula a ordem necessária para o filtro, a frequência de corte, o tipo de filtro e o valor beta que é usado para o cálculo da janela Kaiser. Os coeficientes da janela são calculados a partir do parâmetro beta e da ordem do filtro com a função *kaiser*.

Como parâmetro para a função *kaiserord*, definiu-se uma banda passante de 0 a $0,2f_s$ e uma banda de rejeição de $0,3f_s$ a $0,5f_s$. Definiu-se também um *ripple* de 0,1% em banda passante e atenuação em banda de rejeição de 60dB. Como resultado da função *kaiserord*, obteve-se uma ordem de 40, frequência de corte de $0,25f_s$ e um beta de $6,1819$. Esses parâmetros foram utilizados na função *fir1* para cálculo dos coeficientes do filtro FIR.

O resultado da função *fir1* foi uma sequência de coeficientes contidos no intervalo $[-0,5; 0,5]$, representados como valores em ponto flutuante. Na implementação do filtro em *hardware*, esses coeficientes foram representados com um número em ponto fixo. Optou-se por utilizar todos os bits como fracionários na representação. Dessa forma, o ponto fixo é posicionado à esquerda de todos os dígitos binários.

Os coeficientes em ponto fixo representados com n bits podem ser calculados, pela multiplicação do coeficiente original por 2^n, o arredondamento para o número inteiro mais próximo, e a divisão por 2^n. O intervalo que pode ser representado dessa maneira vai de $-0,5$ a $(0,5 - 1/2^n)$. Todos os coeficientes originais com valor de 0,5 foram igualados ao valor máximo da escala para poderem ser representados.

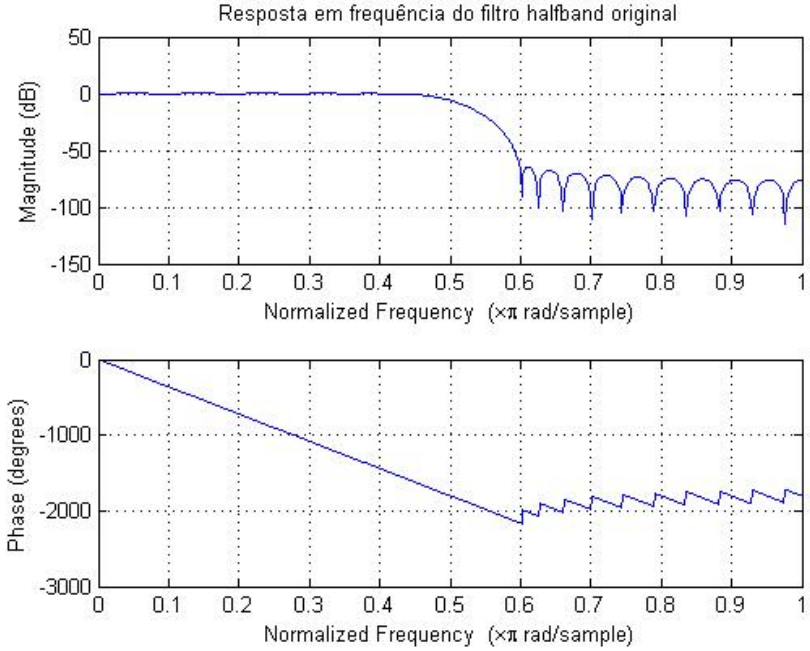
A representação em ponto fixo dos coeficientes gera uma distorção na resposta em frequência. A escolha pelo número de bits foi feita calculando-se as respostas em frequências dos filtros com coeficientes arredondados e comparando com a resposta em frequência do filtro original.

Para a escolha da quantidade de bits utilizada para representar os coeficientes do filtro, foram calculadas as respostas em frequência do filtro para 8, 12, 16, 20 e 24 bits. Esse cálculo foi feito utilizando a ferramenta MATLAB, com a função *freqz*.

Em seguida, elas foram comparadas graficamente com a resposta em frequência do filtro original. A resposta em frequência de cada um dos filtros pode ser vista nas Figuras 18 a 23.

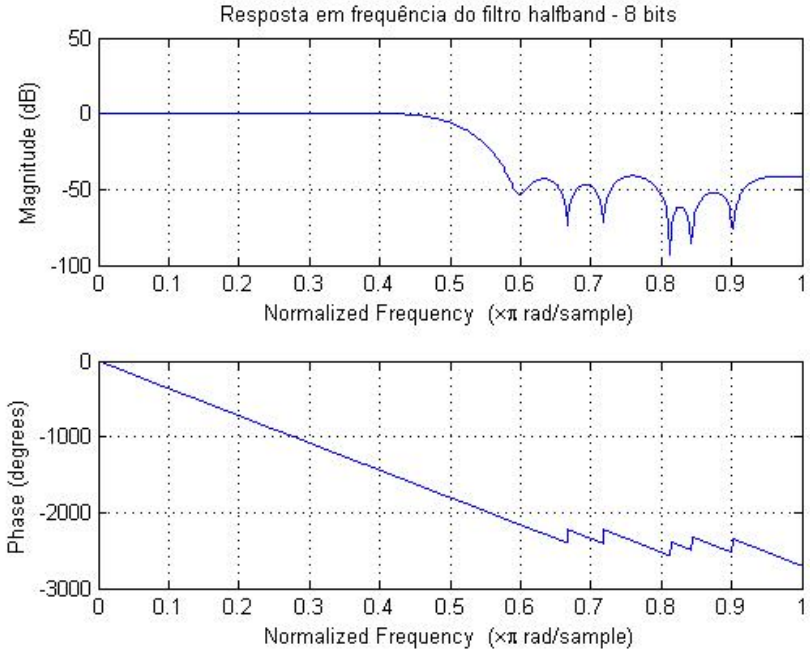
A partir dessas respostas em frequência, foi calculado o *ripple* em banda passante e a atenuação na banda de rejeição. Para o cálculo do *ripple*, foi feita a diferença entre o ganho máximo e o ganho mínimo do filtro na banda passante. Já a atenuação obtida é a atenuação mínima em toda a banda de rejeição. O resultado desses cálculos é mostrado na Tabela 4.

Figura 18: Resposta em frequência do filtro *halfband* decimador representado em ponto flutuante.



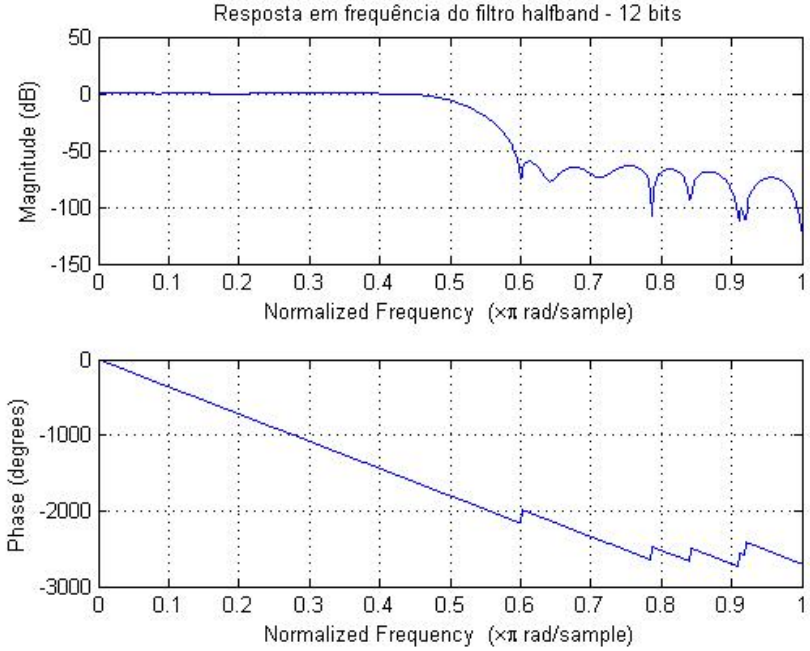
Fonte: Autor.

Figura 19: Resposta em frequência do filtro *halfband* decimador representado com 8 bits.



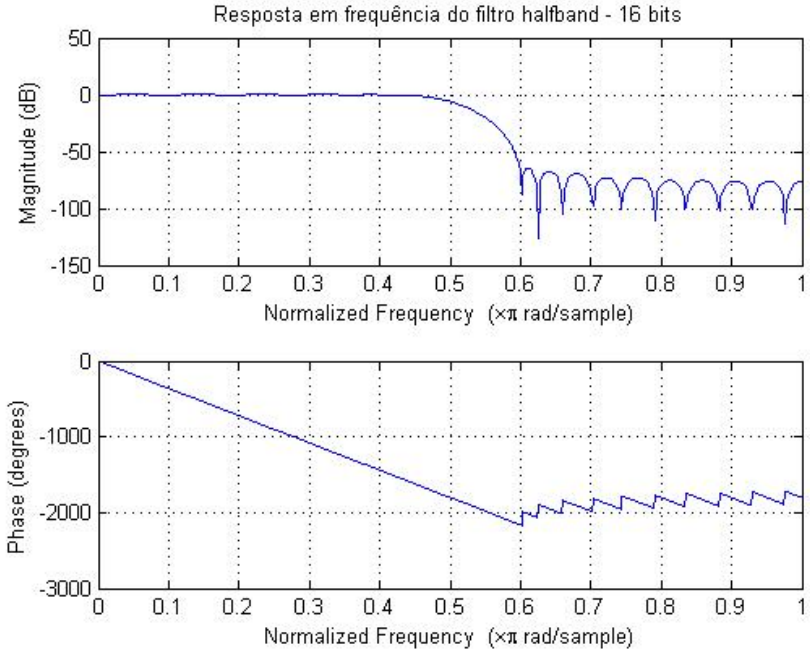
Fonte: Autor.

Figura 20: Resposta em frequência do filtro *halfband* decimador representado com 12 bits.



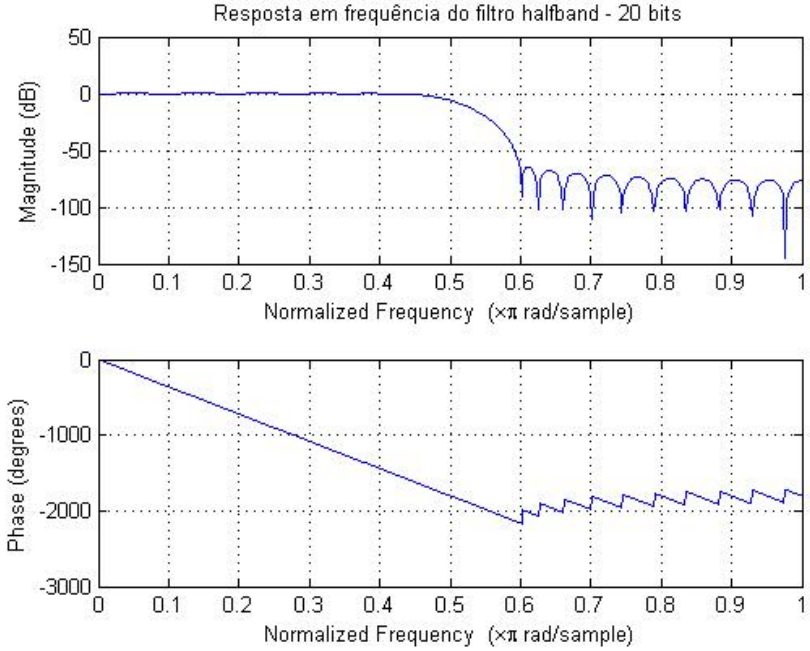
Fonte: Autor.

Figura 21: Resposta em frequência do filtro *halfband* decimador representado com 16 bits.



Fonte: Autor.

Figura 22: Resposta em frequência do filtro *halfband* decimador representado com 20 bits.



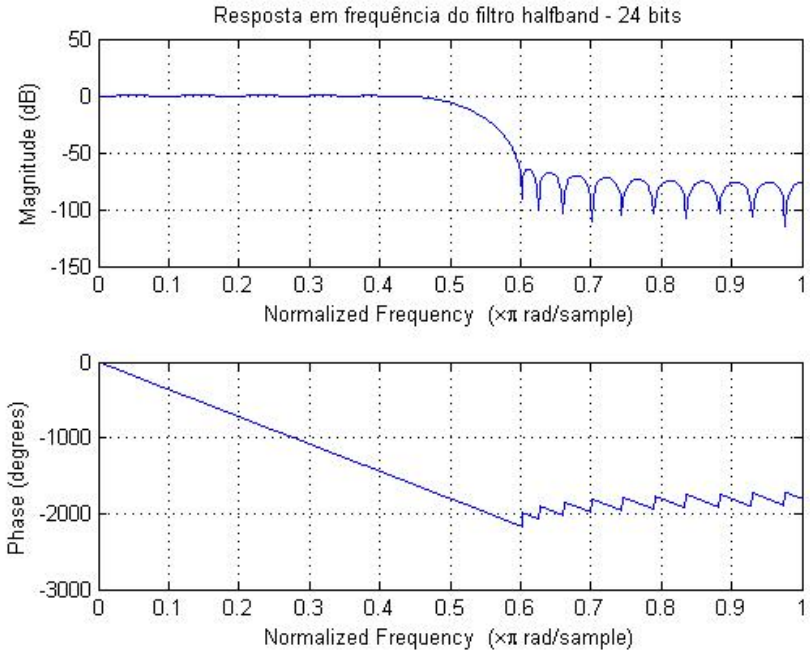
Fonte: Autor.

Tabela 4: Performance do filtro *halfband* decimador com diferentes resoluções em bits.

Resolução	*Ripple* (%)	Atenuação na banda de rejeição (dB)
Original	0,1	64,57
8 bits	1,75	41
12 bits	0,15	59,92
16 bits	0,1	64,73
20 bits	0,1	64,59
24 bits	0,1	64,57

Fonte: Autor.

Figura 23: Resposta em frequência do filtro *halfband* decimador representado com 24 bits.



Fonte: Autor.

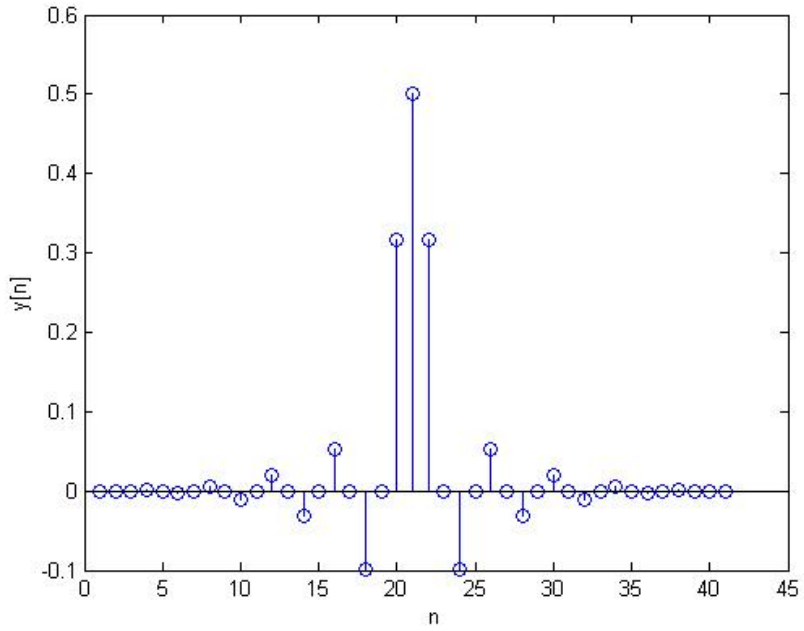
A partir da análise das Figuras 18 a 23 e da Tabela 4, observa-se que o arredondamento dos coeficientes do filtro com 8 e 12 bits aumentam em *ripple* e diminuem a atenuação em banda passante. No entanto, a partir de 16 bits, o *ripple* permanece constante e igual ao do filtro sem arredondamento de coeficientes. Além disso, a atenuação na banda de rejeição varia menos que 1 dB, sempre mantendo-se abaixo do valor de 60 dB definido anteriormente. Portanto, foi escolhido o arredondamento de 16 bits.

Os valores dos coeficientes originais, bem como os valores dos coeficientes do filtro em 16 bits em formato hexadecimal podem ser vistos na Tabela 22, presente nos anexos. A resposta ao impulso do filtro *halfband* decimador utilizado pode ser visto na Figura 24.

Uma vez calculados os coeficientes do filtro, ele foi implementado em VHDL. Uma vantagem da implementação de filtros *halfband* decimadores é a eficiência em *hardware*, uma vez que um a cada dois coeficientes é nulo, à exceção do coeficiente central. Além disso, os seus coeficientes são simétricos [Minzter (1982)].

O filtro projetado de ordem 40 possui 20 coeficientes nulos. Além disso, 10 pares de

Figura 24: Resposta ao impulso do filtro *halfband* decimador.



Fonte: Autor.

coeficientes são iguais. Logo, ele possui apenas 11 coeficientes diferentes.

A equação de recorrência de um filtro FIR genérico pode ser vista na equação 17, onde a ordem do filtro é representada pelo valor N.

$$y[n] = \sum_{k=0}^{N} h[k]x[n-k] \tag{17}$$

Para o filtro *halfband* em questão, a ordem é igual a 40, e todos os coeficientes $h[k]$ são nulos quando k é par. Logo, a equação 17 pode ser reescrita como:

$$y[n] = \sum_{k=0}^{19} h[2k+1]x[n-(2k+1)] \tag{18}$$

Além disso, sabe-se que os coeficientes $h[k]$ são simétricos em torno do coeficiente central, isso é, $k = 20$. Essa relação pode ser expressa matematicamente pela equação 19.

$$h[1] = h[39], h[3] = h[37], ..., h[19] = h[21] \tag{19}$$

Essa relação pode ser expressa de maneira geral pela equação 20.

$$h[2k+1] = h[39-2k], \forall 0 < k < 9 \tag{20}$$

Portanto, a equação 18 pode ser reescrita, resultando nas equações 21 e 22.

$$y[n] = \sum_{k=0}^{9} \Big[h[2k+1]x[n-(2k+1)] + h[39-2k]x[n-(39-2k)]\Big] + h[20]x[n-20] \tag{21}$$

$$y[n] = \sum_{k=0}^{9} \Big[h[2k+1]\Big(x[n-(2k+1)] + x[n-(39-2k)]\Big)\Big] + h[20]x[n-20] \tag{22}$$

Embora as equações 21 e 22 sejam matematicamente equivalentes, a equação 22 utiliza uma operação de multiplicação a menos para cada termo dentro do somatório. Na implementação em *hardware* do filtro *halfband*, isso se traduz na necessidade de um multiplicador a menos.

A implementação deste filtro foi feita no componente VHDL *hb_decimador*. Esse componente recebe como entrada um sinal de *clock* e um sinal de dados, e tem com saída um sinal de *clock* e um sinal de dados. O sinal de *clock* de saída tem a metade da frequência do sinal de entrada, enquanto que o sinal de dados de saída corresponde à convolução entre o sinal de entrada e a resposta ao impulso do filtro. Foi utilizado como base o código disponível em [Surf-Vhdl (2015)].

O componente *hb_decimador* utiliza cinco processos VHDL. O primeiro processo *p_input* armazena os dados de entrada no *buffer p_data* com 41 amostras de 16 bits, descartando sempre a amostra mais antiga quando uma nova amostra chega. O segundo processo *p_sum* realiza a soma das amostras nas posições $2k+1$ e $39-2k$ com k variando de 0 a 9, conforme a equação 22. Esse processo é realizado com um *loop for*. Esses valores são armazenados no *buffer r_data_sum* com 11 amostras. A última amostra do *buffer r_data_sum* recebe a amostra número 20 do *buffer p_data*. Essa amostra é aquela que é multiplicada pelo coeficiente central do filtro.

O terceiro processo *p_mult* multiplica os valores do *buffer r_data_sum* pelo *buffer r_coeff*, também com 11 amostras de 16 bits, onde estão armazenados 11 os coeficientes distintos do filtro. O resultado é armazenado no *buffer r_mult*, contém 32 bits para cada amostra.

O quarto processo *p_acc* soma todos os valores do *buffer r_mult* através de um *loop for* e os armazena na variável *accumulator*, que também tem 32 bits. Por fim, o processo *p_output* gera o sinal de saída a partir da variável *accumulator*. Para isso, é implementado um contador. A cada ciclo de *clock* de entrada, esse contador assume o valor 0 ou o valor 1. A cada vez que ele assume o valor 1, o sinal de *clock* de saída tem uma borda positiva, e o sinal de dados saída é igualado à variável *accumulator* truncada. Quando ele assume o valor 0, o sinal de *clock* de saída assume o mesmo valor, e o sinal de dados de saída mantém o valor anterior. O resultado é um sinal de *clock* de saída de metade da frequência do sinal de entrada, e um sinal de dados de saída que é amostrado a cada borda positiva do sinal de *clock* de saída.

O truncamento da variável *accumulator* consiste em igualar apenas os 16 bits MSB de *accumulator* ao sinal de saída. Esse truncamento é necessário para que o sinal de saída tenha 16 bits, conforme projetado. O valor da variável *accumulator* é o resultado da multiplicação do sinal de entrada de 16 bits pelo coeficiente do filtro de 16 bits. Os coeficientes do filtro foram projetados com ponto fixo, tendo o seu ponto à esquerda de todos os seus 16 bits. A posição do ponto fixo em *accumulator* encontra-se, portanto, à esquerda dos seus 16 bits LSB. Logo, o truncamento de 16 bits de *accumulator* exclui somente bits à direita do ponto fixo, o que equivale a manter somente a parte inteira do sinal.

Com todos os componentes de *pdm_pcm_converter* implementados em VHDL, ele foi simulado utilizando a ferramente ISim do ISE Design Suite. Essa simulação foi feita aplicando-se um sinal PDM com um *clock* de 3,84 MHz na entrada do componente, para observação do sinal PCM de saída e de sua frequência de amostragem. O sinal PDM utilizado foi obtido na simulação em MATLAB, conforme descrito a seguir. Os resultados referentes a esta simulação VHDL são expostos no capítulo de resultados.

3.2.6 Simulação em MATLAB

Além da implementação de todos os componentes VHDL do *pdm_pcm_converter*, foi também realizada uma simulação em MATLAB para atestar o seu funcionamento. Nessa simulação, foi avaliado o sinal de saída do transcodificador PDM-PCM e os seus sinais internos, tanto no domínio tempo quanto no domínio frequência. Embora a simulação permita uma análise do desempenho dos filtros, o seu objetivo principal foi atestar que os filtros projetados estavam funcionais e capazes de produzir um sinal PCM de frequência de amostragem de 48 kHz para um sinal PDM de 1 bit e frequência de amostragem de 3,84 MHz na entrada.

A simulação foi realizada inteiramente em um *script* MATLAB. Todos os blocos do transcodificador PDM-PCM foram na codificados na simulação. Além disso, foi necessário gerar um sinal PDM de entrada. Esse sinal PDM foi gerado por um modulador sigma-delta também presente na simulação. Esse modulador recebeu como entrada na simulação um sinal senoidal, o qual foi reconstruído na saída do transcodificador PDM-PCM.

Todos os sinais da simulação foram representados por um vetor de números de ponto flutuante. Para cada vetor de sinal, foi gerado um vetor tempo correspondente. Cada elemento do vetor tempo define o instante da amostra de mesmo índice do vetor sinal correspondente. O vetor tempo foi gerado com diferença constante entre cada elemento. Essa diferença corresponde ao período de amostragem do sinal.

O primeiro sinal gerado foi o sinal de entrada do modulador sigma-delta. Ele foi definido como uma senoide, para facilitar a análise em domínio frequência do sinal. A frequência da senoide foi definida como 7,5 kHz e foi escolhida arbitrariamente com o único intuito de se encontrar na faixa de frequências audíveis [20 Hz - 20 kHz]. A sua amplitude foi definida como 0,8, para se aproximar da amplitude máxima representável pela modulação sigma-delta ($\Delta = 1$).

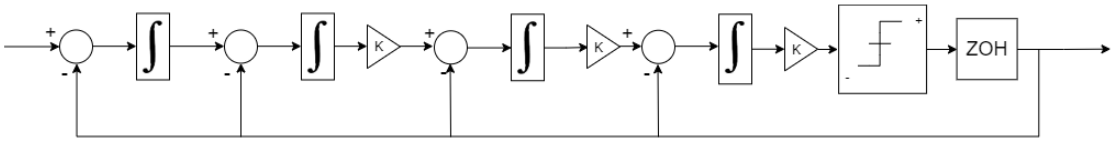
Este sinal de entrada simula um sinal de som na entrada do microfone, o qual é um sinal contínuo por natureza. No entanto, a simulação em MATLAB não permite a representação de sinais verdadeiramente contínuos. Portanto, a solução encontrada foi representar a senoide de entrada com uma frequência de amostragem dez vezes maior que a frequência de amostragem de saída do modulador sigma-delta. Uma vez que essa frequência de saída é 3,84 MHz, a frequência de amostragem da senoide foi de 38,4 MHz.

O modulador modulador sigma-delta simulado tem como objetivo gerar o sinal PDM para a simulação, o que no sistema implementado é feito pelo microfone. Portanto, ele simula o comportamento do microfone. Internamente, o microfone emprega um modulador sigma-delta de quarta ordem, o que significa que ele tem quatro integradores. No entanto, a topologia do modulador não é disponibilizada pelo fabricante do microfone. Por isso, foi necessário escolher uma topologia de modulador muito provavelmente diferente da topologia empregada no microfone.

A topologia escolhida para a simulação do modulador sigma-delta pode ser observada na Figura 25. Optou-se por essa topologia por sua simplicidade, uma vez que os integradores e subtratores estão dispostos de forma similar ao modulador sigma-delta de primeira ordem, exposto na Figura 6 no capítulo de revisão bibliográfica. Os blocos de ganho foram adicionados para garantir a estabilidade do filtro. A necessidade desse bloco se constatou durante a realização de testes com a simulação.

A topologia apresentada foi implementada em script MATLAB. Cada sinal de saída de um bloco subtrator, integrador ou de ganho corresponde a um vetor. Os blocos subtratores foram simulados por uma operação de subtração entre dois elementos dos vetores de

Figura 25: Topologia do modulador sigma-delta simulado.



Fonte: Autor.

entrada do bloco com o mesmo índice. Já o integrador foi simulado pela soma do vetor de entrada do bloco com o vetor de saída de índice anterior. O ganho apenas multiplica o vetor de entrada pelo índice K. O bloco de saturação foi simulado pela comparação do vetor de entrada com 0. Se o valor é maior, o vetor de saída adquire o valor Δ. Caso contrário, ele adquire o valor $-\Delta$. Foi definido $\Delta = 1$, uma vez que essa é a mesma transformação realizada sobre o sinal PDM no código VHDL. O bloco ZOH, por fim, seleciona os elementos do vetor de entrada de modo que o vetor de saída tenha o período de amostragem projetado. Uma vez que o sinal de entrada foi projetado com frequência de amostragem dez vezes maior que o sinal de saída do modulador, o bloco ZOH seleciona uma a cada dez amostras do seu vetor de entrada.

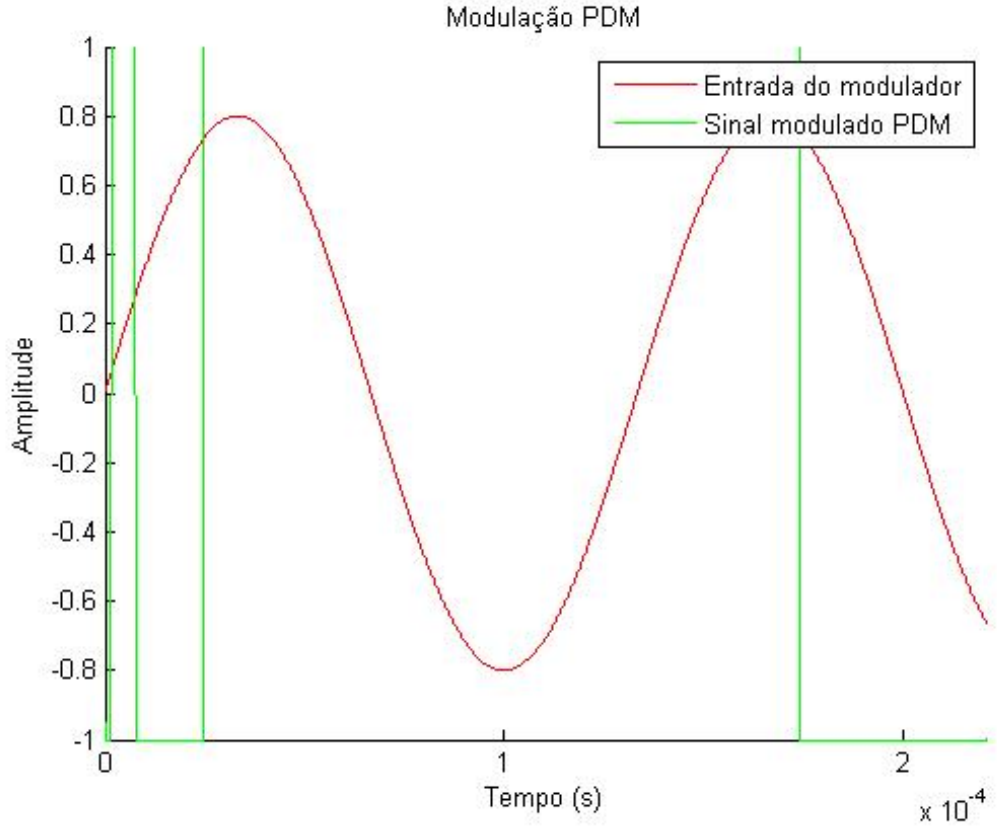
O ganho K foi em primeira instância definido como 1. No entanto, a saída do modulador não correspondia ao sinal PDM relativo à senoide de entrada, demonstrando que o modulador estava instável. Na Figura 26, observa-se a entrada e a saída do modulador sigma-delta com o ganho $K = 1$. Portanto, decidiu-se reduzir o ganho para $K = 0,9$. Observou-se então que o sinal PDM correspondia ao esperado. O resultado final da simulação do modulador sigma-delta é apresentado no capítulo de resultados.

Em seguida, foi simulado o filtro CIC, que recebe como entrada o sinal PDM gerado pelo modulador sigma-delta. Ele é composto por cinco integradores, cinco *combs* e um decimador. Os integradores foram simulados da mesma maneira que os integradores do modulador sigma-delta. No entanto, na saída dos integradores simulados, foi realizado *wrap-around* próprio de somas e subtrações com complemento de 2. A simulação do *wrap-around* consistiu em realizar comparações do valor de saída do integrador com os limites do intervalo representado por um número inteiro de n bits $[-(2^{n-1}); (2^{n-1} - 1)]$. Se o valor de saída está acima do intervalo, ele foi subtraído de 2^n. Já se o valor de saída encontra-se abaixo do intervalo, ele foi somado de 2^n. Dessa forma, o valor de saída encontra-se sempre no intervalo de representação por n bits. Conforme já exposto anteriormente, o sinal de saída dos integradores foi simulado para ser representado por $n = 24$ bits.

O bloco decimador foi simulado pela seleção de uma em cada 20 amostras do seu vetor de entrada. Desse modo, um vetor com 20 vezes menos amostras foi gerado. Já os blocos *comb* foram simulados através da subtração entre uma amostra do vetor de entrada e a amostra com indíce anterior. Além disso, a operação de *wrap-around* anteriormente explicada também foi aplicada na saída de cada bloco *comb*. Por fim, após o último bloco *comb*, foi feito o truncamento do sinal, que consistiu em uma divisão inteira de todas as amostras do vetor por 2^8. Essa divisão simula o truncamento de 8 bits, necessário por causa da passagem do sinal interno do filtro CIC de 24 bits para o sinal de entrada dos filtros *halfband* de 16 bits.

É importante salientar que, mesmo que os vetores dos sinais internos do filtro CIC sejam representados por números de ponto flutuante, os valores deles são sempre números inteiros. Isso acontece porque o sinal PDM de entrada do filtro CIC é um vetor de nú-

Figura 26: Sinal de saída do modulador sigma delta simulado com $k = 1$.



Fonte: Autor.

meros inteiros (todos os elementos são iguais a 1 ou -1) e as operações que são feitas nos integradores e blocos *comb* são somente de soma ou subtração. Não existe operação de divisão, que geraria números não inteiros. Portanto, uma vez que a mantissa dos números em ponto flutuante tenha um número maior de bits do que os limites do sinal projetado (24 bits), não há perdas na representação em ponto flutuante. Essa condição é atendida uma vez que os números do tipo *double* utilizados na simulação tem 64 bits.

Os filtros *halfband* foram simulados através de uma operação de convolução do vetor de entrada do filtro pelos seus coeficientes anteriormente calculados. Essa operação foi feita utilizando a função *conv* da ferramenta MATLAB. Já a decimação foi realizada com um bloco decimador análogo ao utilizado no filtro CIC, mas desta vez selecionando uma amostra a cada duas, gerando um vetor duas vezes menor. Uma vez que a topologia do transcodificador PDM-PCM contém dois filtros *halfband* decimadores, essas operações foram realizadas duas vezes no sinal.

Por fim, os diferentes vetores de sinal gerados na simulação foram desenhados em gráficos, utilizando-se para isso os seus respectivos vetores de tempo. Além disso, foi também calculada a *fft* de diversos sinais. Os gráficos destes sinais e suas respectivas *fft*'s são apresentados no capítulo de resultados.

3.3 Módulo de comunicação

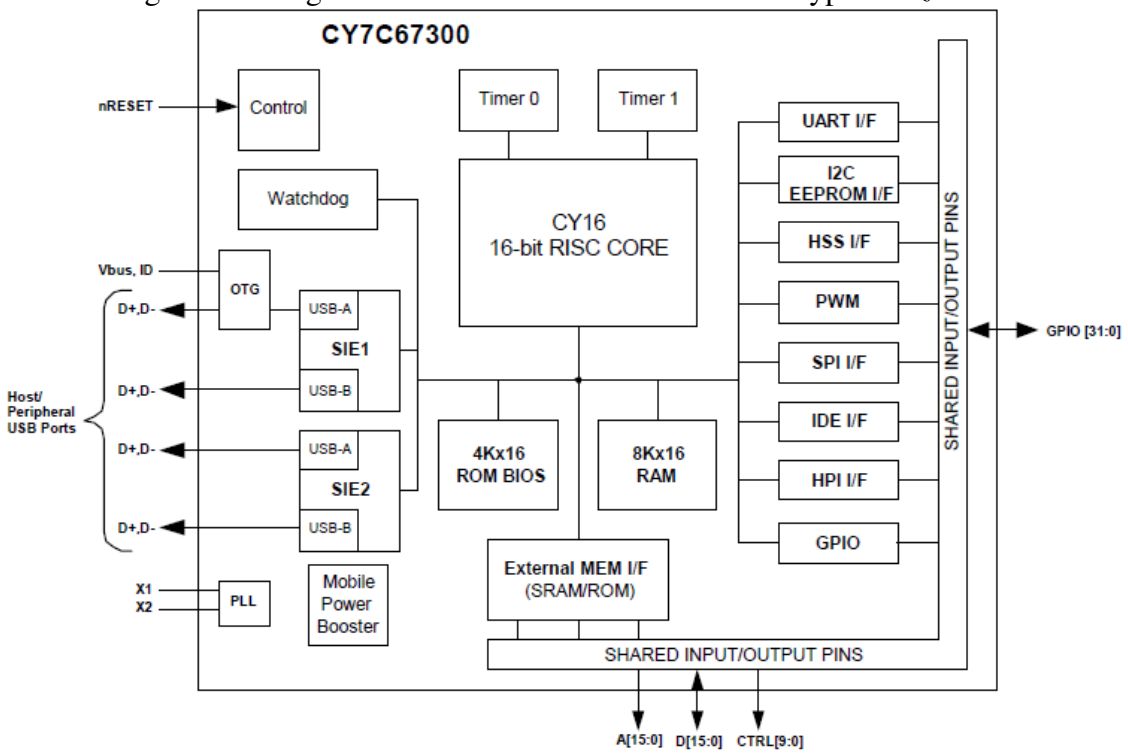
O módulo de comunicação é formado pelo componente de interface HPI, contido na FPGA e codificado em VHDL, e pelo controlador USB Cypress *Ez-Host*. Ambos têm o

objetivo de enviar os sinais PCM produzidos pelo módulo de transcodificação PDM-PCM através da interface USB. O protocolo USB é implementado pelo controlador Cypress *Ez-Host*. A nível de interface USB, ele se comporta como um dispositivo, o que significa que ele responde às requisições do *host*, o qual é mestre do barramento. Antes de apresentar a solução desenvolvida, será detalhado o funcionamento de controlador USB Cypress *Ez-Host*, bom como o protocolo HPI definido pelo controlador.

3.3.1 Controlador USB Cypress *Ez-Host*

O controlador USB Cypress *Ez-Host* implementa o protocolo USB, sendo possível configurá-lo para desempenhar a função de *host* ou dispositivo. Ele possui dois SIEs (*Serial Interface Engines*), de modo que cada um pode ser configurado como *host* ou dispositivo independentemente, possibilitando uma configuração dual. O sistema é controlado por um processador RISC de 16 bits, chamado CY16. O diagrama de blocos do controlador pode ser visto na Figura 27.

Figura 27: Diagrama de blocos do controlador USB Cypress *Ez-Host*.



Fonte: Cypress (2003)

O Cypress *Ez-Host* conta com dois modos de funcionamento: coprocessador e *standalone*. No primeiro, o seu processador interno atua como um escravo de outro processador externo, que envia instruções através do protocolo LCP (*Link Command Protocol*). Essas instruções podem ser enviadas através de suas interfaces HPI, HSS ou SPI.

Já no modo *standalone*, o Cypress *Ez-Host* é controlado somente pelo seu processador interno executando um *firmware* contido em sua memória. Esse *firmware* é primeiramente armazenado em uma memória EEPROM externa, que é lida durante a rotina de inicialização do processador. O modo de operação é definido pelos valor dos pinos de *bootstrap* durante a inicialização do controlador, conforme a Figura 28.

Figura 28: Modos de operação do Cypress *Ez-Host* definidos pelos pinos de *bootstrap*.

GPIO31 (Pin 39)	GPIO30 (Pin 40)	Boot Mode
0	0	Host Port Interface (HPI)
0	1	High Speed Serial (HSS)
1	0	Serial Peripheral Interface (SPI, slave mode)
1	1	I2C EEPROM (Standalone Mode)

Fonte: Cypress (2003)

A construção do *firmware* do Cypress *Ez-Host* utiliza um modelo de camadas. A primeira camada é formada pela BIOS, formado por código assembler para CY16, gravado de forma permanente na memória ROM do chip. O funcionamento da BIOS é baseado em tabelas de interrupções, que executa rotinas na ocorrência de eventos diversos.

A segunda camada é formada pelo *Frameworks*, o qual é constituído por diversos arquivos de código C fornecidos pela Cypress. Esse código é responsável por implementar funções que permitam o uso das funcionalidades de *hardware* do Cypress *Ez-Host* e interagir com a BIOS.

A terceira camada é formada pela aplicação, que é constituída pelo código desenvolvido pelo usuário para gerenciar a aplicação desejada. Esse código tem um conjunto de macros que configuram a camada de *Frameworks* para desempenhar as tarefas específicas da aplicação, assim como as funções que implementam as funcionalidades específicas da aplicação.

Essa divisão em camadas é representada na Figura 29. Observa-se que, assim como existe uma divisão em camadas, também há uma divisão em tipos de funções. A primeira, *Init_Task*, é executada uma vez somente durante a inicialização do dispositivo. *Callback_Tasks* são funções executadas quando ocorre uma interrupção, que pode ser gerada por eventos no protocolo USB ou no próprio *firmware*. *Idle_Task* são o conjunto de tarefas executadas quando os tipos anteriores não estão ativos. Essas funções obedecem a um mecanismo de encadeamento, em que uma camada chama a *Idle_Task* da camada posterior.

3.3.2 Protocolo HPI

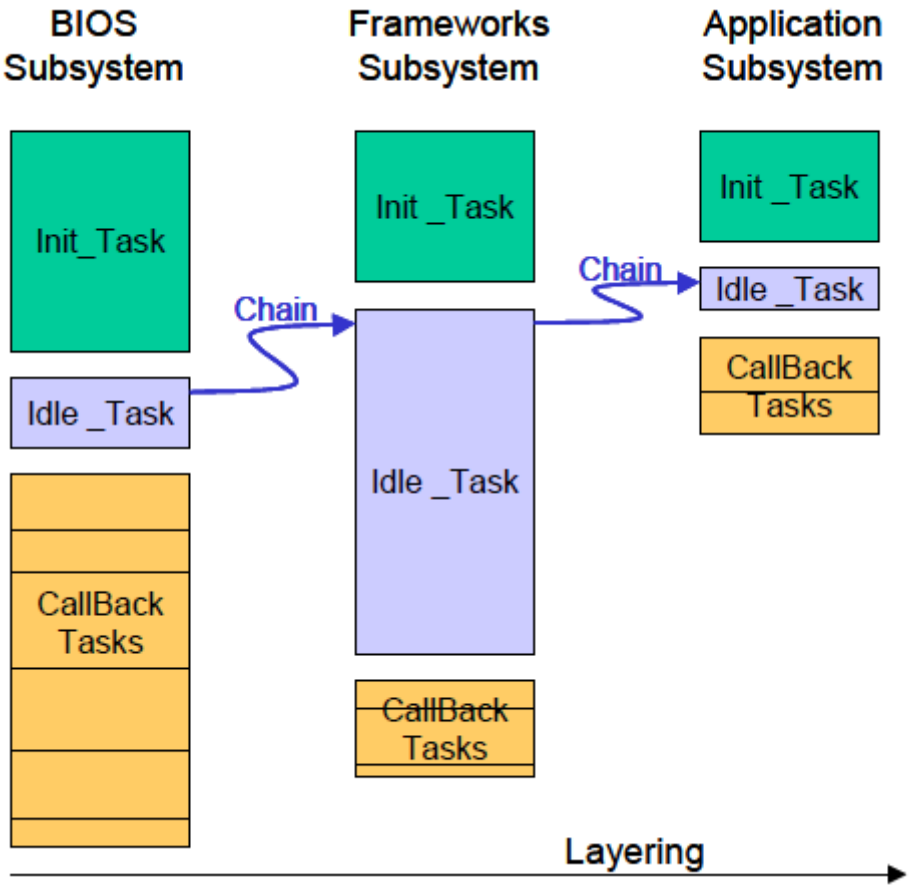
O protocolo HPI (*Host Port Interface*) é definido pelo controlador *Ez-Host* como uma interface de comunicação com um mestre externo, que permite acesso direto à memória RAM (DMA) e envio de instruções através do protocolo LCP. O dispositivo externo sempre atua como mestre. O protocolo define um barramento de dados de 16 bits e uma velocidade de comunicação de até 16 MB/s.

A comunicação com o controlador *Ez-Host* se dá por meio de 16 pinos bidirecionais de dados, além de outros seis pinos de controle [Swanbeck (2011)]. Um diagrama da conexão para interface HPI pode ser vista na Figura 30.

A função de cada um dos seis pinos de controle é descrita abaixo:

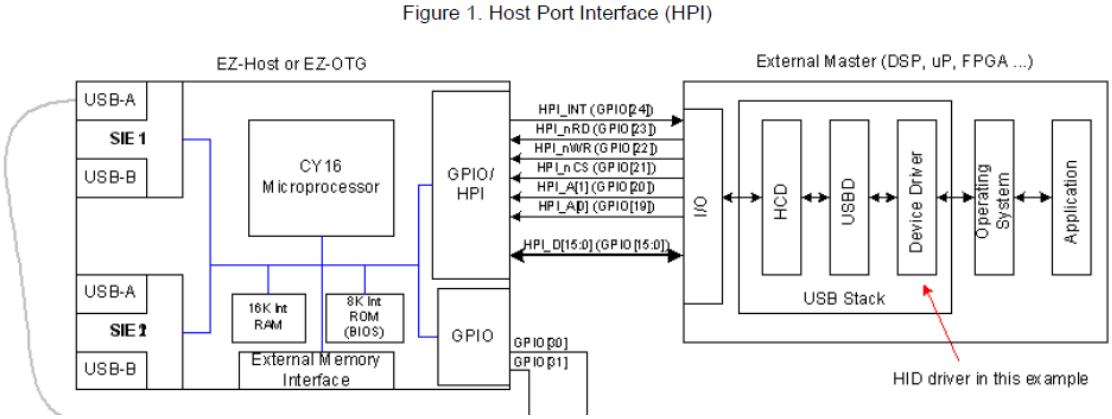
- HPI_A[0:1]: são os pinos de endereço do protocolo HPI, controlados pelo mestre externo ao *Ez-Host*. O seu valor define o registrador acessado pelo mestre durante uma transação. A tabela de registradores com seus respectivos endereços é dada na Figura 31. A coluna à direita indica se o acesso a esse registrador é somente escrita (W), somente leitura (R) ou escrita e leitura (WR).

Figura 29: Representação da arquitetura em camadas do *firmware* do Cypress *Ez-Host*.



Fonte: Hyde (2003)

Figura 30: Diagrama de blocos do Cypress *Ez-Host* com barramento HPI.



Fonte: Swanbeck (2011)

- HPI_nCS: é o sinal enviado pelo mestre para habilitar o controlador USB para operações HPI. Se esse pino não estiver em 0, nenhuma comunicação HPI é válida.
- HPI_nWR: é o sinal enviado pelo mestre para indicar uma operação de escrita. Para haver uma operação, é necessário que haja uma transição para 0.
- HPI_nRD: é o sinal enviado pelo mestre para indicar uma operação de leitura. Para haver uma operação, é necessário que haja uma transição para 0.
- HPI_INT: é o sinal de interrupção enviado pelo *Ez-Host* ao mestre externo. No caso de um evento programável de interrupção, o *Ez-Host* coloca esse pino em 1. A razão da interrupção é indicada no registrador HPI STATUS, o qual é descrito na Figura 32.

Figura 31: Registradores dedicados ao protocolo HPI.

Port Registers	HPI A [1]	HPI A [0]	Access
HPI DATA	0	0	RW
HPI MAILBOX	0	1	RW
HPI ADDRESS	1	0	W
HPI STATUS	1	1	R

Fonte: Swanbeck (2011)

O protocolo HPI também define quatro registradores, que podem ser selecionados através dos pinos HPI_A[1:0] e possuem as seguintes funções:

- HPI DATA: é o registrador onde são colocados os dados da memória interna durante transações DMA (*Direct Memory Access*) de leitura ou de escrita.

- HPI MAILBOX: é o registrador de comunicação bidirecional entre o mestre externo e o processador interno do *Ez-Host*. Ele é utilizado quando o *Ez-Host* é inicializado no modo coprocessador e é comandado pelo mestre externo por instruções enviadas para esse registrador.

- HPI ADDRESS: é o registrador onde é especificado o endereço de memória de leitura e escrita em transações DMA.

- HPI STATUS: é o registrador que utiliza uma estrutura do tipo *bit field* para indicar quais interrupções foram ativadas pelo Cypress *Ez-Host*. A tabela que define esse registrador é indicada na Figura 32.

Figura 32: Registrador HPI STATUS.

Bit16	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8
VBUS Flag	ID Flag	Reserved	SOF/EOP2 Flag	Reserved	SOF/EOP1 Flag	Reset2 Flag	Mailbox IN Flag
Bit7	**Bit6**	**Bit5**	**Bit4**	**Bit3**	**Bit2**	**Bit1**	**Bit0**
Resume2 Flag	Resume1 Flag	SIE2msg	SIE1msg	Done2 Flag	Done1 Flag	Reset1 Flag	Mailbox OUT Flag

Fonte: Swanbeck (2011)

O protocolo HPI também define a estrutura de uma transação DMA de escrita ou leitura de memória. Um diagrama de uma transação exemplo de escrita do dado 0x0074 no registrador 0x01C2 pode ser visto na Figura 33. Primeiramente, é indicado o endereço de interesse por uma operação de escrita no registrador HPI ADDRESS, no caso 0x01C2. Para isso, os pinos HPI_A[1:0] devem estar com o valor 10, os pinos HPI_nWR e HPI_nCS em 0 e o pino HPI_nRD em 1. No momento em que ocorre a borda de subida em HPI_nWR, o valor da porta HPI_D[15:0] é adquirido e escrito em HPI ADDRESS.

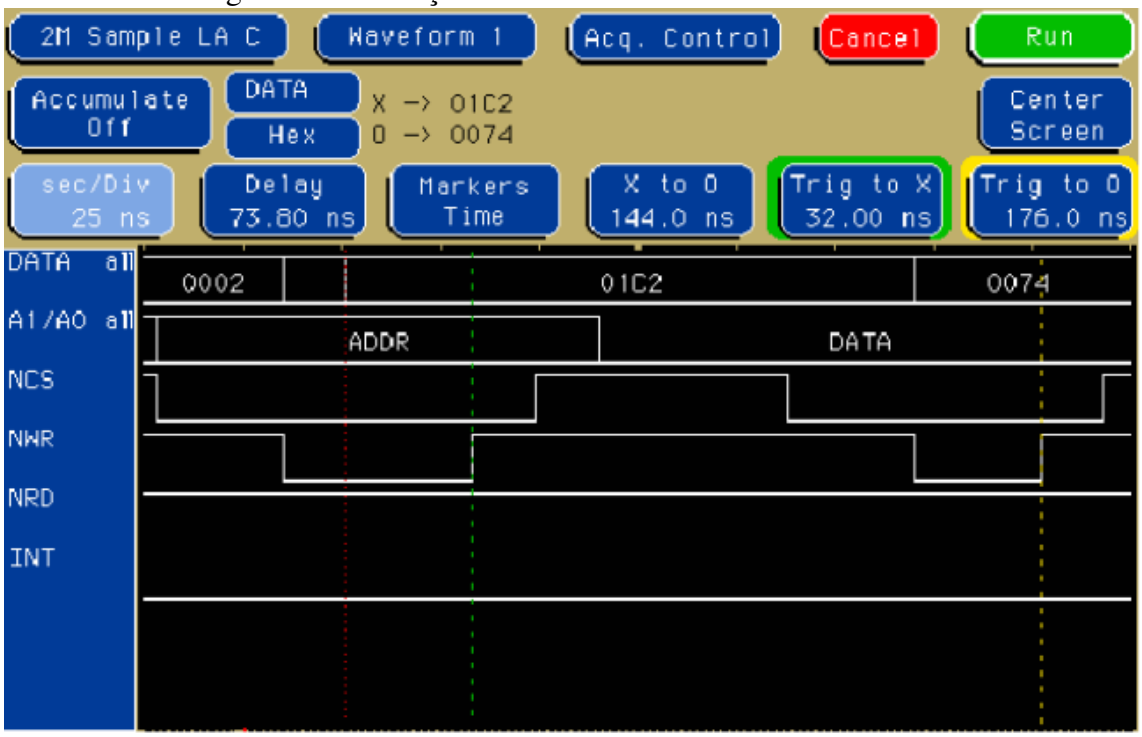
Em seguida, deve ser acessado o registrador HPI DATA. Em uma operação de escrita, é nesse registrador que o mestre externo deve enviar o dado que será escrito no endereço indicado anteriormente. No exemplo, este valor é 0x0074. Para isso, os pinos HPI_A[1:0] devem estar com o valor 00, os pinos HPI_nWR e HPI_nCS em 0 e o pino HPI_nRD em 1. Em uma operação de leitura, são trocados os valores de HPI_nWR e HPI_nRD. Em uma operação de escrita, como no exemplo, quando ocorre a borda de subida em HPI_nWR, o valor de HPI_D[15:0] é escrito no endereço indicado indicado no registrador HPI ADDRESS. Já em uma operação de leitura, o *Ez-Host* disponibiliza o valor da memória no momento em que ocorre uma borda de descida em HPI_nRD.

Se o mestre externo realiza operações de escrita ou leitura em sequência sem escrever novamente em HPI ADDRESS, o seu valor é automaticamente incrementado, de forma que a operação se prolonga em uma seção contínua na memória.

3.3.3 Projeto dos descritores USB

Tendo sido descritos o controlador utilizado para comunicação USB, esta seção começa a descrição do módulo de comunicação desenvolvido. Uma das funções desempenhadas pelo controlador USB é responder às requisições do *host* USB durante o processo de enumeração. Essas requisições são efetuadas pelos diferentes *drivers* USB atuando no sistema operacional do *host*. Para diferentes tipos de dispositivos USB, as requisições

Figura 33: Transação HPI de escrita de dado em memória.



Fonte: Swanbeck (2011)

enviadas durante o processo de enumeração podem variar, uma vez que são diferentes *drivers* que as realizam. No entanto, uma requisição que sempre está presente nesse processo é a *GetDescriptor* [Axelson (2009)]. Ao enviar esse requisição, o *host* USB espera receber os descritores do dispositivo.

No dispositivo desenvolvido no projeto, os descritores do dispositivo foram configurados para que o *host* USB o reconheça como um dispositivo de áudio de seis canais, cada canal com formato PCM e com frequência de amostragem de 48 kHz. Uma vez que o protocolo USB define uma estrutura hierárquica complexa para os descritores, um estudo preliminar foi realizado para que essas informações pudessem ser indicadas de forma correta de acordo com o protocolo.

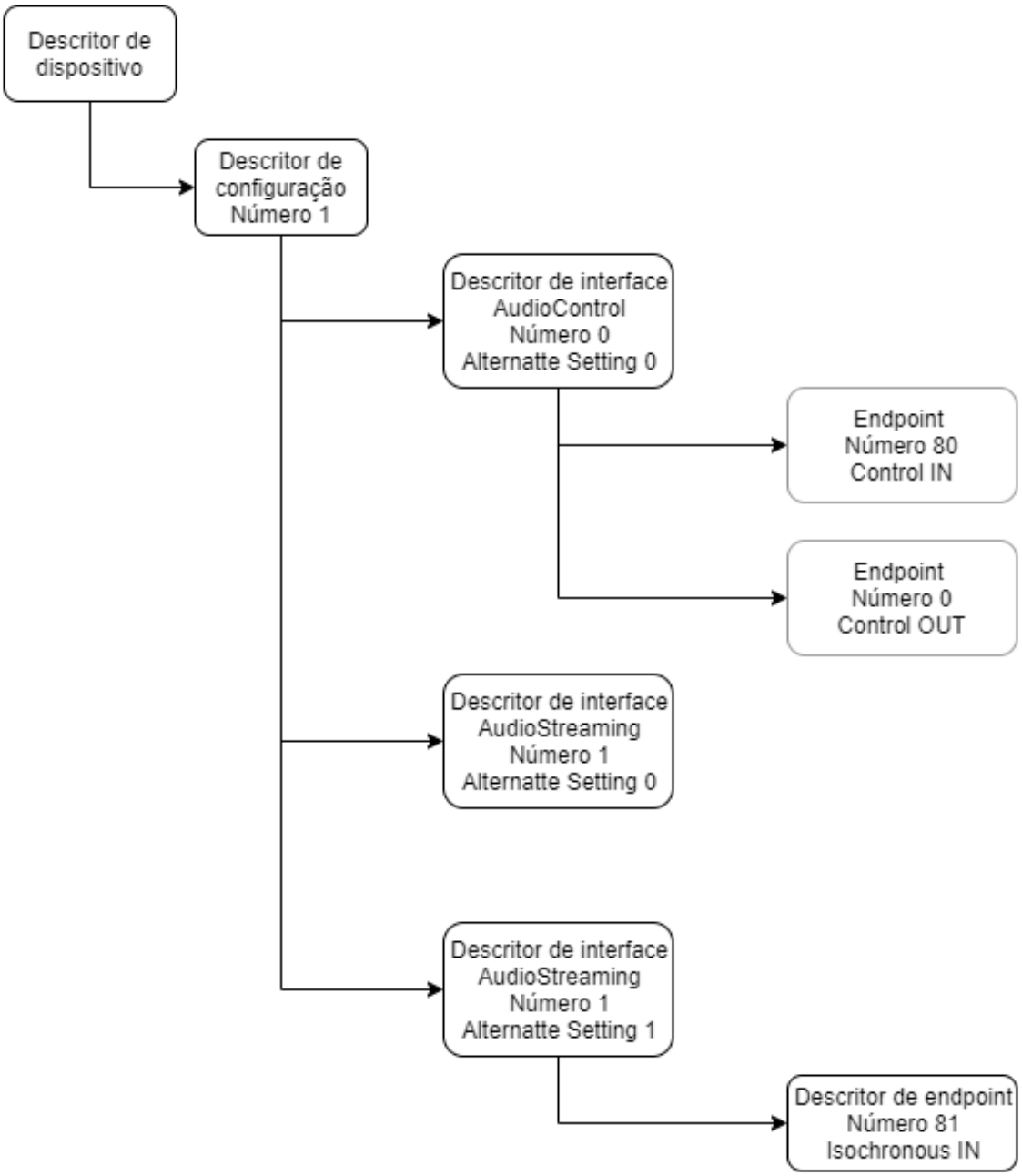
A organização hierárquica dos descritores consiste em quatro níveis: dispositivo, configuração, interface e *endpoint*. O dispositivo do presente projeto foi organizado conforme a Figura 34.

O dispositivo foi projetado com apenas um descritor de configuração pois não era necessário descrever conjuntos distintos de propriedades que pudessem ser selecionados. O dispositivo também apresenta duas interfaces, sendo uma do tipo *AudioControl* e uma do tipo *AudioStreaming*. A segunda interface tem duas *alternate settings* diferentes, e apenas uma delas pode ser selecionada por vez.

A interface *AudioControl* é necessária para a função áudio e é por onde são acessados os controles da função [Ashour et al. (1998b)]. Como pode ser visto no diagrama da Figura 34, ela contém dois *endpoints* do tipo *control*. No entanto, esses correspondem aos *endpoints* de endereço zero que estão presentes em todos os dispositivos USB, e portanto não necessitam de descritores [Ashour et al. (1998b)].

A interface *AudioStreaming* possui os *endpoints* por onde passam os sinais de áudio. Cada interface *AudioStreaming* suporta uma *stream* de áudio. O dispositivo do projeto,

Figura 34: Organização hierárquica dos descritores USB do dispositivo.



Fonte: Autor.

embora seja projetado para enviar seis canais, pode utilizar apenas uma *stream*, uma vez que os canais enviam sinais de áudio sincronizados [Ashour et al. (1998b)]. Logo, essa interface possui apenas um *endpoint* do tipo IN, o que significa que a *stream* de áudio é uma entrada do ponto de vista do *host* USB.

Além disso, foi necessário criar essa interface com duas *alternate settings*. A especificação USB impõe que, para uma interface com *endpoint isochronous*, haja também uma *alternate setting* sem o *endpoint*. Essa imposição é necessária para que a enumeração seja realizada com sucesso mesmo que o *host* USB não seja capaz de alocar a largura de banda necessária para aquele *endpoint* [Silicon (2006)].

Os descritores elaborados foram programados no *firmware* do controlador USB em linguagem C. Para a elaboração do *firmware*, foi utilizado como base um *firmware* exemplo disponibilizado pela Cypress. Nesse *firmware* exemplo, já havia os descritores de um dispositivo, com diferentes funcionalidades daquelas previstas para o presente projeto. Durante a elaboração dos descritores desse projeto, alguns descritores tiveram de ser criados sem nenhuma base, enquanto que outros foram resultado da modificação de valores dos descritores utilizados no *firmware* exemplo utilizado como base.

A seguir, serão apresentados os valores dos descritores de dispositivo, configuração, interface e *endpoint* elaborados em tabelas. Cada tabela terá os diferentes campos dos descritores, constando o nome do campo, o que ele representa e o valor escolhido neste projeto para o campo em forma de número hexadecimal. Os campos que já estavam presentes no *firmware* exemplo e tiveram os seus valores mantidos serão marcados em negrito nas tabelas. A estrutura de todos os descritores são fornecidas no conjunto de documentos que especificam o protocolo USB.

Após apresentar os valores estabelecidos para os descritores, será explicado com mais detalhe como o *firmware* foi implementado. Isso inclui como o ambiente de programação foi configurado, bem como a estrutura do *firmware* e como ele interage com a BIOS do controlador USB.

Além dos descritores que serão apresentados nas tabelas, foi também elaborado o descritor *string*, o qual define nomes para as características desejadas do dispositivo. Esse descritor pode conter diversas *strings*, cada uma indicada por um índice maior que zero. Os outros descritores possuem campos que se referem ao descritor *string* por um índice, o qual indica uma das *strings* do descritor. O descritor *string* foi elaborado com três *strings*. O valor dessas *strings* será apresentado quando o seu índice for indicado pelos outros descritores.

O primeiro descritor a ser elaborado foi o descritor de dispositivo, o qual descreve informações gerais sobre o dispositivo [Compaq et al. (2000)]. Esse descritor contém quatorze campos que são apresentados na Tabela 5. Como pode se observar, todos os valores desse descritor foram mantidos do *firmware* exemplo, pois estão marcados em negrito. Uma vez que o descritor de dispositivo apresenta informações genéricas, não houve necessidade de mudança dos valores.

O campo *bLength* se refere ao tamanho do descritor e é constante para descritores de dispositivo. Ele foi obtido simplesmente a partir da contagem de número de bytes presentes no descritor e foi traduzido para o seu valor em hexadecimal. Este campo está presente em todos os descritores do projeto, e nos próximos descritores que serão apresentados ele não será mais explicado, uma vez que o seu valor é sempre obtido a partir da contagem dos bytes.

O campo *bDescriptorType* indica o tipo do descritor. O valor 0x01 equivale à constante DEVICE e foi obtido a partir da Tabela 6. Esse campo indica o tipo do descritor, no caso um descritor de dispositivo. Assim como *bLength*, *bDescriptorType* também está presente em todos os descritores. Nos outros descritores, o valor de *bDescriptorType* é igualmente proveniente da Tabela 6.

O campo *bcdUSB* indica a versão da especificação USB utilizada como base para a configuração do dispositivo. O valor escolhido indica a especificação 2.0. O campo *bDeviceClass* indica a classe do dispositivo. No entanto, a classe também pode ser definida a nível de interface, o que é obrigatório para dispositivos de áudio [Ashour et al. (1998b)]. O mesmo acontece para o campo *bDeviceSubClass*, e para *bDeviceProtocol*, que é uma consequência dos dois campos anteriores. Por isso, os três valores foram estabelecidos

Tabela 5: Descritor de dispositivo utilizado no dispositivo do projeto.

Campo	Descrição	Valor
bLength	Tamanho do descritor em bytes	0x12
bDescriptorType	Tipo do descritor	0x01
bcdUSB	Número da especificação USB utilizada	0x0200
bDeviceClass	Classe do dispositivo	0x00
bDeviceSubClass	Subclasse do dispositivo	0x00
bDeviceProtocol	Código de protocolo	0x00
bMaxPacketSize0	Máximo tamanho de pacote para o *endpoint* zero	0x40
idVendor	Vendor ID	0x4242
idProduct	Product ID	0xC003
bcdDevice	Número da versão de lançamento do dispositivo	0x0100
iManufacturer	Índice do nome do fabricante no descritor *string*	0x01
iProduct	Índice do nome do produto no descritor *string*	0x02
iSerialNumber	Índice do número de série no descritor *string*	0x00
bNumConfigurations	Número de configurações possíveis do dispositivo	0x01

Fonte: Autor.

Tabela 6: Constantes dos tipos de descritores

Descriptor Types	Value
DEVICE	1
CONFIGURATION	2
STRING	3
INTERFACE	4
ENDPOINT	5
DEVICE_QUALIFIER	6
OTHER_SPEED_CONFIGURATION	7
INTERFACE_POWER	8

Fonte: Compaq et al. (2000)

como 0x00, o que significa que são definidos a nível de interface [Compaq et al. (2000)].

O campo *bMaxPacketSize0* define o tamanho máximo de pacote em bytes para o *endpoint* do tipo *control* de número 0. Esse *endpoint* é padrão para todos os dispositivos USB, e o valor desse campo pode ser definido como 8, 16, 32 ou 64 [Compaq et al. (2000)]. O valor 64 (0x40 em hexadecimal) foi mantido do *firmware* exemplo utilizado como base.

O campo *idVendor* é atribuído pelo USB-IF, organização que reúne os ID's dos fabricantes de dispositivos USB. Uma vez que o valor 0x4242 já estava no *firmware* exemplo ele foi mantido. Já o campo *idProduct* é definido pelo fabricante, e foi igualmente mantido do *firmware* exemplo.

O campo *bcdDevice* define a versão do dispositivo, a qual foi estabelecida como 1.0. O campo *iManufacturer* recebeu o valor 0x01, o que indica que o nome do fabricante é definido pela primeira *string* do descritor *string*. A *string* contida nessa posição é "Cypress Semiconductor", o qual foi mantido do *firmware* exemplo. O campo *iProduct* indica o nome do produto, o qual está presente no descritor *string* com índice 2. Nesse índice está contida a *string* "PD2 UFRGS Davi". Já o campo *iSerialNumber* recebeu o valor

0x00, o que significa que não existe *string* para o número de série do dispositivo. Por fim, o campo *bNumConfigurations* recebeu o valor 1, o que significa que o dispositivo possui apenas um descritor de configuração.

Tabela 7: Descritor de configuração utilizado no dispositivo do projeto.

Campo	Descrição	Valor
bLength	Tamanho do descritor em bytes	0x09
bDescriptorType	Tipo do descritor	0x02
wTotalLength	Tamanho dos descritores da configuração em bytes	0x0079
bNumInterfaces	Número de interfaces para esta configuração	0x02
bConfigurationValue	Número desta configuração	0x01
iConfiguration	Índice da descrição no descritor *string*	0x00
bmAttributes	*Bitmap* com atributos desta configuração	0xC0
bMaxPower	Máximo consumo de corrente do dispositivo	0x01

Fonte: Autor.

Na Tabela 7, observa-se os valores atribuídos ao descritor de configuração. O campo *bDescriptorType* recebe o valor da constante CONFIGURATION, conforme a Tabela 6. O campo *wTotalLength* recebe o valor de tamanho total em bytes de todos os descritores relacionados àquela configuração, incluíndo descritores de interface e *endpoint*. O campo *bNumInterfaces* mostra que a configuração possui duas interfaces, conforme a Figura 34. O campo *bConfigurationValue* recebe o valor 0x01, uma vez que configurações não podem ter número 0 [Compaq et al. (2000)].

O campo *bmAttributes* é um *bitmap* que define propriedades da configuração. O bit de índice seis define se o dispositivo tem a própria alimentação [Compaq et al. (2000)] e foi definido como 1, uma vez que a FPGA onde está o controlador USB tem alimentação própria e não é alimentada via USB. O bit de índice cinco determina se o dispositivo suporta *remote wakeup* [Compaq et al. (2000)] e foi definido como 0. O bit de índice sete deve ser definido como 1 e o restante como 0 [Compaq et al. (2000)], o que gera o valor final de 11000000 em binário ou 0xC0 em hexadecimal.

O campo *bMaxPower* define a consumo máximo de corrente do barramento USB em múltiplos de 2 mA [Compaq et al. (2000)]. O dispositivo não precisa de corrente do barramento e o valor do campo foi deixado como no *firmware* exemplo.

A Tabela 15 mostra o descritor padrão da interface *AudioControl*. O campo *bDescriptorType* foi definido como a constante INTERFACE, conforme a Tabela 6. O campo *bInterfaceNumber* recebeu valor 0x00 uma vez que esta é a primeira interface do dispositivo. O campo *bAlternateSetting*, uma vez que esta interface não necessita de configurações alternativas, recebeu valor zero.

O campo *bNumEndpoints* recebeu o valor zero uma vez que essa interface não possui descritores de *endpoints*. Mesmo que a interface *AudioControl* apresente um *endpoint* do tipo *control* OUT e outro *control* IN, ambos são os *endpoints* padrões e não precisam de descritores. O campo *bInterfaceClass* recebe o valor 0x01 por ser correspondente à constante AUDIO [Ashour et al. (1998b)], que é a classe da interface. Já o campo *bInterfaceSubClass* recebe o valor da constante AUDIO_CONTROL, conforme a Tabela 9. O campo *bInterfaceProtocol* não é utilizado e é definido como zero, conforme estabelecido pela especificação da classe áudio USB [Ashour et al. (1998b)]. Por último, o campo *iInterface* indica que a descrição da interface está no descritor *string* no índice 3, onde

Tabela 8: Descritor padrão da interface *AudioControl* utilizado no dispositivo do projeto.

Campo	Descrição	Valor
bLength	Tamanho do descritor em bytes	0x09
bDescriptorType	Tipo do descritor	0x04
bInterfaceNumber	Número da interface	0x00
bAlternateSetting	Número de *alternate setting*	0x00
bNumEndpoints	Número de *endpoints* da interface	0x00
bInterfaceClass	Classe da interface	0x01
bInterfaceSubClass	Subclasse da interface	0x01
bInterfaceProtocol	Protocolo da interface	0x00
iInterface	Índice de descrição da interface no descritor *string*	0x03

Fonte: Autor.

está a *string* "67300DE030100", que foi mantida do *firmware* exemplo.

Tabela 9: Constantes do tipo *Audio Interface Subclass*.

Audio Subclass Code	Value
SUBCLASS_UNDEFINED	0x00
AUDIOCONTROL	0x01
AUDIOSTREAMING	0x02
MIDISTREAMING	0x03

Fonte: Ashour et al. (1998b)

Tabela 10: Descritor específico de classe da interface *AudioControl* utilizado no dispositivo do projeto.

Campo	Descrição	Valor
bLength	Tamanho do descritor em bytes	0x09
bDescriptorType	Tipo do descritor	0x24
bDescriptorSubType	Subtipo do descritor	0x01
bcdADC	Especificação da classe de áudio USB utilizada	0x0100
wTotalLength	Tamanho total de descritor específico de classe	0x0033
binCollection	Número de interfaces AS e MS da coleção	0x01
baInterfaceNr(1)	Número da primeira interface AudioStreaming	0x01

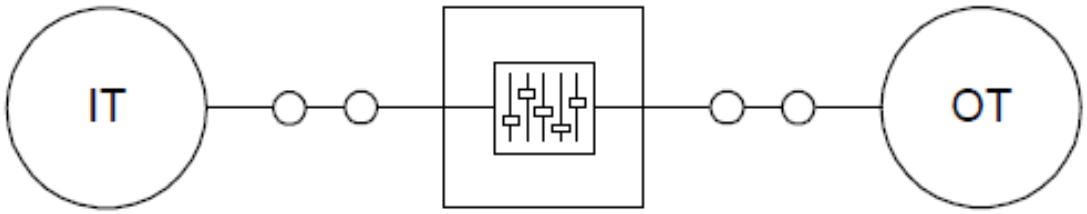
Fonte: Autor.

A Tabela 10 mostra o descritor específico de classe da interface *AudioControl*. Todos os descritores a partir deste não estavam presentes no *firmware* exemplo, e portanto foram totalmente elaborados durante o projeto.

A interface *AudioControl* engloba também todos os descritores dos terminais e *units* da função áudio. Portanto, antes da elaboração do descritor foi necessário definir a arquitetura de terminais e *units* que seria utilizada para o dispositivo. Primeiramente, foi definido que o dispositivo teria um terminal de entrada, que corresponde ao ponto onde o sinal de áudio entra no dispositivo, representando a matriz de microfones. Além disso, é necessário que ele possua um terminal de saída, que representa o fluxo de áudio enviado

ao *host* via USB. Entre estes terminais, foi colocada uma *feature unit*, que dá ao *host* a capacidade de controlar o sinal de áudio que passa por ela. Logo, a conexão entre os terminais e *units* pode ser visto na Figura 35. Essa arquitetura já havia sido utilizada para outros dispositivos de áudio em projetos anteriores [Silicon (2006)].

Figura 35: Representação gráfica da topologia da função áudio do dispositivo USB.



Fonte: Autor.

A partir dessa representação, é possível descrevê-la nos descritores específicos de classe da interface *AudioControl*. O campo *bDescriptorType* recebeu o valor da constante CS_INTERFACE, conforme indicado nas especificações da classe áudio USB [Ashour et al. (1998b)]. O campo *bDescriptorSubType* recebeu o valor da constante HEADER, conforme a Tabela 11. Isso indica que esse descritor é o cabeçalho do descritor do tipo *AudioControl*.

Tabela 11: Constantes do tipo "*Audio Class-Specific AC Interface Descriptor Subtypes*".

Descriptor Subtype	Value
AC_DESCRIPTOR_UNDEFINED	0x00
HEADER	0x01
INPUT_TERMINAL	0x02
OUTPUT_TERMINAL	0x03
MIXER_UNIT	0x04
SELECTOR_UNIT	0x05
FEATURE_UNIT	0x06
PROCESSING_UNIT	0x07
EXTENSION_UNIT	0x08

Fonte: Ashour et al. (1998b)

O valor 0x0100 do campo *bcdADC* indica que foi utilizada a especificação da classe áudio USB versão 1.0. O campo *wTotalLength* foi preenchido com o comprimento total do descritor específico de classe da interface *AudioControl*, considerando os descritores dos terminais e *units*. O campo *binCollection* indica o número de interfaces *AudioStreaming* e *MIDIStreaming* relacionadas com a interface *AudioControl*, e recebeu o valor 0x01 referente à única interface *AudioStreaming* do dispositivo. Finalmente, o campo *baInterfaceNr(1)* indica o número atribuído para esta interface *AudioStreaming*.

A Tabela 12 mostra o descritor do terminal de entrada do dispositivo. O campo *bDescriptorType* recebe o valor da constante CS_INTERFACE, assim como o descritor anteriormente apresentado. Todos os descritores de terminais e *units* do dispositivo apresentam o mesmo valor para esse campo. O campo *bDescriptorSubType* recebeu o valor da constante INPUT_TERMINAL, conforme a Tabela 11. Ao campo *bTerminalID* foi atribuído o valor 0x01, a identificação do terminal.

Tabela 12: Descritor do terminal de entrada utilizado no dispositivo do projeto.

Campo	Descrição	Valor
bLength	Tamanho do descritor em bytes	0x0C
bDescriptorType	Tipo do descritor	0x24
bDescriptorSubType	Subtipo do descritor	0x02
bTerminalID	Constante identificando o terminal	0x01
wTerminalType	Tipo de terminal	0x0205
bAssocTerminal	Terminal de saída associado	0x00
bNrChannels	Número de canais de áudio do terminal	0x06
wChannelConfig	Descrição da posição espacial de cada canal	0x0000
iChannelNames	Índice da descrição dos canais no descritor *string*	0x00
iTerminal	Índice da descrição do terminal no descritor *string*	0x00

Fonte: Autor.

O campo *wTerminalType* recebeu o valor 0x0205, que identifica o terminal como uma matriz de microfones, conforme [Ashour et al. (1998c)]. O campo *bAssocTerminal* é utilizado para casos em que o terminal de entrada está associado a um terminal de saída, por exemplo quando o microfone está anexado a um *headset* que é um terminal de saída [Silicon (2006)]. No entanto, isso não se aplica ao dispositivo deste projeto, e portanto o campo recebe o valor 0x00. O valor 0x06 do campo *bNrChannels* é o número de canais utilizados no projeto.

O campo *wChannelConfig* é um *bitmap* que descreve a posição espacial de cada canal. Cada um dos seus 12 bits LSB corresponde a uma posição espacial predefinida [Ashour et al. (1998b)]. No entanto, uma vez que a posição dos microfones da matriz utilizada no projeto não se encaixa nestas posições predefinidas, escolheu-se não fornecer informações da sua posição espacial nos descritores. O campo *iChannelNames* possibilita ao dispositivo enviar uma descrição da posição espacial dos canais que não utilizaram *wChannelConfig*, através do descritor *string*. No entanto, escolheu-se não utilizar essa característica tampouco. O último campo deste descritor, *iTerminal*, indica a posição da descrição do terminal de entrada no descritor *string*, funcionalidade que também não foi utilizada. A partir deste descritor, todos os outros campos nos outros descritores que indicam posições no descritor *string* foram definidos como 0.

Na Tabela 13, observa-se o descritor da *feature unit*. O campo *bDescriptorSubType* tem o valor da constante FEATURE_UNIT, conforme a Tabela 11. O campo *bUnitID* identifica a *unit* com o valor 0x02. O campo *bSourceID* indica que a entrada da *feature unit* é conectada no terminal de entrada, o qual tem identificação 0x01. O campo *bControlSize* indica o número de bytes dos elementos da lista *bmaControls*. Uma vez que cada elemento é um *bitmap* que deve representar 10 controles, são necessários 2 bytes para representá-los completamente.

Os elementos *bmaControls(n)* indicam, através de um *bitmap*, os controles que essa *feature unit* pode executar sobre cada um dos canais de áudio ou sobre todos eles ao mesmo tempo. Cada bit corresponde a um controle diferente, e os controles que recebem o valor 1 são interpretados como habilitados para aquele canal. Os controles correspondentes a cada bit são apresentados na lista a seguir [Ashour et al. (1998b)]:

- Bit 0: *Mute*
- Bit 1: Volume

Tabela 13: Descritor de *feature unit* utilizado no dispositivo do projeto.

Campo	Descrição	Valor
bLength	Tamanho do descritor em bytes	0x15
bDescriptorType	Tipo do descritor	0x24
bDescriptorSubType	Subtipo do descritor	0x06
bUnitID	Constante identificando a *unit*	0x02
bSourceID	ID da *unit* ou terminal ao qual essa *unit* está conectada	0x01
bControlSize	Tamanho em bytes pot elemento da lista bmaControls	0x02
bmaControls(0)	*Bitmap* dos controles suportados no canal master	0x0001
bmaControls(1)	*Bitmap* dos controles suportados no canal 1	0x0000
bmaControls(2)	*Bitmap* dos controles suportados no canal 2	0x0000
bmaControls(3)	*Bitmap* dos controles suportados no canal 3	0x0000
bmaControls(4)	*Bitmap* dos controles suportados no canal 4	0x0000
bmaControls(5)	*Bitmap* dos controles suportados no canal 5	0x0000
bmaControls(6)	*Bitmap* dos controles suportados no canal 6	0x0000
iFeature	Índice da descrição desta *unit* no descritor *string*	0x0000

Fonte: Autor.

- Bit 2: Controle de tom - Baixos
- Bit 3: Controle de tom - Médios
- Bit 4: Controle de tom - Agudos
- Bit 5: Equalizador gráfico
- Bit 6: Ganho automático
- Bit 7: *Delay*
- Bit 8: *Bass boost*
- Bit 9: *Loudness*

O campo *bmaControls(0)* se refere aos controles implementados no canal master, ou seja, que afeta todos os canais ao mesmo tempo. O valor 0x0001 indica que a *feature unit* tem apenas o controle de *mute* neste canal. Já os outros elementos da lista estão todos com valor 0x0000, o que indica que não existe nenhum controle possível para um canal separado dos demais. Esses valores foram igualmente utilizados no dispositivo USB de áudio de [Silicon (2006)].

A Tabela 14 indica os valores do descritor do terminal de saída. O campo *bDescriptorSubType* recebe o valor da constante OUTPUT_TERMINAL, conforme a Tabela 11. O campo *bTerminalID* identifica o terminal com o valor 0x03. O campo *wTerminalType* recebe o valor 0x0101 que indica que o terminal é do tipo USB *streaming*, o que significa que a saída do terminal é um *endpoint* USB [Ashour et al. (1998c)]. O campo *bAssocTerminal* tem valor 0x00, o que indica que não existe terminal associado a este. Finalmente, ao campo *bSourceID* foi atribuído o valor 0x02, o que indica que a entrada deste terminal é a *feature unit*.

Tabela 14: Descritor do terminal de saída utilizado no dispositivo do projeto.

Campo	Descrição	Valor
bLength	Tamanho do descritor em bytes	0x09
bDescriptorType	Tipo do descritor	0x24
bDescriptorSubType	Subtipo do descritor	0x03
bTerminalID	Constante identificando o terminal	0x03
wTerminalType	Tipo de terminal	0x0101
bAssocTerminal	Terminal de entrada associado	0x00
bSourceID	ID da *unit* ou terminal ao qual essa *unit* é conectada	0x02
iTerminal	Índice da descrição do terminal no descritor *string*	0x00

Fonte: Autor.

Tabela 15: Descritor padrão da interface *AudioStreaming* de *alternate setting* número zero utilizado no dispositivo do projeto.

Campo	Descrição	Valor
bLength	Tamanho do descritor em bytes	0x09
bDescriptorType	Tipo do descritor	0x04
bInterfaceNumber	Número da interface	0x01
bAlternateSetting	Número da *alternate setting*	0x00
bNumEndpoints	Número de *endpoints* da interface	0x00
bInterfaceClass	Classe da interface	0x01
bInterfaceSubClass	Subclasse da interface	0x02
bInterfaceProtocol	Protocolo da interface	0x00
iInterface	Índice de descrição da interface no descritor *string*	0x00

Fonte: Autor.

A Tabela 15 apresenta o descritor padrão da interface *AudioStreaming*. O campo *bDescriptorType* recebeu o valor da constante INTERFACE, conforme a Tabela 6. O campo *bInterfaceNumber* indica que o número da interface é 0x01, conforme o esquema da Figura 34. O valor atribuído ao campo *bAlternateSetting* indica que esta é a *alternate setting* número zero. Conforme a Figura 34, este descritor não tem nenhum *endpoint* associado, conforme as especificações USB da classe áudio [Silicon (2006)]. Logo, o campo *bNumEndpoints* recebe o valor 0x00.

O campo *bInterfaceClass* recebe a constante AUDIO [Ashour et al. (1998b)], enquanto que *bInterfaceSubClass* tem o valor da constante AUDIO_STREAMING, conforme a Tabela 9. O valor 0x00 é atribuído ao campo *bInterfaceProtocol*, uma vez que ele não é utilizado em interfaces de classe áudio [Ashour et al. (1998b)].

A Tabela 16 mostra o descritor padrão da interface *AudioStreaming* de *alternate setting* número 1. Este descritor é muito similar ao apresentado anteriormente. A diferença entre eles é que o segundo representa um *alternate setting* com um *endpoint*, ao passo que o primeiro não possui *endpoint*. Por isso, os únicos campos que trocam de valor entre eles são *bAlternateSetting* e *bNumEndpoints*.

A Tabela 17 mostra os valores utilizados no descritor específico de classe da interface *AudioStreaming*, o qual se refere à interface de *alternate setting* número 1. O campo *bDescriptorType* recebe o valor da constante CS_INTERFACE [Ashour et al. (1998b)].

Tabela 16: Descritor padrão da interface *AudioStreaming* de *alternate setting* número um utilizado no dispositivo do projeto.

Campo	Descrição	Valor
bLength	Tamanho do descritor em bytes	0x09
bDescriptorType	Tipo do descritor	0x04
bInterfaceNumber	Número da interface	0x01
bAlternateSetting	Número da *alternate setting*	0x01
bNumEndpoints	Número de *endpoints* da interface	0x01
bInterfaceClass	Classe da interface	0x01
bInterfaceSubClass	Subclasse da interface	0x02
bInterfaceProtocol	Protocolo da interface	0x00
iInterface	Índice de descrição da interface no descritor *string*	0x00

Fonte: Autor.

Tabela 17: Descritor específico de classe da interface *AudioStreaming* utilizado no dispositivo do projeto.

Campo	Descrição	Valor
bLength	Tamanho do descritor em bytes	0x07
bDescriptorType	Tipo do descritor	0x24
bDescriptorSubtype	Subtipo do descritor	0x01
bTerminalLink	ID do terminal conectado	0x03
bDelay	Atraso introduzido pelo envio dos dados	0x01
wFormatTag	Formato do áudio enviado pelo *endpoint*	0x0001

Fonte: Autor.

O campo *bDescriptorSubtype*, por sua vez, recebe o valor da constante AS_GENERAL, conforme indicado na especificação de classe áudio USB [Ashour et al. (1998b)]. O campo *bTerminalLink* indica o valor do ID do terminal ao qual ele está conectado. Neste caso, é o terminal de saída de ID número 0x03. Já o campo *bDelay* corresponde ao atraso introduzido no caminho de dados e é medido em *frames* USB, que têm duração de 1 ms [Ashour et al. (1998b)]. Uma vez que essa informação não é utilizada para realização de cálculos relevantes, estipulou-se esse tempo como um *frame*.

Na Tabela 18, observa-se o descritor de tipo de formato da interface *AudioStreaming*. O campo *bDescriptorType* tem atribuído o valor da constante CS_INTERFACE [Ashour et al. (1998a)]. O campo *bFormatType* recebe a constante FORMAT_TYPE_I [Ashour et al. (1998a)], em conformidade com o valor de *wFormatTag* do descritor anterior. Conforme já especificado anteriormente, *bNrChannels* tem valor 0x06. O valor do campo *bSubframeSize* indica o tamanho em bytes de cada *subframe* de áudio. Um *subframe* corresponde a uma amostra de áudio de um canal. Anteriormente neste documento, já havia sido definido que o sinal PCM teria 16 bits. Portanto, cada *subframe* tem dois bytes. Da mesma forma, *bBitResolution* recebe o valor 16 (0x10 em hexadecimal).

O valor do campo *bSamFreqTime* indica o número de frequências de amostragens suportadas pelo dispositivo. Neste projeto, o dispositivo foi projetado para operar somente a 48 kHz. Por isso, *bSamFreqTime* recebe o valor 0x01. Da mesma forma, *tSamFreq[1]* é igual a 48000 (0x00BB80 em hexadecimal).

Tabela 18: Descritor de tipo de formato utilizado no dispositivo do projeto.

Campo	Descrição	Valor
bLength	Tamanho do descritor em bytes	0x0B
bDescriptorType	Tipo do descritor	0x24
bDescriptorSubType	Subtipo do descritor	0x02
bFormatType	Formato de áudio utilizado pela interface	0x01
bNrChannels	Número de canais de áudio	0x06
bSubframeSize	Tamanho em bytes de uma *subframe* de áudio	0x02
bBitResolution	Número de bits utilizados em cada *subframe*	0x10
bSamFreqTime	Frequências de amostragem suportadas	0x01
tSamFreq[1]	Frequência de amostragem em Hz	0x00BB80

Fonte: Autor.

Tabela 19: Descritor padrão do *endpoint* utilizado no dispositivo do projeto.

Campo	Descrição	Valor
bLength	Tamanho do descritor em bytes	0x09
bDescriptorType	Tipo do descritor	0x05
bEndpointAddress	Endereço do *endpoint*	0x81
bmAttributes	*Bitmap* com atributos deste *endpoint*	0x05
wMaxPacketSize	Máximo tamanho de pacote que esse *endpoint* envia	0x0240
bInterval	Intervalo entre requisições de transferências de dados	0x01
bRefresh	Intervalo entre envios da *pipe* de sincronização	0x00
bSynchAddress	Endereço do *endpoint* de sincronização, se requerido	0x00

Fonte: Autor.

Na Tabela 19, pode ser visto o descritor padrão de *endpoint*. O campo *bDescriptorType* recebe a constante ENDPOINT, conforme a Tabela 6. O campo *bEndpointAddress* define o endereço do *endpoint*. Dois *endpoints* diferentes não podem possuir o mesmo endereço, e o seu valor é determinado pelo fato de o *endpoint* ser IN ou OUT. *Endpoints* IN tem o bit MSB igual a 1, enquanto que nos *endpoints* OUT ele é igual a 0 [Ashour et al. (1998b)]. Do ponto de vista do *host* USB, a *stream* de áudio que passa por este *endpoint* é de entrada, portanto ele é do tipo IN. Uma vez que o *endpoint* padrão do tipo *control* recebe o endereço 0, o presente *endpoint* recebe valor 1. Portanto, seu endereço é 0x81.

O campo *bmAttributes* é um *bitmap* com características do *endpoint*. Os bits número zero e um definem o tipo de transferência suportado por aquele *endpoint*. Por se tratar um dispositivo de áudio, este *endpoint* é do tipo *isochronous*, para que possa enviar dados em tempo real. Os valores destes bits para os diferentes tipos de *endpoint* são: 00 para *control*, 01 para *isochronous*, 10 para *bulk* e 11 para *interrupt* [Axelson (2009)]. Portanto, foi definido o valor 01. Os bits número dois e três definem o tipo de sincronização do *endpoint*. Para os três tipos de sincronização, os valores estabelecidos para esses bits são: 01 para assíncrono, 10 para adaptativo e 11 para síncrono [Ashour et al. (1998b)]. Uma vez que o *clock* do sinal de áudio gerado no dispositivo não tem relação síncrona com o *clock* do barramento USB, este *endpoint* é assíncrono e estes bits recebem o valor 01. Os outros bits são reservados e definidos como 0 [Ashour et al. (1998b)]. Portanto, o valor

de *bmAttributes* é 00000101 em binário ou 0x05 em hexadecimal.

O campo *wMaxPacketSize* indica o tamanho máximo de pacote para o *endpoint*. Em *endpoints* do tipo *isochronous*, ocorre o envio de um pacote por *frame*, sabendo que um *frame* tem duração de 1 ms [Axelson (2009)]. O dispositivo foi projetado para enviar seis canais de áudio com frequência de amostragem de 48 kHz. Sabendo que cada amostra tem dois bytes, o fluxo total de dados é 576 kbytes/s. Uma vezes que uma *frame* tem 1 ms, é necessário o envio de um pacote de 576 bytes a cada *frame* para alcançar o fluxo de dados necessário. Portanto, *wMaxPacketSize* recebeu o valor 576, ou 0x0240 em hexadecimal.

O campo *bInterval* é igual ao intervalo, em milissegundos, que o *host* deve utilizar para requisitar dados daquele *endpoint*. Para dispositivos de áudio, este intervalo é de 1 ms [Ashour et al. (1998b)]. O campo *bRefresh* indica a taxa em que a *pipe* de sincronização envia dados de *feedback* de sincronização [Ashour et al. (1998b)]. Neste dispositivo, não é utilizado *endpoint* de sincronização, e portanto este valor é definido como 0x00. Pela mesma razão, o campo *bSynchAddress* também recebe o valor 0x00.

Tabela 20: Descritor específico de classe do *endpoint* utilizado no dispositivo do projeto.

Campo	Descrição	Valor
bLength	Tamanho do descritor em bytes	0x07
bDescriptorType	Tipo do descritor	0x25
bDescriptorSubType	Subtipo do descritor	0x01
bmAttributes	*Bitmap* com atributos do *endpoint*	0x80
bLockedDelayUnits	Unidade utilizada para o campo *wLockDelay*	0x02
wLockDelay	Tempo necessário para acertar circuito de *clock*	0x0000

Fonte: Autor.

Na Tabela 20, observa-se o descritor específico de classe do *endpoint*. O campo *bDescriptorType* recebe o valor da constante CS_ENDPOINT, conforme indicado pela especificação da classe áudio USB [Ashour et al. (1998b)]. Pelo mesmo motivo, o campo *bDescriptorSubType* recebe o valor da constante EP_GENERAL.

O campo *bmAttributes* é um *bitmap* com atributos do *endpoint*. Os bits número zero e um, se iguais a 1, indicam que o *endpoint* oferece controle da frequência de amostragem e de *pitch*. Uma vez que essa funcionalidade não está prevista no dispositivo, estes bits foram definidos como 0. Já o bit número sete indica que o *endpoint* suporta pacotes somente com tamanho igual a *wMaxPackets*, que neste caso é igual a 576. Uma vez que todos os envios de pacotes são feitos com este tamanho, o bit número sete é igual a 1. Os outros bits são reservados e devem ser definidos como 0. Logo, o valor de *bmAttributes* é 10000000 em binário ou 0x80 em hexadecimal.

Os campos *bLockedDelayUnits* e *wLockDelay* são utilizados principalmente por *endpoints* síncronos e indicam quanto tempo o *clock* interno do dispositivo leva para se adequar ao *clock* do barramento USB. *bLockedDelayUnits* define a unidade em que *wLockDelay* é definido, e o valor 0x02 significa que este valor é medido em amostras PCM [Ashour et al. (1998b)]. Para *endpoints* assíncronos, *wLockDelay* deve ser igual a 0x0000 [Silicon (2006)].

Assim conclui-se a apresentação dos descritores utilizados e dos valores atribuídos a eles. Todos estes valores foram programados no *firmware* utilizado no controlador USB Cypress *Ez-Host*. Para apresentar como isto foi feito, primeiramente será explicado o processo utilizado para criar o ambiente de *software* necessário para criar, compilar e

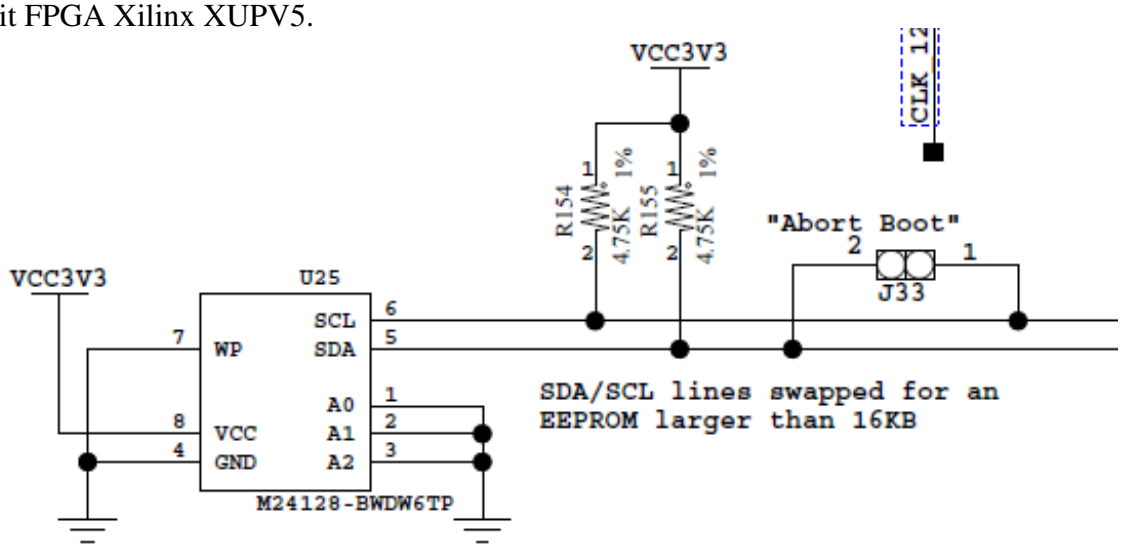
enviar o *firmware* para o Cypress *Ez-Host*.

3.3.4 *Firmware* do controlador USB

O controlador USB Cypress *Ez-Host* presente no kit Xilinx XUPV5 é diretamente conectado à porta USB da placa [Xilinx (2011)]. Além disso, ele se conecta, via I^2C, a uma memória EEPROM externa. No barramento também são conectados resistores de pull-up, o que garante que o tensão nestes pinos seja igual à da alimentação durante a inicialização do controlador. Uma vez que estes pinos são também os pinos de *bootstrap* do controlador, ele opera sempre em modo *standalone*, conforme a Figura 28.

Além dos resistores de pull-up, o barramento I^2C é conectado a um jumper, conforme o diagrama da Figura 36. Quando este jumper é conectado, os dois pinos do barramento são conectados entre si, o que impossibilita a comunicação entre o controlador USB e a memória EEPROM. Este artifício é utilizado quando não se deseja que o controlador seja inicializado a partir do *firmware* contido na EEPROM, mas sim somente da sua própria BIOS.

Figura 36: Conexão entre o controlador USB Cypress Ex-Host e a memória EEPROM no kit FPGA Xilinx XUPV5.



Fonte: Xilinx (2008)

Para a utilização do controlador USB, primeiramente foi instalado em um computador com sistema Windows XP o *software* SuiteUSB versão 3.4. Este *software* é providenciado gratuitamente pela Cypress e contém o *driver* USB [Cypress (2011)]. Este *driver* é necessário para que o sistema operacional possa se comunicar corretamente com o controlador USB antes do envio do seu *firmware*.

Foi também instalado o *software* referente ao kit de avaliação do controlador USB Cypress *Ez-Host*, chamado CY3663. Este *software* é igualmente disponibilizado pela Cypress. A instalação deste *software* deu acesso a muitos arquivos úteis ao projeto, tais como documentação do controlador USB, códigos fonte do *firmware* em linguagem C, compilador, *linker* e outras ferramentas necessárias.

Os *firmwares* disponibilizados são agrupados em *simple examples* (SE) e *design examples* (DE). No total, existem 10 pastas de *firmwares* SE e 4 pastas de *firmware* DE disponibilizadas. Os DE's são exemplos completos, testados pelo sistema de qualidade da

Cypress, que demonstram funcionalidades do controlador USB [Narayana (2011)]. Já SE's são exemplos simples utilizados para demonstrar princípios e são explicados detalhadamente em um livro disponibilizado juntamente com o *software* do kit CY3663 [Hyde (2003)].

Os *firmwares* também são agrupados nas categorias *coprocessor* e *standalone*, indicando qual modo de funcionamento do controlador USB eles utilizam. Neste projeto, interessa-se somente ao modo *standalone*, uma vez que a conexão dos pinos de *bootstrap* do controlador no kit Xilinx XUPV5 permite apenas este modo.

Os arquivos de *firmwares* citados até aqui correspondem à camada de aplicação, sendo a terceira camada da arquitetura representada na Figura 29. A instalação do *software* do kit CY3663 também disponibiliza o *firmware* da camada *Frameworks*. Ele é constituído de diversos arquivos de código C contidos em uma pasta comum e tem como função interagir com a BIOS do controlador USB.

O *firmware* base utilizado para o desenvolvimento do *firmware* do projeto foi o SE3. Mesmo que fosse mais desejável utilizar um *firmware* DE como base, por sua maior confiabilidade, não havia nenhum *firmware* DE em que o controlador se comportasse como dispositivo USB e não *host*, para o modo *standalone*. Entre os SE's, o SE3 é o único que *firmware* de dispositivo USB que implementa funcionalidades similares às requeridas pelo projeto.

O SE3 é a implementação de um dispositivo com luzes e botões, em que o dispositivo envia para o *host* informações sobre quais botões foram pressionados, e o *host* envia para o dispositivo quais as luzes devem ser acesas. Uma vantagem da utilização do SE3 é o detalhamento da estrutura do *firmware* existente no livro [Hyde (2003)]. A elaboração do *firmware* do projeto consistiu portanto na modificação do *firmware* SE3 para desempenhar as funcionalidades deste projeto, o que foi feito principalmente pela alteração dos descritores e da adição e modificação de funções no código.

O *firmware* é desenvolvido com o GNU Toolset em conjunto com o emulador UNIX da Cygnus, o Cygwin. Nele, estão contidas as ferramentas necessárias para a geração, a partir do código fonte em linguagem C, do arquivo binário que é passado para o controlador USB, tais como compilador, *assembler*, *make*, *linker* e *debugger*. Estas ferramentas estão presentes na instalação do *software* do kit CY3663 [Hyde (2003)].

Este ambiente é aberto por um arquivo executável presente na mesma pasta do *firmware* disponibilizado pelo CY3663. O processo de programação do controlador USB inicia-se com a execução deste arquivo. Em seguida, a execução do comando *make* na janela que se abre realiza o processo de compilação do *firmware* e geração do arquivo binário correspondente.

Ao invés de enviar o arquivo binário diretamente para o controlador USB, optou-se por utilizar uma ferramenta que permite que ele seja escrito diretamente na memória EEPROM, para que, quando o controlador seja inicializado, ele leia diretamente o *firmware* da memória EEPROM. Para isso, é necessário adicionar ao arquivo binário os *scan records* [Hyde (2003)]. Isso é feito através do comando *scanwrap*, seguido do nome do arquivo do binário, o nome do arquivo binário a ser gerado e o endereço de base em que o *firmware* é salvo na EEPROM. Finalmente, o envio do *firmware* para a EEPROM é feito através da ferramenta *qtui2c*, também contida no Toolset instalado [Hyde (2003)].

Para este envio ser possível, é necessário que o controlador USB tenha sido inicializado com o jumper conectando os pinos do barramento I^2C. Caso contrário, o controlador teria sido inicializado com o último *firmware* registrado na EEPROM, e não mais seria reconhecido pelo computador como um dispositivo Cypress, mas sim pelas características

indicadas no descritor contido no *firmware* antigo.

Sabendo de como o ambiente de *software* foi instalado, serão agoras expostas as modificações realizadas no *firmware* do SE3, utilizado como base, para criar o *firmware* utilizado no dispositivo do projeto. Estas modificações foram feitas em três arquivos de código de linguagem C: os *headers app.h* e *fwxcfg.h* e o arquivo de código *app.c*.

Primeiramente, foram feitas modificações nas estruturas de dados do código que representavam os descritores. Foram também realizadas modificações para que o dispositivo pudesse responder requisições USB específicas desta aplicação. A maior parte das requisições já são diretamentes respondidas pela BIOS do controlador USB e não precisam ser previstas em *firmware*. Finalmente, foram também feitas funções para que a informação proveniente dos canais de áudio recebidas via HPI pudessem ser escritas no *buffer* do *endpoint isochronous* do dispositivo a cada *frame* USB.

O arquivo *fwxcfg.h* contém diversas macros que são definidas ou não, de acordo com a aplicação. Estas macros selecionam características do *firmware* da camada *Frameworks* [Hyde (2003)]. No *firmware* do SE3, este arquivo é utilizado para definir o SIE1 como o utilizado pela aplicação. No entanto, no kit Xilinx XUPV5, a porta USB é conectada à saída do SIE2. Portanto, todas as macros definidas para ativar o SIE1 foram desativadas. No lugar, todas as macros correspondentes relativas ao SIE2 foram definidas. Outra modificação realizada neste arquivo foi a definição da macro FWX_TIMER0_NOTIFY_1, o que habilitou as interrupções do *timer* 0 no controlador. Este *timer* foi utilizado na aplicação para gerar interrupções periódicas de 1 ms, utilizadas para a escrita de dados no *buffer* do *endpoint isochronous*.

Os arquivos *app.c* e *app.h* sofreram mais modificações, uma vez que neles são definidas as funções e variáveis específicas da aplicação. Os descritores foram armazenados em variáveis do tipo *struct* definidas neste código. À exceção do descritor de dispositivo e do descritor *string*, todos os outros descritores foram armazenados na *struct* USB_ALL_DESCRIPTORS. Uma vez que o descritor de dispositivo do *firmware* original do SE3 não foi modificado, foi mantida a mesma variável para representá-lo.

O descritor *string* foi colocado na variável *device_strings*, que utilizou o tipo UNICODE_STRING_DESCRIPTOR, criado na camada *Frameworks* do *firmware*. A esta variável foi atribuído os valores das três *strings* do descritor apresentadas anteriormente.

A variável USB_ALL_DESCRIPTORS também já existia no *firmware* original, e foi modificada para conter o valor dos novos descritores. Cada um dos descritores é representado por uma *struct* diferente, todos contidos em USB_ALL_DESCRIPTORS. Cada campo dos descritores foi representado por uma variável de tipo *uint8*, para os campos de apenas um byte, e tipo *uint16* para os campos de dois bytes. Os diferentes campos das *structs* de cada um dos descritores são definidos no arquivo *app.h*, enquanto que os seus valores são atribuídos na inicialização das variáveis, no arquivo *app.c*. Os valores atribuídos às variáveis correspondem aos valores escolhidos para os descritores anteriormente apresentados.

Considerando o modelo de camadas da Figura 29, o *firmware* desenvolvido não apresenta *Idle_Task* na camada de aplicação. Essa característica já estava presente no *firmware* do SE3. Ele é formado, portanto, apenas por funções executadas durante a inicialização (*Init_Task*) e em resposta a eventos específicos (*CallBack Tasks*).

A função executada na inicialização é chamada de *app_init*. Dentro desta função do *firmware* SE3, foram retiradas diversas funções que interagiam com os botões e luzes da aplicação do SE3. Para o envio dos descritores, a BIOS possui endereços definidos na memória que contêm ponteiros para as variáveis dos descritores. Utilizou-se a fun-

ção WRITE_REGISTER, dentro da função *app_init*, para escrever o endereço das variáveis criadas para os descritores nestes endereços fixos de memória. Isso foi feito para o descritor de dispositivo, o descritor *string* e para a variável USB_ALL_DESCRIPTORS contendo todos os outros descritores. Deste modo, quando o *host* USB envia requisições do tipo *GetDescriptor*, a BIOS do controlador USB automaticamente responde esta requisição enviando as estruturas de dados contidas nestes endereços de memória, sem a necessidade que este envio seja feito no *firmware*.

A rotina da inicialização também foi utilizada para ativar rotinas de *callback* específicas da aplicação deste projeto que não são realizadas pela BIOS. Para isso, utiliza-se a função READ_REGISTER para salvar os ponteiros das funções padrão da BIOS executadas durante eventos específicos, e em seguida usa-se a função WRITE_REGISTER para escrever nesses endereços de memória ponteiros de funções definidas no código. Esse processo é feito para três funções da BIOS. Finalmente, na inicialização são executadas também a função *BIOSHpiInit*, que inicializa a interface HPI do controlador, e a função *susb_init*, que inicializa a interface SIE2 no modo *full speed*. A seguir são apresentadas as três funções que interceptam as rotinas da BIOS para eventos específicos, fazendo parte do grupo *Callback Tasks*.

A primeira função interceptada da BIOS é a resposta a requisições *setConfiguration*. O objetivo desta interceptação foi pode atribuir o valor 1 à *flag configured*. Desta maneira, todas as outras funções de *callback* definidas no código só são executadas quando o valor desta *flag* for 1. Isto previne o dispositivo USB de responder a requisições USB antes de ser devidamente configurado.

A segunda função interceptada foi a resposta a requisições padrão do *host* USB. O objetivo desta função resume-se a modificar o valor da *flag alternateSettingCorrect*. Dentro da função gerada para responder a este evento, observa-se se a requisição é do tipo *SetInterface* e se ela tem como endereço a interface número 1, que é a interface de tipo *AudioStreaming*. Se for o caso, a requisição tem por objetivo selecionar uma das *alternate settings* da interface. Sabendo que apenas a *alternate setting* de número 1 possui o *endpoint isochronous*, a *flag alternateSettingCorrect* é igualada 1 apenas se a requisição *SetInterface* for direcionada a ela. Esta *flag* controla a função que envia os sinais de áudio, e o controle é feito de modo que o dispositivo envie o sinal de áudio apenas quando a *alternate setting* selecionada for a de valor 1.

A terceira função interceptada é a resposta a requisições específicas de classe. Essas requisições são enviadas aos terminais e *units* da função áudio. A *feature unit*, por implementar o controle de *mute* no canal master, recebe requisições do tipo GET_CUR para o seu atributo *mute*. Quando o *host* USB envia esta requisição, ele espera receber o valor atual deste atributo no dispositivo. Logo, a função de intercepção verifica se esta é a requisição e, se for o caso, envia o valor zero, indicando que o canal master não está em *mute*. Além disso, os terminais e *units* podem receber requisições USB do tipo GET_MEM. Esta requisição é utilizada para o envio de um bloco de parâmetros genéricos [Ashour et al. (1998b)]. Durante a fase de testes, verificou-se que os terminais e *units* estavam recebendo estas requisições, mesmo sem que tenham um propósito específico. Por isso, no caso de recepção desta requisição, esta função de interceptação foi programada para enviar uma sequência de zeros do tamanho que foi requisitado.

As três funções apresentadas constituem o grupo de *Callback Tasks*. Como observado, os eventos específicos aos quais elas respondem são requisições USB. Se estas funções não fossem implementadas, a BIOS do controlador USB responderia estas requisições com o código STALL [Hyde (2003)], significando que os controles específicos da aplica-

ção não são suportados pelo dispositivo. Durante a fase de testes, observou-se que este comportamento não permitia a correta enumeração do dispositivo. Sem as funções implementadas, o sistema Windows, utilizado nos testes, indicava que o dispositivo não havia sido reconhecido corretamente quando conectado ao computador.

A descoberta de quais requisição específicas enviadas pelo *host* USB deveriam ser respondidas nas *Callback Tasks* também foi feita durante a fase de testes. O envio destas requisições é definido pelo conjunto de *drivers* USB que atuam no momento da enumeração. Durante o desenvolvimento do *firmware*, o dispositivo foi conectado a uma porta USB de um computador com sistema operacional Windows e foram observadas as requisições USB enviadas para o dispositivo através do *software* USBLyzer. Assim, observou-se quais requisições o dispositivo deveria responder para ser corretamente enumerado, e assim foram implementadas as funções de *callback* para respondê-las de forma adequada. O *driver* USB que define se o dispositivo pertence à classe áudio é o *Usbaudio.sys*. Uma vez que este *driver* é exclusivo de sistemas Windows, não se sabe se o dispositivo seria corretamente enumerado em outros sistemas operacionais, uma vez que outras requisições USB poderiam ser feitas que não estão previstas no *firmware*.

As funções do *firmware* até aqui citadas são responsáveis por garantir a enumeração do dispositivo. No entanto, o envio dos sinais de áudio é realizado por um conjunto de funções acionadas por interrupções do *timer* 0. Este *timer* é ativado pelo *Frameworks* quando a macro FWX_TIMER0_NOTIFY_1 é definida. O tempo entre interrupções é definido pela variável FWX_TIMER0_COUNT_VALUE, que no *firmware* do SE3, utilizado como base, já definia o tempo entre interrupções como 1 ms.

A cada interrupção do *timer* 0 é executada a função *timer0_notify_1*. Nesta função, é executada a função *susb_send* caso as *flags configured* e *alternateSettingCorrect* forem iguais a 1. Esta condição significa que a única configuração do dispositivo está selecionada e que a *alternate setting* número 1 da interface *AudioStreaming* está selecionada. Os parâmetros da função *susb_send* são a variável *audio_report_info*, do tipo USBTXRXINFO, o *endpoint* e o SIE utilizado. Os dois últimos receberam o valor 1 e SIE2, respectivamente.

O tipo USBTXRXINFO é uma *struct* definida no *Frameworks*, e contém como atributos os campos *buffer* e *length*. O parâmetro *buffer* é o endereço em memória da variável a ser enviada pelo USB, e o parâmetro *length* é o tamanho em bytes da variável enviada. Juntos, estes dois atributos definem a porção de memória onde se encontram os dados enviados no *endpoint* designado em um *frame* USB. Conforme definido anteriormente, o dispositivo do projeto foi projetado para enviar 576 bytes a cada *frame* USB para atender ao fluxo de dados relativos aos seis canais de áudio. Por isso, o parâmetro *length* é definido como 576. Já o valor do parâmetro *buffer* foi escolhido como o valor fixo 0x0700. Deste modo, a cada microssegundo, ocorre o envio de uma porção de memória começando no endereço 0x0700 e com um total de 576 bytes através do *endpoint* do tipo *isochronous* do dispositivo. Esta porção de memória é, portanto, a faixa com endereço inicial 0x0700 e endereço final 0x093F.

As modificações efetuadas no *firmware* base do SE3 até aqui apresentadas foram suficientes para criar o *firmware* utilizado no dispositivo do projeto. Uma vez finalizadas as modificações, o arquivo binário foi gerado e passado para a EEPROM do controlador USB Cypress *Ez-Host* seguindo o processo anteriormente descrito.

3.3.5 Interface HPI em FPGA

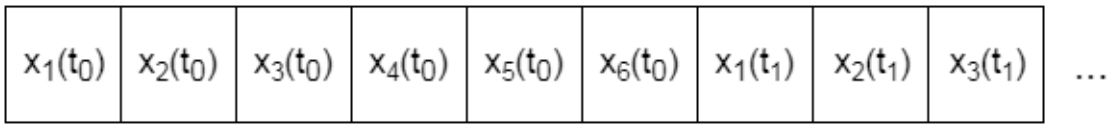
Conforme explicado, os dados enviados através do *endpoint isochronous* do dispositivo correspondem a uma porção fixa da memória. Durante a operação do dispositivo, o *host* USB reconhece estes dados como provenientes dos seis canais de áudio USB. No entanto, não existe nenhuma função do *firmware* do controlador USB que escreve dados nestes endereços. A escrita nessa região da memória é feita pela própria FPGA, através do protocolo HPI. Uma vez que este protocolo fornece acesso direto à memória do controlador, não são necessárias funções em *firmware* que escrevam nestes endereços. Esta faixa de memória foi justamente escolhida por se tratar de uma porção da RAM definida como livre pelo controlador USB Cypress *Ez-Host* [Hyde (2003)].

A escrita dos dados PCM dos canais de áudio via HPI é feita por um componente codificado em VHDL contido na FPGA. Este componente escreve os dados não somente na região de memória definida pelo *firmware*, mas também na ordem correta. Uma vez que o controlador envia os dados desta faixa sem fazer nenhuma modificação, a ordem dos dados deve ser a mesma que o *host* USB utiliza para reconstruir os sinais de áudio.

Esta ordem de dados é definda no protocolo USB [Ashour et al. (1998a)], e consiste na organização correta de todas as amostras PCM dos seis canais de um intervalo de 1 ms. Cada amostra PCM corresponde a um *subframe* áudio. Conforme definido anteriormente, cada *subframe* é formado por dois bytes, uma vez que uma amostra PCM tem 16 bits. Um *frame* áudio é uma coleção de *subframes*, cada uma contendo uma amostra de um canal diferente [Ashour et al. (1998a)]. Portanto, no caso do dispositivo de seis canais deste projeto, cada *frame* áudio é formada de seis *subframes*. Já uma *stream* áudio é uma coleção de *frames* áudio, ordenadas em tempo crescente. A ordem dos diferentes canais dentro do *frame* áudio deve ser mantida constante dentro da *stream* áudio [Ashour et al. (1998a)].

A partir desta descrição, conclui-se que a porção de memória enviada pelo controlador USB a cada milissegundo é uma *stream* áudio contendo 48 *frames*, cada *frame* com seis *subframes*. Deste modo, não somente todos os canais são enviados, como também garante-se a frequência de amostragem de 48 kHz. As amostras de áudio PCM são portanto organizadas nesta porção de memória conforme a Figura 37. Cada valor $x_n[t_i]$ corresponde a uma amostra PCM do canal n no instante de tempo t_i.

Figura 37: Organização das amostras de áudio na USB *frame*.



Fonte: Autor.

A escrita dos dados na memória do controlador USB via HPI foi foi projetada baseando-se na Figura 33. Assim como na Figura, a escrita HPI de um dado na memória constitui no envio primeiramente do endereço de memória, seguido do envio do dado em si. Durante o envio do endereço de memória, os sinais de controle HPI_A[1:0] são mantidos em 10, de modo que o valor enviado no barramento de dados seja escrito no registrador HPI ADDRESS. Já durante o envio do dado em si, HPI_A[1:0] é mantido em 00, para que se escreva no registrador HPI DATA, conforme a Figura 31. Uma vez que o barramento de dados HPI tem o mesmo número de bits que uma amostra PCM, cada operação de escrita corresponde a uma amostra PCM.

As transições dos sinais de controle do dispositivo foram também projetadas conforme a Figura 33. Para uma operação de escrita, parte-se de um estado onde HPI_nCS e HPI_nRD são iguais a 1. Em seguida, HPI_nCS vai para 0 e após isso, HPI_nWR vai para 0. Após isso, HPI_nWR vai para 1 de novo. Neste instante, o valor no barramento de dados é adquirido pelo controlador USB [Swanbeck (2011)]. Finalmente, HPI_nCS vai para 1 também. Durante este processo, o sinal HPI_nRD fica todo tempo em 1. O valor do barramento de dados e do sinal HPI_A[1:0] são mudados apenas quando os três sinais de controle mencionados estão em 1. Os intervalos entre cada uma destas transições foram projetados para serem constantes.

O componente VHDL projetado para realizar a comunicação HPI entre a FPGA e o controlador USB é chamado *hpi_interface*, e recebe como entrada os seis sinais PCM de 16 bits provenientes do componente *pdm_pcm_converter*. Além disso recebe dois sinais de *clock*, um de frequência igual à frequência dos sinais PCM, isso é, 48 kHz, o outro de 100 MHz. A saída deste bloco são os sinais do protocolo HPI: HPI_D[15:0] (barramento de dados), HPI_A[1:0], HPI_nCS, HPI_nWR e HPI_nRD. O sinal HPI_INT é também uma entrada do componente, uma vez que é controlado pelo controlador USB, mas não é utilizado para a escrita dos sinais na memória do controlador.

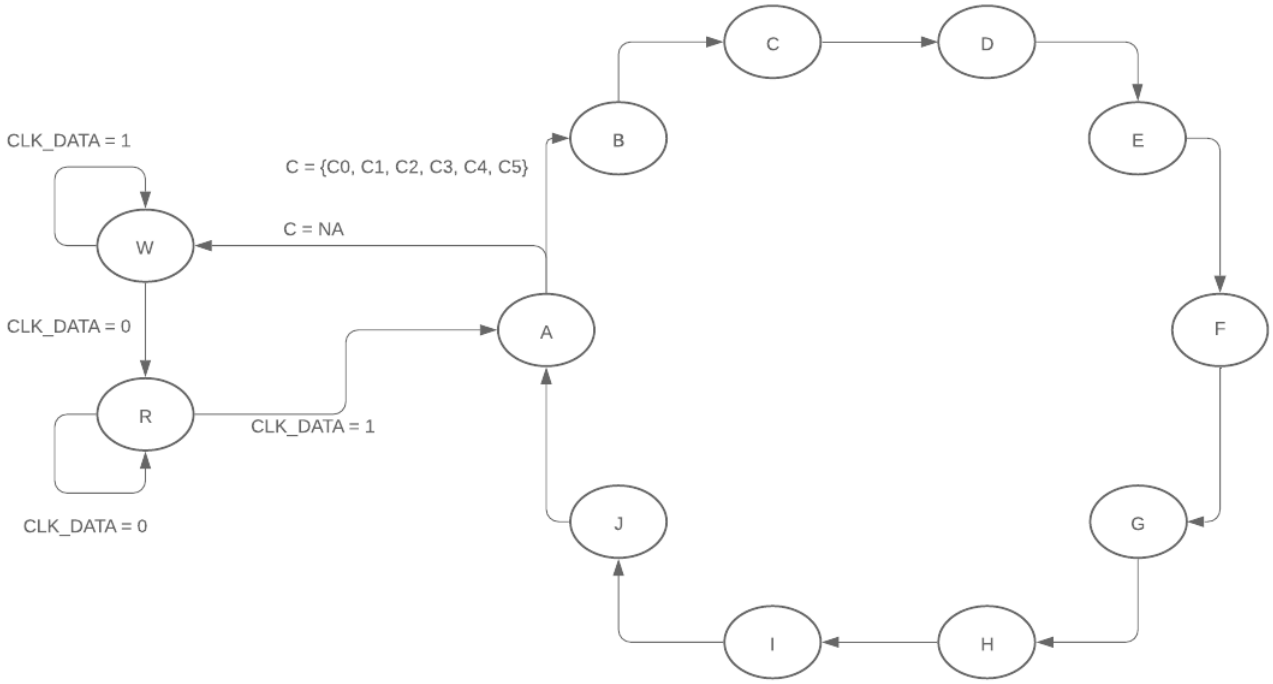
O componente *hpi_interface* foi projetado para escrever os dados dois seis sinais PCM de entrada a cada ciclo do sinal de *clock* de 48 kHz. Este sinal de *clock* corresponde a um dos sinais de *clock* produzidos na saída do módulo de transcodificação PDM-PCM. Uma vez que cada componente *pdm_pcm_converter* deste módulo produz um sinal de *clock* sincronizado com o sinal PCM de saída, e todos os sinais PCM são sincronizados entre si, qualquer um dos sinais de *clock* produzidos pode ser utilizado como entrada do componente *hpi_interface*.

A cada borda positiva deste *clock* de 48 kHz, o componente escreve, em ordem, o valor de cada uma das seis entradas de 16 bits na memória do controlador USB utilizando a ordem de transições dos sinais de controle HPI anteriormente explicadas. Para cada canal, escreve-se primeiramente o endereço de memória seguido do dado em si. A cada canal escrito, o endereço de memória é incrementado de 2, uma vez que o endereçamento da memória do controlador USB é feito por bytes, e cada amostra PCM ocupa 2 bytes. O endereço de memória inicial é 0x0700. No momento em que foram escritos todos os 576 bytes de memória do controlador USB designados para escrita via HPI, volta-se a escrever no endereço inicial. O endereço final de escrita é, portanto, 0x093E. Este ciclo de escrita nesta faixa de memória se repete indefinidamente enquanto o dispositivo está ativo. Em um ciclo do *clock* de 48 kHz, uma vez que os dados de todas as entradas são escritos na memória do controlador, o componente *hpi_interface* para de enviar dados no barramento HPI, esperando pela próxima borda positiva do *clock*. O funcionamento descrito garante que a ordem dos dados na memória do controlador USB sejá compatível com o estipulado na Figura 37.

O funcionamento deste componente é implementado através de uma máquina de estados. Cada estado corresponde a uma configuração de valores dos sinais de controle HPI. Dez estados (A a J) são utilizados para envio dos dados via HPI, enquanto outros dois (W e R) são estados de espera. A cada ciclo do *clock* de 48 kHz, a máquina de estados passa pelos estados A a J em ordem crescente seis vezes, e depois passa pelos estados de espera W e R. Os estados A a E realizam a escrita do endereço de memória, enquanto que os estados F a J realizam a escrita dos dados dos canais PCM.

O diagrama temporal da máquina de estados é apresentado na Figura 38. Já os sinais de controle HPI para cada estado são apresentados na Tabela 21.

Figura 38: Máquina de estados implementada na interface HPI.



Fonte: Autor.

Se cada estado possui valores constantes associados para cada um dos sinais de controle HPI, isso é, HPI_nWR, HPI_nRD, HPI_nCS e HPI_A, o mesmo não pode ser dito para o barramento de dados. Cada volta pelos estados A a J escreve um endereço e um dado diferente. Enquanto que o endereço é um valor que começa em 0x0700 e é incrementado de 2 a cada escrita, o dado escrito corresponde cada vez a uma entrada diferente do componente VHDL *hpi_interface*. Para implementar esta funcionalidade, a máquina de estados possui duas variáveis, *address* e *channel*.

A variável *address* tem 16 bits e recebe inicialmente o valor 0x0700. A cada transição entre o estado J e A, esta variável é incrementada em 2. Se a variável tiver atingido o valor máximo 0x093E, ela recebe novamente o valor 0x0700. Durante os estados A a E, o barramento de dados recebe o valor desta variável *address*.

A variável *channel*, por outro lado, foi definida utilizando um tipo próprio para ela, *channel_type*. Este tipo consiste em um conjunto de sete valores: NA, C0, C1, C2, C3, C4, e C5. Nos estados de repouso W e R, a variável *channel* tem o valor NA. Na transição de R para A, *channel* recebe o valor C0. Na próxima passagem do estado A para o estado B, *channel* é incrementado e recebe o valor C1. Este processo se repete até que *channel* receba o valor C5. Quando a máquina de estados chega ao estado A e *channel* tem valor C5, ele recebe o valor NA e a próxima transição não é mais para o estado B e sim para o estado de repouso W. Deste modo, garante-se que a máquina de estados passa pelos estados A a J seis vezes. Quando a máquina passa pelos estados F a J, o

Tabela 21: Sinais de controle HPI para cada estado da máquina de estados.

Estados	HPI_nCS	HPI_nWR	HPI_nRD	HPI_A	HPI_D
A	1	1	1	10	*address*
B	0	1	1	10	*address*
C	0	0	1	10	*address*
D	0	1	1	10	*address*
E	1	1	1	10	*address*
F	1	1	1	00	*data*
G	0	1	1	00	*data*
H	0	0	1	00	*data*
I	0	1	1	00	*data*
J	1	1	1	00	*data*
W	1	1	1	11	*address*
R	1	1	1	11	*address*

Fonte: Autor.

valor do barramento de dados é igual ao sinal *data*. O valor de *data* depende do valor da variável *channel* (de C0 a C5). Quando *channel* é igual a C0, *data* é igual à primeira entrada de *hpi_interface*. Quando *channel* é igual a C1, *data* é igual à segunda entrada de *hpi_interface*, e assim sucessivamente, de modo que cada valor de *channel* corresponde a uma entrada do componente.

A implementação desta máquina de estados garante que os sinais de controle HPI sigam os valores apresentados na Figura 33. A transição entre os estados A a E ocasiona as mesmas transições nos sinais HPI_nCS e HPI_nWR da Figura, mantendo o valor 10 em HPI_A e o valor do endereço no barramento de dados. Da mesma forma, a transição entre os estados F a J gera as mesmas transições em HPI_nCS e HPI_nWR, mantendo HPI_A em 00 e um dos sinais de entrada no barramento de dados HPI.

Depois de realizar as seis escritas via HPI, a máquina de estados passa para o estado de repouso W. Quando o sinal de *clock* de 48 kHz, também chamado de CLK_DATA, tem borda de descida, a máquina transita do estado W para o estado R. Quando ocorre a próxima borda de subida de CLK_DATA, ocorre transição do estado R para o estado A, onde a variável *channel* adquire o valor C0, conforme já explicado.

As transições da máquina de estados ocorrem sempre na borda de subida de um sinal de *clock* de 5 MHz. Este sinal é gerado a partir do *clock* de entrada de 100 MHz. Para gerar o *clock* de 5 MHz, foi utilizado um processo VHDL com uma variável *counter* de 8 bits. A cada borda de subida do *clock* de 100 MHz, a variável *counter* é incrementada. Quando *counter* atinge o valor 10, o *clock* de saída do processo é igualado 0. Já quando o *counter* atinge o valor 20, o *clock* de saída é igualado a 1 e *counter* recebe o valor 0 novamente. Deste modo, gera-se um *clock* de saída com frequência 20 vezes menor que 100 MHz, ou seja, com 5 MHz.

Cada operação de escrita de um dado na memória do controlador leva dez transições de estados com a máquina de estados implementada. Sabendo que cada transição de estados ocorre no tempo de um período do *clock* de entrada, a escrita de seis dados na memória leva sessenta períodos do *clock*. Esta escrita deve acontecer em um tempo menor que um período de amostragem do sinal PCM de 48 kHz. Utilizando o *clock* de 5 MHz para cadenciar as transições da máquina de estados, o tempo total para completar a escrita é igual a $60 \times (5MHz)^{-1} = 12\mu s$, abaixo do período de amostragem de $(48kHz)^{-1} =$

$20,833\mu s$. Portanto, a frequência escolhida de 5 MHz é suficiente para garantir a taxa de transmissão de dados requerida.

Uma vez implementado o componente *hpi_interface*, foi realizada a simulação de seu código VHDL com o simulador ISim da ferramenta ISE Design Suite. Para a simulação, foram gerados os dois *clocks* de entrada nas frequências estipuladas (5 MHz e 48 kHz), e os sinais de cada uma das entradas foram mantidos constantes. Observou-se se os sinais HPI de saída correspondiam às especificações do protocolo. Os resultados desta simulação são apresentados no capítulo de resultados.

4 RESULTADOS

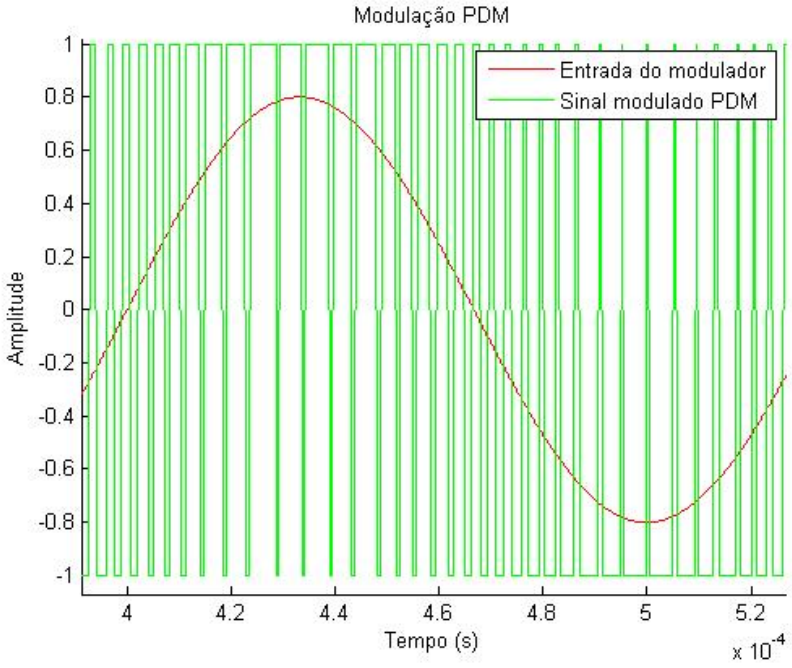
4.1 Transcodificador PDM-PCM

Os resultados do transcodificador PDM-PCM são divididos nas seguintes seções: resultados da simulação dos filtros em MATLAB, resultados da simulação do código VHDL e resultados da implementação.

4.1.1 Resultados da simulação em MATLAB

Considerando os parâmetros de simulação descritos nos capítulos anteriores, o sinal PDM obtido na saída do modulador sigma-delta de quarta ordem simulado pode ser observado na Figura 39.

Figura 39: Representação do sinal modulado PDM.

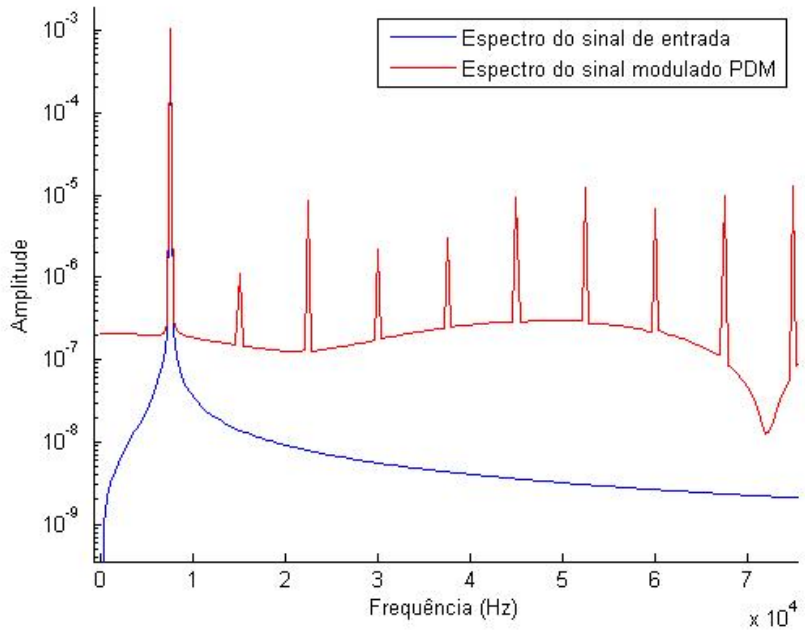


Fonte: Autor.

Durante a simulação, foi também realizada a transformada de Fourier desse sinal e comparada com a do sinal de entrada. O gráfico da Figura 40 mostra o espectro frequencial do sinal PDM até 80 kHz. Observa-se que ele apresenta um pico proeminente em 7,5 kHz, assim como o sinal de entrada. Esse pico é esperado, pois corresponde à frequência

da senoide de entrada. No entanto, o sinal PDM também apresenta picos nos múltiplos da frequência fundamental.

Figura 40: Espectro Frequencial do sinal modulado PDM em baixas frequências.



Fonte: Autor.

O espectro em uma banda mais larga de frequências é mostrado na Figura 41. Observa-se que a distorção em frequências mais altas é ainda maior, gerando picos de amplitude da mesma ordem de grandeza do sinal de entrada. Isso evidencia a necessidade de filtragem nessas frequências para que o sinal de entrada possa ser recuperado com precisão.

O primeiro bloco do transcodificador PDM-PCM na simulação é o filtro CIC, que recebe como entrada o sinal PDM e produz um sinal de saída de frequência de amostragem vinte vezes menor. A sua topologia utilizada na simulação é idêntica à implementada em FPGA, ou seja, cinco integradores em série, seguidos de um bloco decimador e cinco filtros comb.

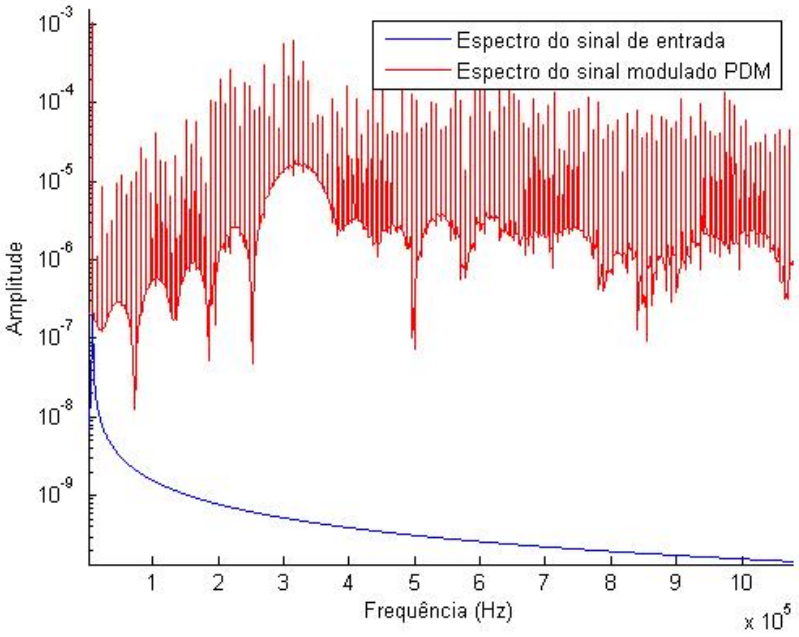
Partindo dessa topologia, o filtro CIC foi simulado tendo como entrada o sinal PDM gerado pelo modulador Sigma-Delta anterior. Obteve-se assim o sinal de saída da Figura 42. Nela, são apresentados o sinal de entrada do modulador sigma-delta, o sinal PDM correspondente e o sinal de saída do filtro CIC. Sabendo que a saída do filtro CIC é na verdade um sinal de 16 bits e está restrito na faixa [-32768, 32767], para se obter o sinal da Figura 42 ele foi dividido pelo valor 32768 para se adequar à faixa [-1, 1].

Observa-se que o sinal de saída apresenta menor amplitude que o sinal de entrada. No entanto, ambos os sinais são senoidais. Isso indica que o filtro CIC permite recuperar o sinal analógico de entrada em formato digital.

Uma análise mais detalhada da distorção do sinal pode ser feita pelo seu espectro frequencial na Figura 43. Observa-se que o pico na frequência do sinal de entrada tem amplitude mais elevada que os picos nas harmônicas, tendo amplitude aproximadamente cem vezes maior, o que corresponde a uma diferença de aproximadamente 40dB.

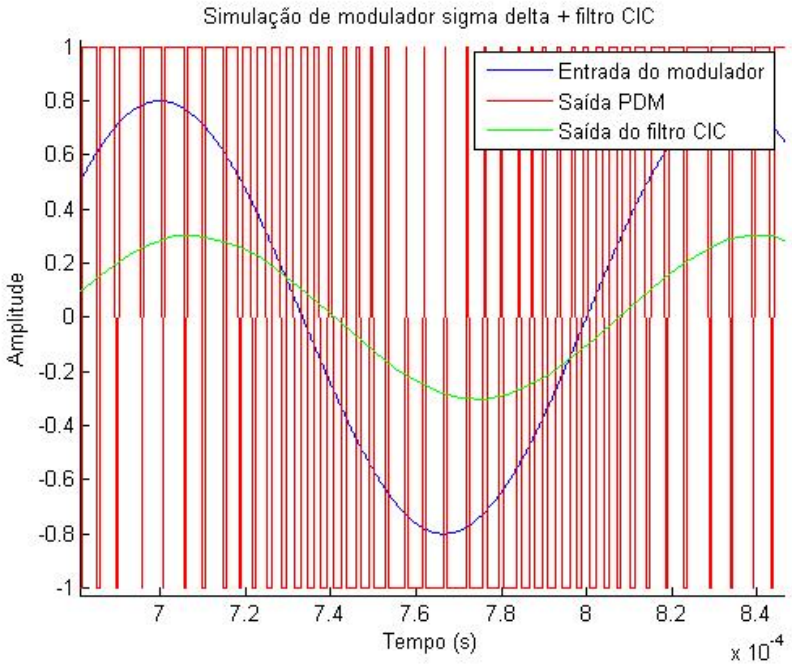
Uma característica importante de ser observada no filtro CIC é o comportamento do sinal nas saídas dos integradores. Conforme explicado no capítulo de revisão bibliográfica, o sinal na saída dos integradores do filtro CIC está sujeito a sofrer *overflow*, mas

Figura 41: Espectro Frequencial do sinal modulado PDM em altas frequências.



Fonte: Autor.

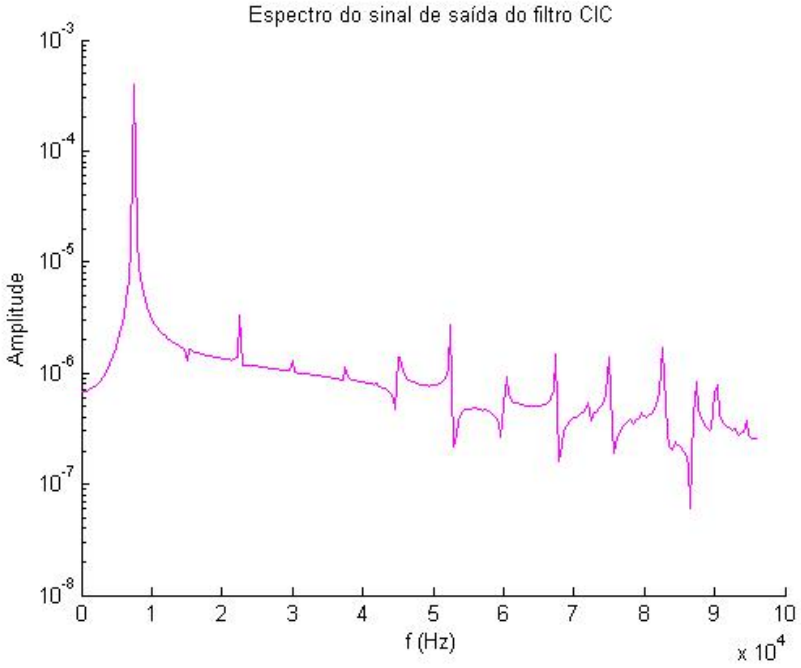
Figura 42: Representação do sinal de saída do filtro CIC.



Fonte: Autor.

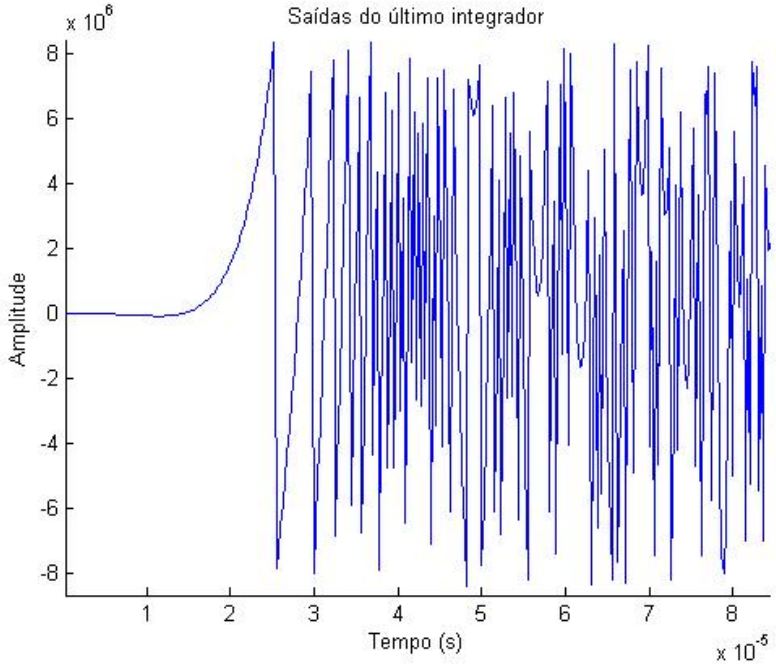
isso não impede o funcionamento normal do filtro. Visto que o sinal de saída do quinto integrador é o resultado de cinco operações de integração encadeadas, ele é o primeiro a sofrer *overflow*. Portanto, traçou-se o gráfico desse sinal, representado na Figura 44.

Figura 43: Espectro frequencial do sinal de saída do filtro CIC.



Fonte: Autor.

Figura 44: Sinal de saída do quinto integrador do filtro CIC.



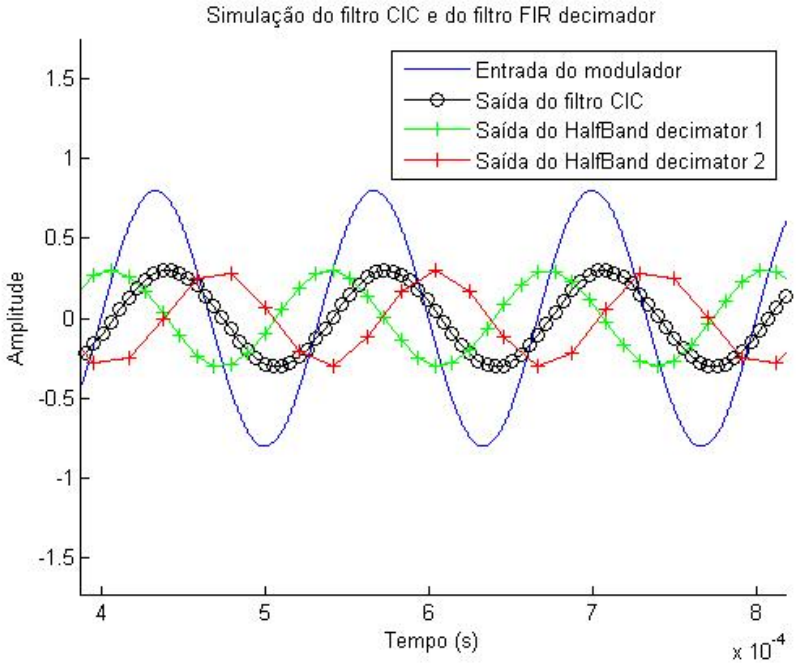
Fonte: Autor.

Observa-se que em pouco mais de vinte microssegundos, esse sinal já tem overflow, e a partir desse ponto isso se repete em uma taxa cada vez mais rápida. No entanto, como observado anteriormente, isso não impede o filtro de funcionar, uma vez que a quantidade de bits mínima para esse sinal é atendida.

O bloco seguinte a ser testado foi o filtro *halfband* decimador, utilizado em duas instâncias após o filtro CIC. Na Figura 45, podem ser vistos os sinais de saída dos filtros

halfband decimadores, comparados com o sinal de entrada do modulador e o sinal de saída do filtro CIC.

Figura 45: Sinal de saída de cada filtro do transcodificador PDM-PCM.



Fonte: Autor.

Observa-se que o sinal de saída dos filtros *halfband* decimadores têm a aparência de uma senoide, além de ter a mesma frequência do sinal de entrada. Além disso, observa-se pelas marcas colocadas em cada amostra que a frequência de amostragem do sinal diminui a cada etapa do filtro, conforme esperado.

Na Figura 46, podem ser observados os sinais da Figura 45 no domínio frequência. Observa-se a formação de lóbulos no domínio frequência próximos ao pico de 7,5kHz. Na saída do segundo *halfband* decimador, o lóbulo mais próximo do pico tem amplitude pouco mais que dez vezes menor. Ou seja, a diferença entre o pico e o lóbulo é um pouco maior que 20dB.

4.1.2 Resultados da simulação VHDL

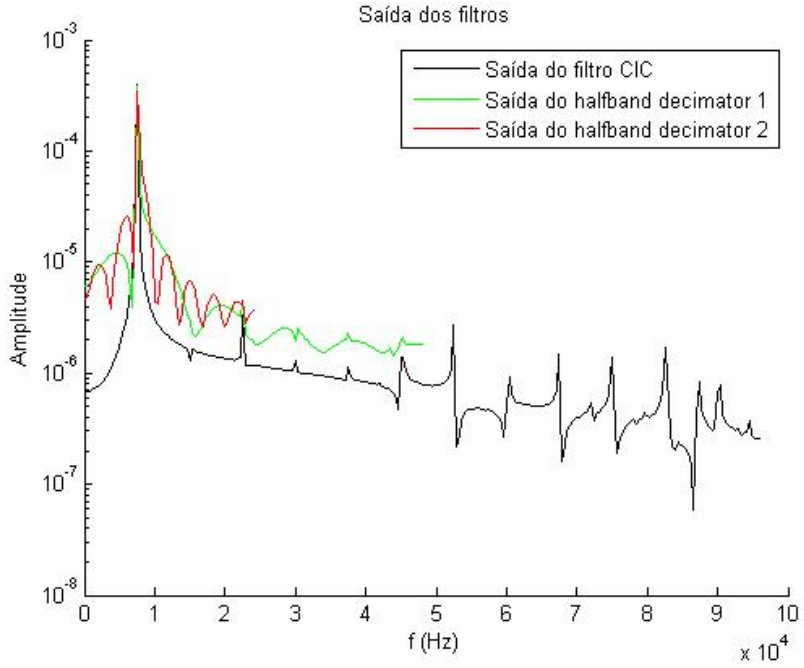
Uma vez completa a descrição em VHDL do transcodificador PDM-PCM, realizou-se a simulação do VHDL a partir da ferramenta ISE Simulator (ISim), o qual faz parte do ISE Design Suite.

Primeiramente, foi feita a simulação do bloco de geração de *clock*. Ele utiliza como entrada o *clock* gerado na própria placa de 100MHz, e gera na sua saída um *clock* de 3,84MHz. Portanto, o resultado esperado para o período do sinal de saída é então de 260,4167ns. Na Figura 47 observa-se o resultado da simulação do bloco.

A Figura 47 mostra dois sinais: acima o *clock* de entrada de 100MHz (CLK_I) e abaixo o sinal de saída (CLK_O). Os cursores estão posicionados a uma distância de um período de *clock* e marcam os tempos 229,992 ns e 490,408 ns, resultando em um período calculado de 260,416 ns.

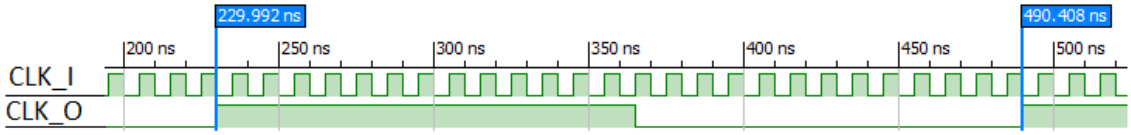
Em seguida, foi simulado o resto do transcodificador PDM-PCM. O sinal PDM de entrada aplicado foi gerado em MATLAB, utilizando um modulador sigma-delta de quarta

Figura 46: Espectro frequencial da saída de cada filtro do transcodificador PDM-PCM.



Fonte: Autor.

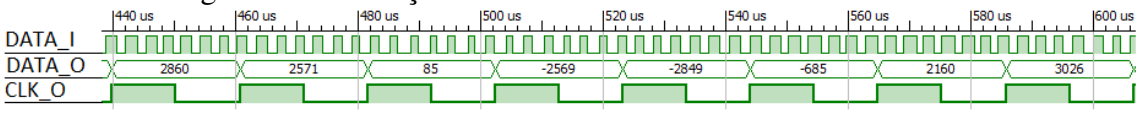
Figura 47: Simulação VHDL do bloco de geração de *clock*.



Fonte: Autor.

ordem a partir de uma senoide de 7,5kHz. Portanto, espera-se recuperar na saída do transcodificador um sinal semelhante a essa senoide com a mesma frequência. O resultado da simulação pode ser visto na Figura 48.

Figura 48: Simulação VHDL do transcodificador PDM-PCM.



Fonte: Autor.

Observam-se três sinais na Figura 48. O primeiro é o sinal de entrada PDM (DATA_I), enquanto que os dois últimos são o sinal PCM de saída (DATA_O) e o *clock* (CLK_O) associado a esse sinal, de 48kHz. Embora não se possa visualizar se o sinal de saída é uma senoide somente com a Figura, observa-se que a diferença entre dois picos nesse caso (2860 e 3026) é de sete ciclos.

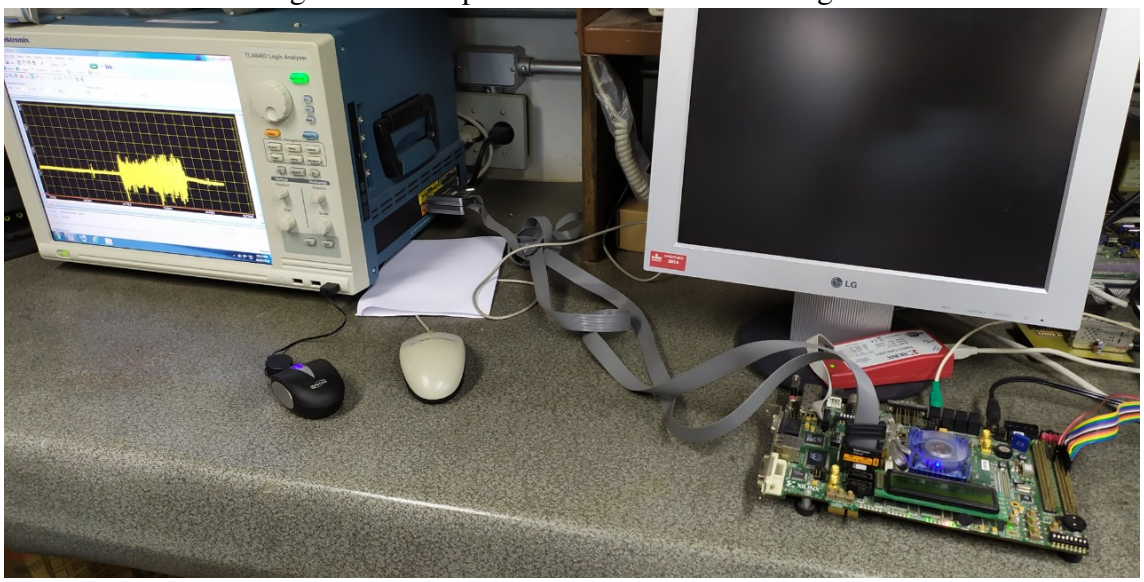
Uma vez que o sinal PDM corresponde a uma senoide de 7,5 kHz, é esperado um sinal PCM de saída com a mesma frequência. Sabendo que a frequência de amsotragem é de 48 kHz, em um segundo o sinal de saída tem 48000 amostras e 7500 ciclos. Logo, existe uma média de 6,4 amostras por ciclo. Sabendo que cada ciclo tem um pico, a distância entre dois picos quaisquer é de seis ou sete amostras, de modo que a média dessa distância seja

6,4 amostras. O resultado observado na simulação é portanto condizente com o esperado.

4.1.3 Resultados da implementação

Uma vez o transcodificador PDM-PCM já simulado em MATLAB e em ISE Simulator, o sistema foi implementado na FPGA. Para que se pudesse testar o dispositivo separadamente do sistema de comunicação USB, dois métodos foram utilizados. No primeiro teste, a placa Xilinx XUPV5 foi conectada ao analisador lógico TLA6402 pelo conector Mictor [Tektronix (2013)]. Os sinais de interesse foram direcionados no código VHDL para as saídas da FPGA conectadas ao Mictor. O setup de teste descrito pode ser visto na Figura 49.

Figura 49: Setup de teste com analisador lógico.



Fonte: Autor.

No código VHDL, foi estabelecido que dos pinos do conector Mictor, 16 seriam utilizados para o sinal PCM de 16-bits gerados a partir do sinal PDM vindo de um dos microfones digitais. Agrupando esses 16 sinais como constituintes de um número inteiro no ambiente do analisador lógico, foram gerados gráficos de aquisição que permitem a visualização do sinal PCM.

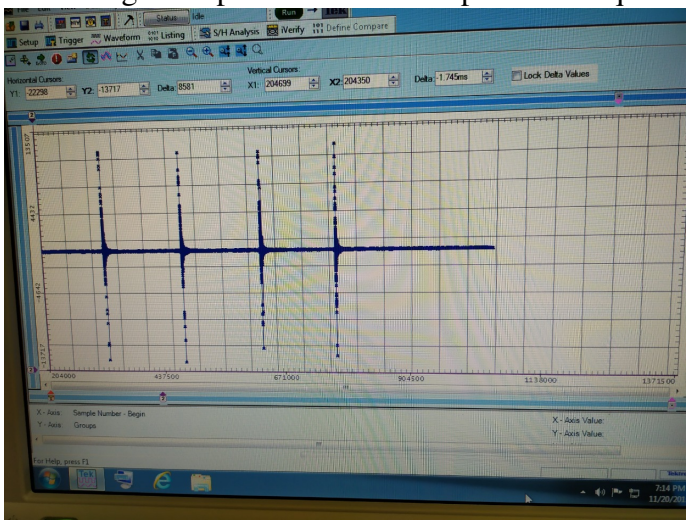
Definido o tempo de aquisição como 10 segundos, diferentes tipos de sons foram gerados para verificar se o transcodificador PDM-PCM gera seu sinal de saída de forma satisfatória. As Figuras 50, 51 e 52 mostram os resultados dessas aquisições.

Durante a aquisição realizada na Figura 50, foram feitos quatro estalos próximos ao microfone. O que se observa na tela do analisador lógico são os picos correspondentes, precisamente o que se esperaria observar no sinal de áudio.

A Figura 51 mostra o resultado de uma aquisição realizada enquanto alguém falava próximo ao microfone. No segmento inicial, observa-se o momento em que a voz começa. Em seguida, observam-se as variações de amplitude comumente observadas durante a fala de uma pessoa.

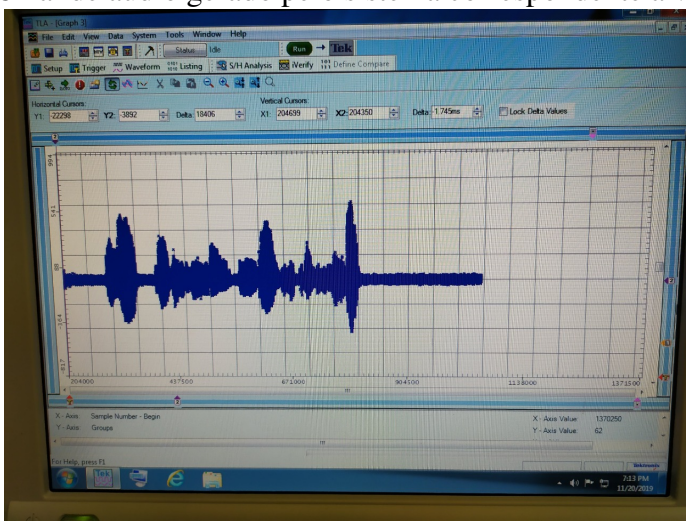
Durante o terceiro teste, foi gerado um tom puramente senoidal de frequência de 1kHz próximo ao microfone. Aproximando-se o sinal, pode-se observar na Figura 52 que o resultado da aquisição é senoidal. Além disso, através do posicionamento de dois cursores em dois picos adjacentes da senoide, observou-se que o período é de 1 ms, conforme

Figura 50: Sinal de áudio gerado pelo sistema correspondente a quatro estalos de dedo.



Fonte: Autor.

Figura 51: Sinal de áudio gerado pelo sistema correspondente a voz humana.



Fonte: Autor.

esperado.

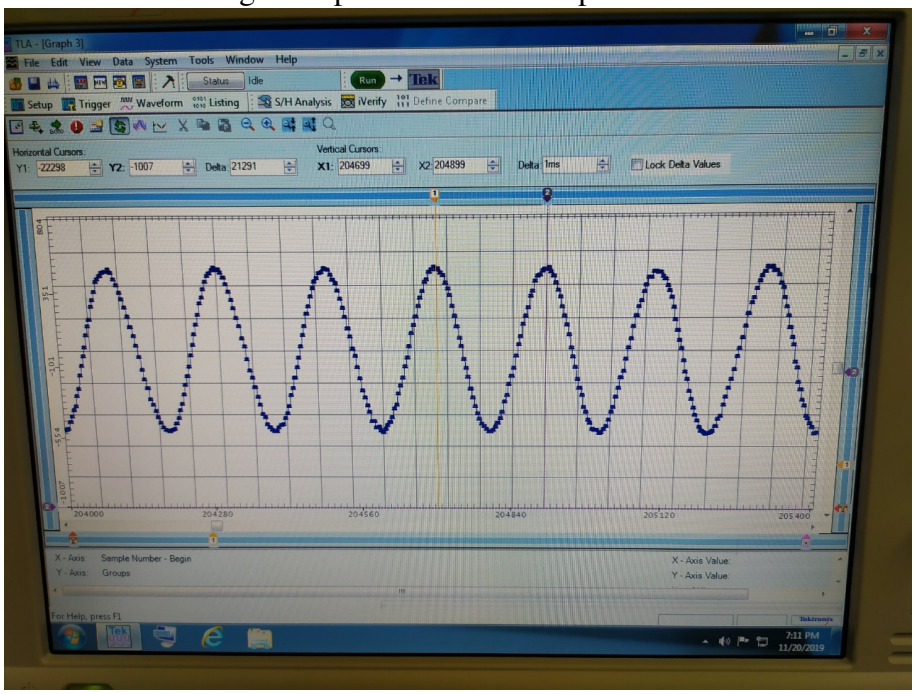
Esses testes realizados demonstram que o bloco transcodificador PDM-PCM obteve êxito na recriação do sinal de áudio. O primeiro teste demonstrou a rápida responsividade dos filtros com sinais de variação brusca. O segundo demonstrou a criação de envoltórias de sinal compatíveis com sinal de voz. Já o terceiro demonstrou a obtenção de um sinal com a frequência esperada e amplitude constante, e aspecto da onda condizente com um tom puramente senoidal.

O segundo método de teste utilizado permitiu uma avaliação subjetiva da qualidade do áudio. Foi utilizado o codec AC97 presente na placa XUPV5 [Xilinx (2011)] para reproduzir o áudio gerado em um alto falante externo.

Para esse teste, o alto falante foi conectado no conector de saída do codec AC97, conforma e Figura 53.

Durante o teste, foram gerados sons próximos aos microfones, e o sinal PCM de saída do transcodificador foi enviado aos alto-falantes. Observou-se que o som reproduzido

Figura 52: Sinal de áudio gerado pelo sistema correspondente a uma senoide de 1kHz.



Fonte: Autor.

correspondia ao som gerado próximo do microfone.

No entanto, observou-se a existência de uma quantidade considerável de ruído que se adicionou ao sinal de entrada. Esse ruído pode ter sido causado pelo transcodificador PDM-PCM, mas também pode ter outras fontes, como os microfones, o circuito ao qual eles estão acoplados, ou o alto-falante utilizado.

Observou-se também que parte desse ruído variou conforme o microfone selecionado. Uma vez que os filtros utilizados para cada microfone foram os mesmos, essa variabilidade na perda de qualidade é causada por imperfeições na placa de microfones. Além dos microfones, outro elemento que pode gerar ruído é a transmissão do sinal da placa de microfones para a FPGA. Essa transmissão é feita por cabos paralelos não blindados, que não são adequados para transmissão de sinais na faixa de frequência do PDM (3,84 MHz). A interação entre a irradiação desses sinais pode gerar *crosstalk*, o que causa interferência no sinal e se traduz pelo ruído escutado durante esse teste. A investigação desse problema é sugerida como um trabalho futuro.

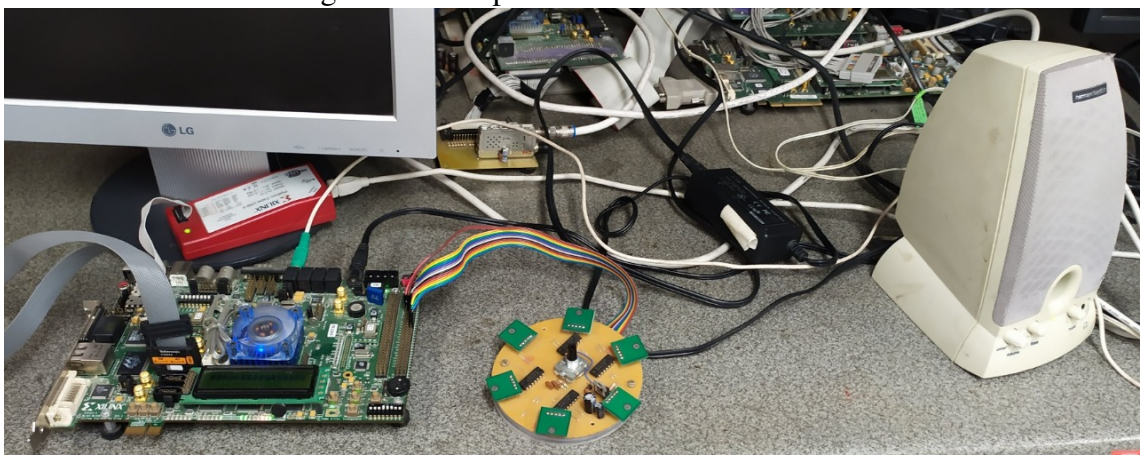
4.2 Sistema de comunicação

4.2.1 Resultados da interface HPI

Conforme explicado nos capítulos anteriores, o sistema de comunicação se baseia no controlador USB Cypress *Ez-Host*, sendo a comunicação entre a FPGA e o *Ez-Host* realizada pela interface HPI. Portanto, os resultados referentes ao sistema de comunicação atestam que a comunicação é funcional em ambas interfaces HPI e USB.

O protocolo HPI é definido pelo controlador USB *Ez-Host* e tem um mestre externo ao controlador, que no sistema deste projeto é a FPGA. Para validar a comunicação da FPGA através do protocolo HPI, realizou-se a simulação do código VHDL. Foram simulados os seguintes sinais, que aparecem na mesma ordem nas figuras seguintes:

Figura 53: Setup de testes com alto-falante.



Fonte: Autor.

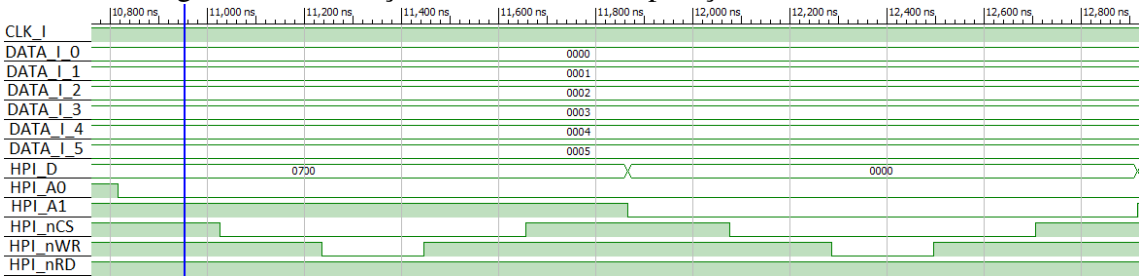
- CLK_I - *Clock* do sinal de áudio (48 kHz), entrada do bloco de interface HPI
- DATA_I_0 - Amostra de áudio do microfone 0 (16 bits)
- DATA_I_1 - Amostra de áudio do microfone 1 (16 bits)
- DATA_I_2 - Amostra de áudio do microfone 2 (16 bits)
- DATA_I_3 - Amostra de áudio do microfone 3 (16 bits)
- DATA_I_4 - Amostra de áudio do microfone 4 (16 bits)
- DATA_I_5 - Amostra de áudio do microfone 5 (16 bits)
- HPI_D (barramento de dados do HPI)
- HPI_A0
- HPI_A1
- HPI_nCS
- HPI_nWR
- HPI_nRD

A Figura 54 mostra a transição dos sinais do barramento HPI para a escrita de um dado na memória do Cypress *Ez-Host*. Os seis sinais indicados no esquema com valor de 0000 a 0005 correspondem aos sinais dos seis microfones, que no exemplo foram estabelecidos como constantes para teste. Observa-se que os sinais do barramento HPI de controle e de dados se comportam como o previsto no protocolo, conforme exposto nos capítulos anteriores, na Figura 33. Esse resultado se refere à escrita do valor do sinal do primeiro microfone na memória do Cypress *Ez-Host*, que está estabelecido como 0.

Na Figura 55, mostra-se a escrita dos dados dos seis microfones para um ciclo de *clock* do sinal PCM, com frequência de 48kHz.

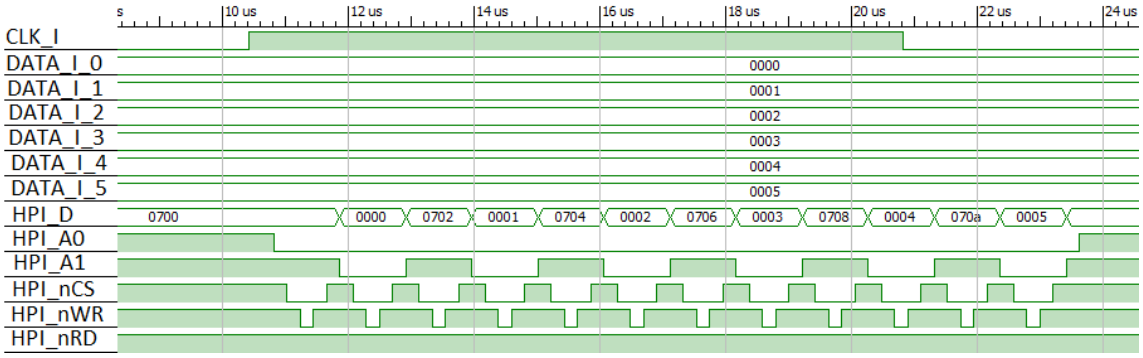
O sinal acima na Figura 55 é o *clock* do sinal PCM. Observa-se que a partir do momento em que um ciclo começa, ocorre a escrita dos seis dados em memória. Observa-se

Figura 54: Simulação VHDL de uma operação de escrita em HPI.



Fonte: Autor.

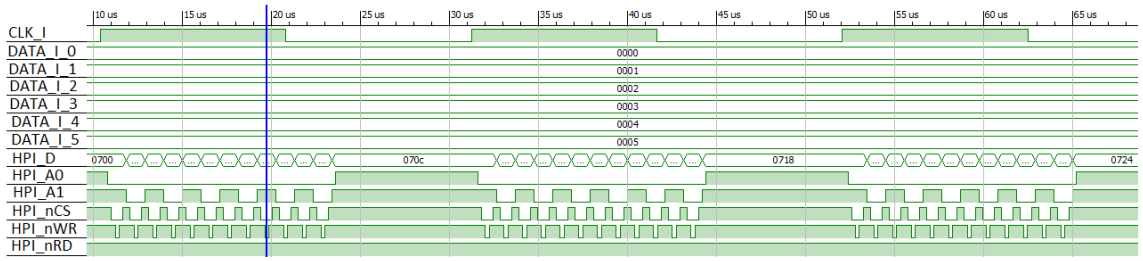
Figura 55: Simulação VHDL de seis operações de escrita HPI.



Fonte: Autor.

que sempre o endereço é incrementado a cada escrita de dado. Além disso, o período do *clock* PCM é mais que suficiente para escrever os dados dos seis microfones, havendo um período de espera ao final da escrita.

Figura 56: Simulação VHDL de diversos ciclos de escrita HPI.



Fonte: Autor.

Na Figura 56, observa-se o funcionamento da interface HPI em uma escala de tempo maior. Observa-se que ocorre a escrita em memória dos dados de todos os microfones uma vez a cada ciclo de *clock*. Observa-se também que o endereço da memória onde o dado é escrito aumenta a cada escrita de dados, o que é necessário para que os dados ocupem a parte da memória destinada ao envio via USB.

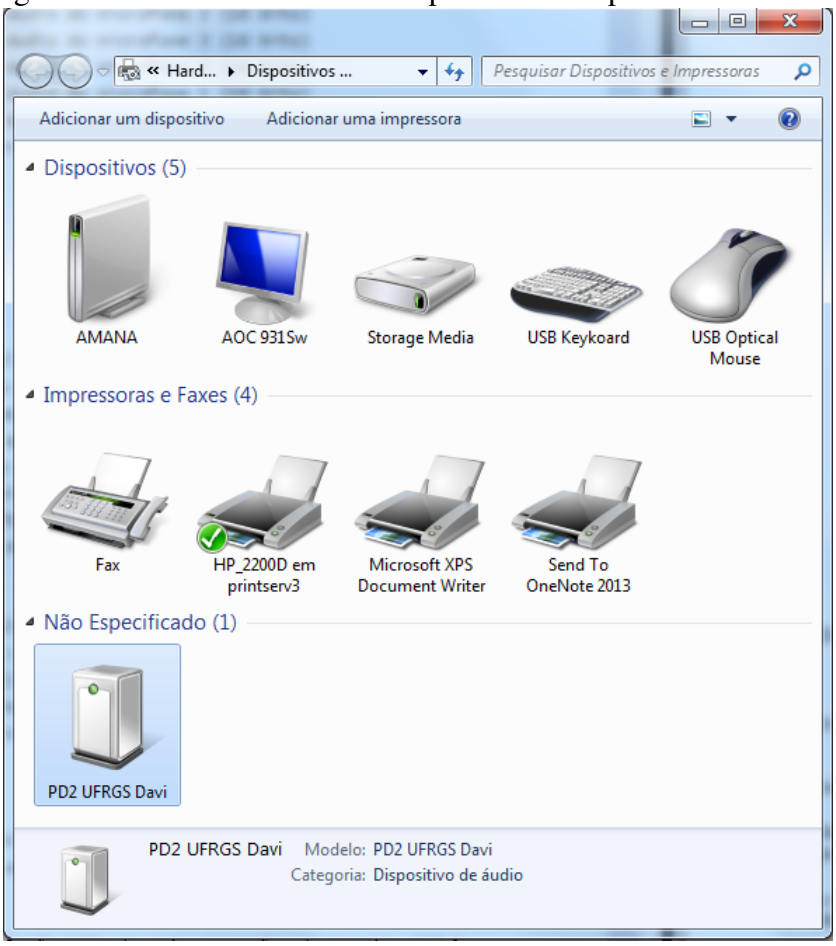
4.2.2 Resultados da enumeração USB

Da mesma forma que a interface HPI, foi verificado também que a interface USB está funcional. Para isso, a placa Xilinx XUPV5 foi conectada, através de sua porta USB, a um computador com Windows 7, e foram observados o processo de enumeração e o envio contínuo de dados. Na placa, o controlador *Ez-Host* é conectado diretamente à porta

USB da placa, de modo que no momento em que a placa é conectada ao computador, o controlador USB controla toda a comunicação.

Observou-se, no momento em que a FPGA foi conectada, que um novo dispositivo foi detectado pelo sistema. Na janela "Dispositivos e Impressoras", observou-se a presença de um dispositivo classificado como "Não especificado", conforme a Figura 57. O nome do dispositivo corresponde ao especificado no *firmware* do controlador USB, "PD2 UFRGS Davi".

Figura 57: Reconhecimento do dispositivo USB por *host* Windows.



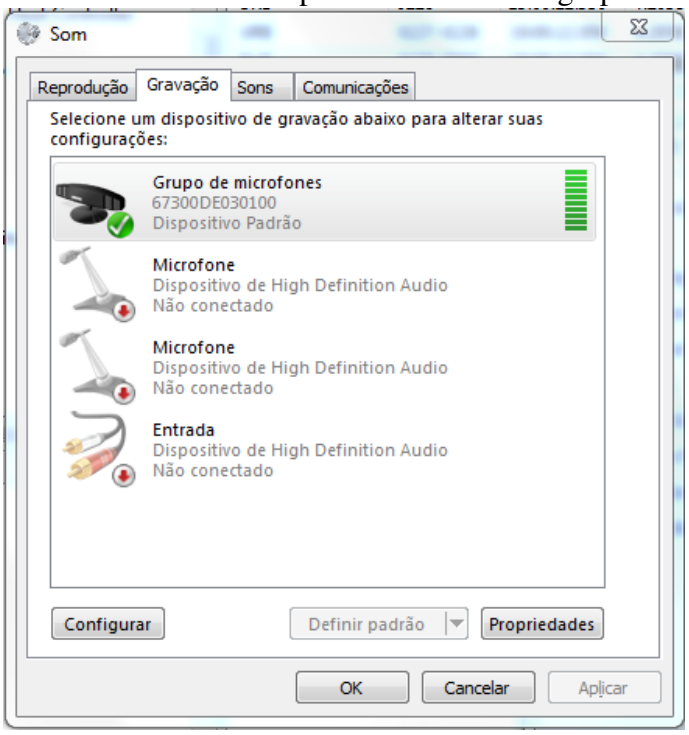
Fonte: Autor.

Além disso, foi observado o dispositivo na lista de dispositivos de gravação conectados ao computador, conforme a Figura 58. O dispositivo de gravação no topo da ista corresponde ao dispositivo do projeto. Observa-se que o Windows o reconhece como um Grupo de microfones, conforme indicado no descritor USB do dispositivo especificado no *firmware* do Cypress *Ez-Host*.

4.3 Resultados do sistema completo

Os últimos resultados obtidos no projeto tiveram o objetivo de observar o sinal de áudio recebido pelo *host* USB para todos os canais. Para isso, utilizou-se o *software* de edição de áudio Audacity. Primeiramente, para verificar unicamente a comunicação USB, o *Ez-Host* foi programado para enviar como sinal de áudio uma onda quadrada de frequência 1kHz para cada um dos seis canais. Em seguida, com o *software* Audacity, foi

Figura 58: Reconhecimento do dispositivo USB como grupo de microfones.



Fonte: Autor.

feita uma gravação do sinal enviado pelo dispositivo durante um intervalo de tempo para os seis canais. Por fim, fez-se um *zoom* temporal para observar a forma de onda do sinal recebido.

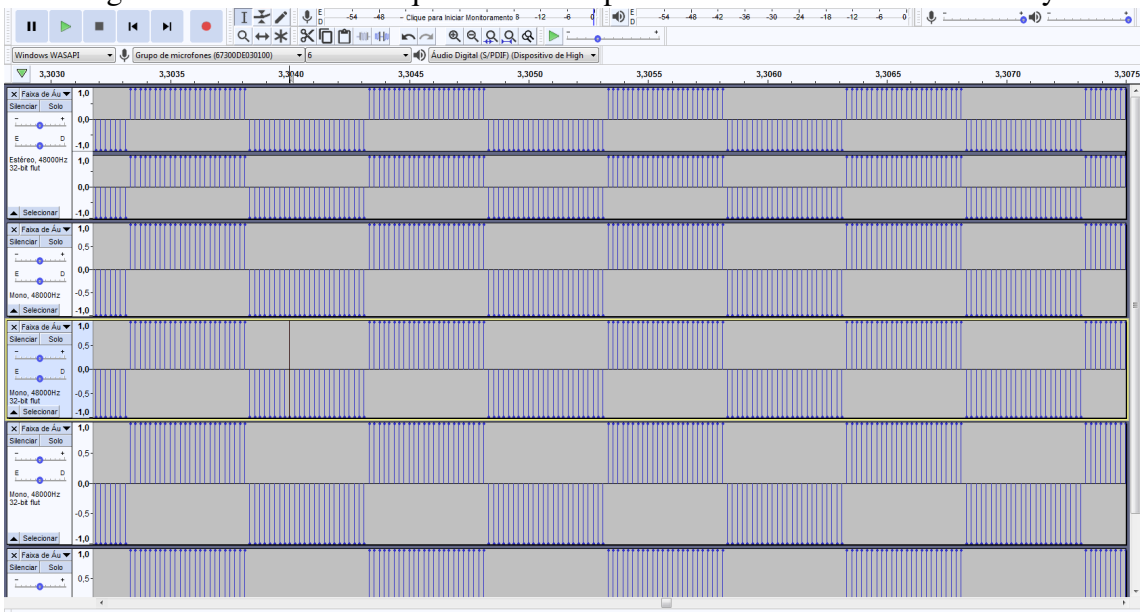
A Figura 59 mostra esse resultado. Observa-se que foram recebidos seis sinais distintos, e que cada sinal tem a forma de uma onda quadrada. Além disso, a partir da contagem na Figura do número de amostras, observa-se que cada ciclo possui 24 amostras positivas e 24 amostras negativas. Sabendo que a frequência de amostragem do sinal é 48 kHz, conclui-se que o sinal recebido tem 1 kHz. Esse resultando corresponde à frequência de envio programada no controlador USB.

O último teste realizado foi a aquisição do sinal dos microfones pelo *software* Audacity. Esse teste contemplou todas as etapas do projeto, isso é, a aquisição do sinal PDM do microfone, a transcodificação para PCM, e o envio do sinal via USB. Para isso, o controlador USB foi reprogramado com o seu *firmware* original, isso é, para o envio periódico dos dados do endereço de memória utilizado para a escrita dos dados do microfone via HPI pela FPGA.

A FPGA foi programada com um código VHDL no qual o sinal PCM de cada um dos seis microfones se conectava na entrada do componente de interface HPI. Com essa configuração, esperava-se receber os sinais dos seis microfones simultaneamente pelo *software* Audacity. No entanto, esse resultado não foi observado. O sinal recebido não era um sinal de áudio. A observação próxima das amostras do sinal recebido mostrou que a transmissão dos sinais PCM para o controlador USB via interface HPI não funcionou, uma vez que as amostras não correspondiam ao sinal gerado pelo transcodificador PDM-PCM. Ao invés disso, observou-se que as amostras recebidas via USB correspondiam provavelmente a valores desconhecidos escritos previamente durante a inicialização do controlador na seção de memória destinada para a transmissão HPI.

Em um segundo momento, foram feitas modificações no código VHDL, de modo que

Figura 59: Sinal de onda quadrada de dispositivo USB recebido no Audacity.



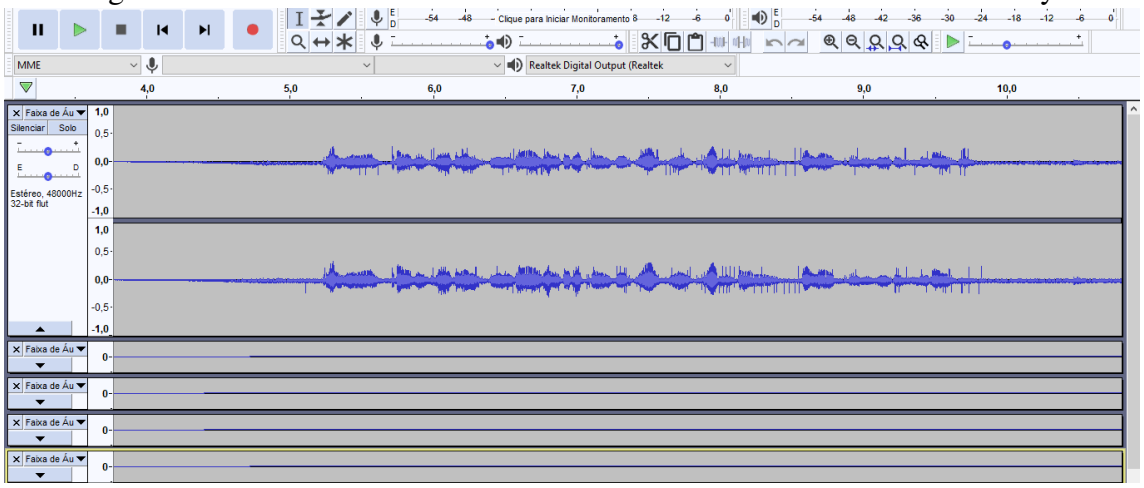
Fonte: Autor.

apenas os sinais PCM correspondentes a dois microfones fossem conectados ao componente de interface HPI. As outras entradas desse componente foram conectadas a um valor constante. Observou-se, através do *software* Audacity, que os sinais de áudio dos dois canais foi recuperado pelo *host* USB. Esse resultado pode ser visto na Figura 60. Durante a aquisição, foi gerado um sinal de voz próximo à placa de aquisição de áudio.

Observa-se na Figura 60 que os dois sinais apresentam aspecto semelhante. Essa característica é esperada, uma vez que os dois microfones ativos estavam captando o mesmo sinal de áudio em pontos próximos. Logo, a diferença entre os sinais captados deve ser unicamente devida à diferença de tempo de propagação do som para os pontos onde estão localizados os microfones, desconsideradas as imperfeições de captação e transmissão do sinal. Uma vez que essa distância é da ordem de centímetros e considerando a velocidade de propagação do som no ar, essa diferença deve ser imperceptível na escala de tempo de segundos em que o sinal foi captado.

Um aspecto negativo observado neste teste foi o ruído adicionado ao áudio. Do mesmo modo que no teste realizado com o alto-falante conectado à saída de áudio do kit Xilinx XUPV5, observou-se que o sinal de áudio recuperado pelo *software* Audacity apresentava quantidade considerável de ruído nos dois canais. O estudo das causas deste ruído é sugerido como um trabalho futuro.

Figura 60: Sinal de áudio de dois microfones recebido via USB no Audacity.



Fonte: Autor.

5 CONCLUSÕES

Este trabalho apresentou o desenvolvimento de um sistema com matriz de microfones PDM baseado em FPGA com comunicação USB. O sistema é composto de um módulo de transcodificação PDM-PCM descrito em VHDL e um módulo de comunicação USB, formado por uma interface HPI na FPGA e um controlador USB responsável por implementar o protocolo.

Os resultados finais observados foram satisfatórios para os dois módulos do sistema. O módulo de transcodificação PDM-PCM permitiu a aquisição dos dados de áudio provenientes do microfone, que foram escutados através da interface AC97 da placa e observados com um analisador lógico. Já o módulo de comunicação USB permitiu a recepção desses dados em um *host* Windows para dois canais simultaneamente.

Os principais problemas encontrados no projeto foram o ruído adicionado ao sinal de áudio e a falha na transmissão dos seis canais via USB. No entanto, apesar das imperfeições encontradas na solução implementada, os resultados obtidos demonstram que o sistema é funcional tanto na geração de sinais de áudio quanto na sua transmissão via USB.

Trabalhos futuros a partir desse projeto poderiam ser realizados. Objetivos destes trabalhos poderiam ser o aprimoramento tanto da qualidade do áudio recuperado quanto da comunicação USB. Além disso, poderiam ser realizadas tentativas de aplicar este dispositivo em sistemas de localização de emissores ou de aprimoramento de sinais de fala, conforme mencionado no capítulo de introdução.

Os resultados do teste realizado com o codec AC97 do kit Xilinx XUPV5 mostraram que a qualidade do sinal de áudio varia com o microfone selecionado. Uma vez que o módulo de transcodificação PDM-PCM implementado em VHDL faz exatamente as mesmas operações nos sinais dos seis microfones, estes resultados sugerem que existe variabilidade nos sinais de entrada. Portanto, um estudo poderia ser realizado com a placa de aquisição de áudio para observar se o sinal PDM gerado pelos microfones apresentam a relação sinal ruído especificada pelos seus fabricantes, ou então se existe ruído adicionado ao sinal na transmissão entre os microfones e a FPGA. Uma vez encontrada a fonte de variabilidade, a solução deste problema levaria a um aprimoramento da qualidade do áudio e diminuição da variabilidade da qualidade para os diferentes microfones.

Outros estudos poderiam ser realizados sobre os filtros utilizados no módulo transcodificador PDM-PCM. Esses estudos poderiam ser realizados nas simulações MATLAB, conduzidos de forma a descobrir as causas dos lóbulos observados no sinal de saída no domínio frequência. É possível que o reprojeto dos coeficientes do filtro *halfband* decimador possa atenuar o ruído percebido no sinal de áudio.

Além disso, outra possibilidade seria trabalhar sobre a comunicação USB e verificar as causas das falhas de transmissão dos sinais dos seis canais. Uma linha de investigação

seria descobrir se o problema se encontra na transmissão HPI ou no protocolo USB em si. A resolução de tal problema poderia até mesmo abrir a possibilidade da implementação do sistema para matrizes de microfones com mais canais.

REFERÊNCIAS

Aguirre, P. C. C. d. (2014). Projeto e análise de moduladores sigma-delta em tempo contínuo aplicados à conversão ad. Master's thesis, Universidade Federal do Rio Grande do Sul.

Ashour, G., Brackenridge, B., Tirosh, O., Todd, C., Zimmermann, R., and Knapen, G. (1998a). *Universal Serial Bus Device Class Definition for Audio Data Formats*.

Ashour, G., Brackenridge, B., Tirosh, O., Todd, C., Zimmermann, R., and Knapen, G. (1998b). *Universal Serial Bus Device Class Definition for Audio Devices*.

Ashour, G., Brackenridge, B., Tirosh, O., Todd, C., Zimmermann, R., and Knapen, G. (1998c). *Universal Serial Bus Device Class Definition for Terminal Types*.

Axelson, J. (2009). *USB Complete: The Developer's Guide*. Lakeview Research LLC, 4 edition.

B. Hogenauer, E. (1981). An economical class of digital filters for decimation and interpolation. *IEEE Transactions on Acoustics, Speech, and Signal Processing*.

Carlson, A. B. (2002). *Communication Systems*. McGraw-Hill, 4 edition.

Cirrus (2015a). *Configuring the WM7216 WM7236 Always-On Microphones*. Cirrus Logic.

Cirrus (2015b). *Interfacing WM72xx Digital Microphones*. Cirrus Logic.

Cirrus (2016). *Low Power Bottom Port Digital Silicon Microphone WM7236E*. Cirrus Logic.

Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips (2000). *Universal Serial Bus Specification*.

Cypress (2003). *EZ-Host^{TM} Programmable Embedded USB Host/Peripheral Controller*. Cypress Semiconductors.

Cypress (2011). *Cypress CyUsb.sys Programmer's Reference*.

Ganji, B. A. and Majlis, B. Y. (2004). Condenser microphone performance simulation using equivalent circuit method. *IEEE International Conference on Semiconductor Electronics*.

Gareta, A. A. (2007). *A multi-microphone approach to speech processing in a smart-room environment*. PhD thesis, Universitat Politècnica de Catalunya.

Hegde, N. (2010). *Seamlessly Interfacing MEMS Microphones with Blackfin® Processors*. Analog Devices.

Hoffmann, G. A. (2002). Study of the audio coding algorithm of the mpeg-4 aac standard and comparison among implementations of modules of the algorithm. Master's thesis, Universidade Federal do Rio Grande do Sul.

Hyde, J. (2003). *USB Multi-Role Device Design By Example*. Cypress Semiconductors.

Kite, T. (2012). Understanding pdm digital audio.

Kumar, N. (2016). *Optimal Design of FIR IIR Filters by using Evolutionary Algorithms*. Lambert Academic Publishing.

Lewis, J. (2012). *Common Inter-IC Digital Interfaces for Audio Data Transfer*. Analog Devices.

Lewis, J. (2013). *Analog and Digital MEMS Microphone Design Considerations*. Invensense.

Loibl, M., Walser, S., Klugbauer, J., Freiertag, G., and Siegel, C. (2016). Measurement of digital mems microphones. *GMA/ITG-Fachtagung Sensoren und Messsysteme*.

McClellan, J., Parks, T., and Rabiner, L. (1973). A computer program for designing optimum fir linear phase digital filters. *IEEE Transactions on Audio and Electroacoustics*.

Mei, G., Xu, R., Lao, D., Kwan, C., and Stanford, V. (2006). Real-time speaker verification with a microphone array. *Proceedings of the 2006 International Conference on Pervasive Systems Computing*.

Minzter, F. (1982). On half-band, third-band, and nth-band fir filters and their design. *IEEE Transactions on Acoustics, Speech, and Signal Processing*.

Narayana, M. M. (2011). *EZ-HOST/EZ-OTG: Standalone mode versus Co-processor mode*. Cypress Semiconductors.

Pavan, S., Schreirer, R., and C. Temes, G. (2017). *Understanding Sigma-Delta Converters*. IEEE Press, 2 edition.

Reitbauer, C., Rainer, H., Noisternig, M., Rettenbacher, B., and Graf, F. (2012). Micarray - a system for multichannel audio streaming over ethernet. *5th Congress of the Alps Adria Acoustics Association*.

Rosti, A.-V. (2004). 1-bit a/d and d/a converters. http://www.cs.tut.fi/sgn/arg/rosti/1-bit/, (acessado em 09/09/2019).

Self, D. (2010). *Audio Engineering Explained*. Focal Press, 1 edition.

Silicon, L. (2006). *USB Audio Class Tutorial*.

Smith, J. O. (2007). *Introduction to Digital Filters with Audio Applications*. W3K Publishing.

Smith, S. W. (1999). *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 2 edition.

Surf-Vhdl (2015). How to implement fir filter in vhdl. https://surf-vhdl.com/how-to-implement-fir-filter-in-vhdl/, (acessado em 09/09/2019).

Swanbeck, S. (2011). *Using the Host Port Interface (HPI) in Cypress OTG-Host*. Cypress Semiconductors.

Tektronix (2013). *Tektronix Logic Analyzer Family Quick Start User Manual*.

Tenoudji, F. C. (2012). *Analog and Digital Signal Analysis*. Springer.

Todorović, D., Salom, I., Čelebić, V., and Prezelj, J. (2017). Implementation and application of fpga platform with digital mems microphone array. *Proceedings of 4th International Conference on Electrical, Electronics and Computing Engineering*.

Xilinx (2008). *ML505/6/7 Block Diagram*.

Xilinx (2011). *ML505/ML506/ML507 Evaluation Platform User Guide*.

Xilinx (2012). *Virtex-5 FPGA User Guide*. Xilinx.

ANEXO A COEFICIENTES DO FILTRO *HALFBAND* DECIMADOR.

Tabela 22: Coeficientes do filtro *halfband* decimador.

Número	Decimal	Hexadecimal	Número	Decimal	Hexadecimal
0	0.0000	0x0000	21	0.3161	0x50EA
1	-0.0005	0xFFE2	22	0.0000	0x0000
2	0.0000	0x0000	23	-0.0995	0xE685
3	0.0014	0x005D	24	0.0000	0x0000
4	0.0000	0x0000	25	0.0532	0x0D9F
5	-0.0033	0xFF2A	26	0.0000	0x0000
6	0.0000	0x0000	27	-0.0318	0xF7DA
7	0.0064	0x01A5	28	0.0000	0x0000
8	0.0000	0x0000	29	0.0194	0x04F6
9	-0.0115	0xFD0F	30	0.0000	0x0000
10	0.0000	0x0000	31	-0.0115	0xFD0F
11	0.0194	0x04F6	32	0.0000	0x0000
12	0.0000	0x0000	33	0.0064	0x01A5
13	-0.0318	0xF7DA	34	0.0000	0x0000
14	0.0000	0x0000	35	-0.0033	0xFF2A
15	0.0532	0x0D9F	36	0.0000	0x0000
16	0.0000	0x0000	37	0.0014	0x005D
17	-0.0995	0xE685	38	0.0000	0x0000
18	0.0000	0x0000	39	-0.0005	0xFFE2
19	0.3161	0x50EA	40	0.0000	0x0000
20	0.5000	0x7FFF			

Fonte: Autor.