unesp UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
CAMPUS DE GUARATINGUETÁ

GUSTAVO MOSCARDO MULATINHO

RECONHECIMENTO DE VOZ PARA APLICAÇÕES EM AUTOMAÇÃO IMPLEMENTADO EM FPGA

Guaratinguetá
2011

GUSTAVO MOSCARDO MULATINHO

RECONHECIMENTO DE VOZ PARA APLICAÇÕES EM AUTOMAÇÃO IMPLEMENTADO EM FPGA

Trabalho de Graduação apresentado ao Conselho de Curso de Graduação em Engenharia Elétrica da Faculdade de Engenharia do Campus de Guaratinguetá, Universidade Estadual Paulista, como parte dos requisitos para obtenção do diploma de Graduação em Engenharia Elétrica.

Orientador: Prof. Dr. Leonardo Mesquita

Guaratinguetá
2011

Mulatinho, Gustavo Moscardo
M954r Reconhecimento de voz para aplicações em automação implementado em FPGA / Gustavo Moscardo Mulatinho – Guaratinguetá : [s.n], 2011.
117 f : il.
Bibliografia: f. 115-117

Trabalho de Graduação em Engenharia Elétrica – Universidade Estadual Paulista, Faculdade de Engenharia de Guaratinguetá, 2011.
Orientador: Prof. Dr. Leonardo Mesquita

1. Reconhecimento automático da voz 2. Automação 3. VHDL (Linguagem descritiva de hardware) I. Título

CDU 621.381

unesp UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
CAMPUS DE GUARATINGUETÁ

RECONHECIMENTO DE VOZ PARA APLICAÇÕES EM AUTOMAÇÃO IMPLEMENTADO EM FPGA

GUSTAVO MOSCARDO MULATINHO

ESTE TRABALHO DE GRADUAÇÃO FOI JULGADO ADEQUADO COMO PARTE DO REQUISITO PARA OBTENÇÃO DO DIPLOMA DE
"GRADUADO EM ENGENHARIA ELÉTRICA"

APROVADO EM SUA FORMA FINAL PELO CONSELHO DE CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Prof.Dr. SAMUEL EUZÉDICE DE LUCENA
Coordenador

BANCA EXAMINADORA:

Prof. Dr. LEONARDO MESQUITA
Orientador/UNESP-FEG

Prof. Dr. SAMUEL EUZÉDICE DE LUCENA
UNESP-FEG

Profa. Dra. PALOMA MARIA SILVA ROCHA RIZOL
UNESP-FEG

Novembro de 2011

DADOS CURRICULARES

GUSTAVO MOSCARDO MULATINHO

NASCIMENTO	10.12.1988 – SÃO PAULO / SP
FILIAÇÃO	Jorge Mulatinho Yone Moscardo Mulatinho
2007/2011	Curso de Graduação Engenharia Elétrica – Universidade Estadual Paulista – Faculdade de Engenharia campus Guaratinguetá

à minha família, por todo o apoio em toda a minha vida e amor incondicional, sem ela jamais chegaria onde estou nem seria quem sou.

AGRADECIMENTOS

Agradeço primeiramente a Deus, sem Ele jamais teria conseguido passar por todos os obstáculos encontrados na minha vida até aqui, agradeço pela força, paciência e fé que tem me dado.

aos meus pais, irmã e cunhado, por sempre me incentivarem a nunca desistir, por me apoiarem nos momentos mais difíceis pelos quais passei e por sempre estarem dispostos a estender a mão quando mais precisei.

ao meu orientador Prof. Dr. Leonardo Mesquita, por ter acreditado na minha capacidade para realizar este trabalho.

aos professores que fizeram parte da minha graduação em especial ao Prof. Dr. Samuel Euzédice de Lucena e Profa. Dra. Paloma Maria Silva Rocha Rizol, pelas conversas e experiências profissionais vividas passadas para mim.

aos meus amigos que fizeram parte da minha vida, por todas as risadas, dificuldades nos estudos pelas quais passamos juntos, sempre apoiando uns aos outros.

“Que os vossos esforços desafiem as impossibilidades, lembrai-vos de que as grandes coisas do homem foram conquistadas do que parecia impossível.”

Charles Chaplin

MULATINHO, G. M. **Reconhecimento de voz para aplicações em automação implementado em FPGA:** projeto e síntese de um sistema de reconhecimento automático de voz de palavras isoladas independente do locutor, utilizando Modelos Escondidos de Markov**.** 2011. Trabalho de Guaduação em Engenharia Elétrica – Faculdade de Engenharia do Campus de Guaratinguetá, Universidade Estadual Paulista, Guaratinguetá, 2011.

RESUMO

Muitos são os filmes de ficção científica em que são utilizadas máquinas capazes de dialogar com os seres humanos. Porém, o homem ainda está longe de chegar em tais máquinas, como o personagem C3PO do filme *Star Wars*. Durante as últimas seis décadas muito se têm investido nos estudos de reconhecimento automático de voz, surgindo ao longo desses anos diversas técnicas que podem ser utilizadas por ambas as aplicações de *software* e *hardware*. Diversos são os tipos de sistemas de reconhecimento automático de voz, dentre os quais o utilizado para este trabalho é o sistema de palavras isoladas independentes do locutor, utilizando Modelos Escondidos de Markov como técnica de reconhecimento da palavra. Este trabalho tem por finalidade projetar e sintetizar as duas primeiras etapas de um sistema de reconhecimento de voz, sendo tais etapas: a aquisição do sinal de voz e o pré-processamento do mesmo. Sendo estas etapas desenvolvidas em um componente reprogramável denominado FPGA, utilizando linguagem de programação de hardware VHDL, tendo em vista o alto desempenho que este componente pode proporcionar e a flexibilidade da linguagem. Neste trabalho é apresentado todo o conteúdo teórico de processamento digital de sinais, como a teoria de Transformadas Rápidas de Fourier e filtros digitais e também toda a teoria de reconhecimento de voz utilizando Modelos Escondidos de Markov e processador LPC. Também são apresentados todos os resultados obtidos por cada um dos blocos sintetizados e verificados em *hardware*.

PALAVRAS-CHAVE: Reconhecimento Automático de Voz, Automação, FPGA, VHDL, Modelos Escondidos de Markov.

MULATINHO, G. M. **Speech Recognition for Automation implemented in FPGA:** synthesis and project of an automatic speech recognition system of isolated words independents of the speaker, using Hidden Markov Models**.** 2011. Undergraduate Conclusion Work in Electral Engineering – Faculdade de Engenharia do Campus de Guaratinguetá, Universidade Estadual Paulista, Guaratinguetá, 2011.

ABSTRACT

In many movies of scientific fiction, machines were capable of speaking with humans. However mankind is still far away of getting those types of machines, like the famous character C3PO of Star Wars. During the last six decades the automatic speech recognition systems have been the target of many studies. Throughout these years many technics were developed to be used in applications of both software and hardware. There are many types of automatic speech recognition system, among which the one used in this work were the isolated word and independent of the speaker system, using Hidden Markov Models as the recognition system. The goals of this work is to project and synthesize the first two steps of the speech recognition system, the steps are: the speech signal acquisition and the pre-processing of the signal. Both steps were developed in a reprogrammable component named FPGA, using the VHDL hardware description language, owing to the high performance of this component and the flexibility of the language. In this work it is presented all the theory of digital signal processing, as Fast Fourier Transforms and digital filters and also all the theory of speech recognition using Hidden Markov Models and LPC processor. It is also presented all the results obtained for each one of the blocks synthesized e verified in hardware.

KEYWORDS: Automatic Speech Recognition, Automation, FPGA, VHDL, Hidden Markov Models

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

ANN – *Artificial Neural Network*

CLB - *Configurable Logic Block*

CODEC – Codificador Decodificador

CPLD - *Complex Programmable Logic Device*

CPU – *Central Processing Unit*

DFT – *Discrete Fourier Transform*

DTW - *Dynamic Time Warping*

DVD - *Digital Video Disc*

E^2PROM – *Electrically Erasable Programmable Read-Only Memory*

EPROM – *Electrical Programmable Read-Only Memory*

FFT – *Fast Fourier Transform*

FIR – *Finite Impulse Response*

FPGA – *Field Programmable Gate Array*

HMM – *Hidden Markov Model*

IDFT – *Inverse Discrete Fourier Transform*

IEEE – *Institute of Electrical and Electronic Engineer*

IIR – *Infinite Impulse Response*

IP – *Intelectual Property*

LAB - *Logic Array Blocks*

LE – *Logic Element*

LPC – *Linear Predictive Coding*

LSB – *Least Significant Bit*

LSI – *Linear Shift Invariant*

LUT - *Look-Up Table*

MFCC – *Mel Frequency Cesptral Coding*

PAL - *Programmable Array Logic*

PLA - *Programmable Logic Array*

PLD – *Programmable Logic Device*

PLL – *Phase Locked Loop*

PROM – *Programmable Read-Only Memory*

RAV – Reconhecimento Automático de Voz

SDRAM – *Static Dynamic Random Access Memory*

SOC – *System On a Chip*

TFTD – Transformada de Fourier de Tempo Discreto

USB – *Universal Serial Bus*

VHDL – *VHSIC Hardware Description Language*

VHSIC - *Very High Speed Integrated Circuit*

SUMÁRIO

1-INTRODUÇÃO

1.1- Histórico do Reconhecimento de Voz

Este item foi fundamentado na referência RABINER, L., 1993. Diversos filmes de ficção científica da década de setenta como "2001 – Uma odisseia no espaço" e "Guerra nas estrelas", por exemplo, já nos mostravam máquinas capazes de conversar com os seres humanos, receber ordens e executar tarefas apenas através de comandos de voz. Quem não se lembra no famoso R2D2 fiel companheiro de Luke Skywalker, personagem de "Guerra nas Estrelas"? Porém ainda estamos longe de conseguir este tipo de sistema de reconhecimento, mas muitos avanços já foram feitos desde então.

O reconhecimento automático de voz tem sido estudado por mais de cinco décadas, nas quais diversas técnicas de reconhecimento de padrões foram desenvolvidas junto com novas tecnologias que estavam surgindo.

Os primeiros estudos começaram nos anos 1950's quando diversos pesquisadores tentaram explorar as ideias fundamentais da acústica aplicada à fonética. Em 1952, no *Bell Laboratories*, Davis, Biddulph e Balashek criaram um sistema de reconhecimento de dígito isolado para um único locutor. O sistema consistia em mensurar o espectro de ressonância nas regiões das vogais em cada número dito pelo locutor. Em 1959 na "*University College in England*", os pesquisadores Fry e Denes tentaram construir um reconhecedor de fonemas para reconhecer quatro vogais e nove consoantes. Tais cientistas utilizaram um analisador de espectro e um comparador de padrões para fazer o reconhecimento.

Nos anos 1960's diversas ideias fundamentais para o reconhecimento de voz foram publicadas. Em 1962, no Japão, Sakai e Doshita da Universidade de Kyoto, construíram um *hardware* para reconhecimento de fonemas, o qual consistia em um hardware que segmentava a fala, usado em conjunto com um analisador de cruzamento de zero, em diferentes regiões da fala de entrada, para se realizar o reconhecimento da voz. Uma outra tentativa de construir um reconhecedor de dígitos em hardware, foi de Nagata e seus colegas no "*NEC Laboratories*" em 1963. Essa tentativa foi talvez a mais notável e produtiva para o reconhecimento de voz no Japão nessa década. Uma pesquisa muito importante no final dos anos 1960's foram os esforços de Martin e seus colegas do "*RCA Laboratories*", para desenvolver soluções realísticas para os problemas associados com a não-uniformidade da escala de tempo nos eventos de voz. Martin desenvolveu um conjunto de métodos elementares para a normalização do tempo, baseado na capacidade de fielmente detectar o início e o fim da uma fala. Martin além de desen-

volver o método fundou uma das primeiras empresas, a "*Threshold Technology*", a qual criava e vendia soluções em reconhecimento automático de voz.

Nos anos 1970's as pesquisas em reconhecimento automático de voz adquiriram marcos extremamente importantes. Primeiro na área de reconhecimento de palavras isoladas, as quais se tornaram tecnologias viáveis e úteis, graças aos estudos fundamentais, nesta área, de Velichko e Zagoruyko na Rússia, Sakoe e Chiba no Japão e Itakura nos Estados Unidos. Os estudos russos ajudaram no avanço de ideias de reconhecimento de padrões aplicados ao reconhecimento de voz. Os estudos japoneses nos mostraram como métodos dinâmicos de programação poderiam ser aplicados com sucesso, e a pesquisa de Itakura nos apresentou a fundamentação dos LPC (*linear predictive coding*).

Assim como os estudos em sistemas de reconhecimento de palavras isoladas foram focados nos anos 1970's, os estudos de sistemas de reconhecimento de palavras conectadas foram focados nos anos 1980's. Aqui o principal objetivo era criar sistemas robustos capazes de reconhecer fluentemente frases inteiras, baseados na comparação de padrões concatenados de palavras isoladas. O reconhecimento de voz nos anos 1980's foi marcado por uma troca de abordagem baseada em modelos, para uma abordagem de modelagem estatística. O principal método de estudo nesta época foi o modelo escondido de Markov (*HMM – Hidden Markov Model*). A metodologia das HMM's já era muito bem entendida e conhecida, em alguns laboratórios, mas seus métodos e teoria não foram amplamente publicados até metade desta década, quando foram largamente utilizados em quase todos os laboratórios de reconhecimento de voz no mundo inteiro.

Outra tecnologia que foi reintroduzida no fim dos anos 1980's foi a aplicação das redes neurais artificiais (*ANN – Artificial Neural Networks*) para o reconhecimento de voz. As ANN's já tinham sido criadas nos anos 1950's porem não se mostraram como úteis devido à diversos problemas práticos. Porém nos anos 1980's as ANN's foram profundamente estudas conhecendo-se assim suas limitações e forças devido à tecnologia da época. Diversos novos métodos de implementação de sistemas de reconhecimento automático de voz foram desenvolvidos no final desta década e na década seguinte.

1.2 – O corpo humano e o sistema de voz

Para que possamos criar máquinas capazes de entender a fala humana, precisamos inicialmente entender o processo de produção da voz, desde a sua formulação, pelo locutor, até a sua percepção pelo receptor da mensagem.

O processo de fala se inicia quando o locutor deseja transmitir uma mensagem à outra pessoa. Para isso inicialmente o locutor formula, em sua mente, a mensagem a ser transmitida, depois essa mensagem é codificada na língua que o locutor quer falar, ou seja, a mensagem é transformada em fonemas, os quais em sequência conjunta formam as palavras a serem ditas. Após esta etapa o cérebro manda sinais neurológicos para os diversos músculos da boca, língua, pulmão e outros que serão detalhados mais adiante, para que usados de forma apropriada, possam produzir os diversos sons dos diferentes fonemas de cada língua a ser falada. As ondas sonoras então produzidas são transmitidas pelo ar até o receptor, começa-se aí o processo de percepção da fala.

As ondas sonoras propagadas pelo ar, que contém a informação a ser recebida, chegam até a orelha do receptor, neste órgão, mais precisamente na orelha interna, a vibração do ar faz com que a membrana auditiva, o tímpano, vibre e essa vibração é transformada em impulsos elétricos pelos músculos, ossos e células ali envolvidos. Esses impulsos são então transmitidos até o cérebro através dos nervos. No cérebro a linguagem é decodificada, de acordo com a memória do receptor, o qual irá interpretar cada palavra e cada fonema da mensagem transmitida e finalmente a mensagem é entendida pelo receptor. Com isso encerra-se todo o processo de produção e percepção da fala. A Figura 1.1 abaixo ilustra todo o processo acima descrito.

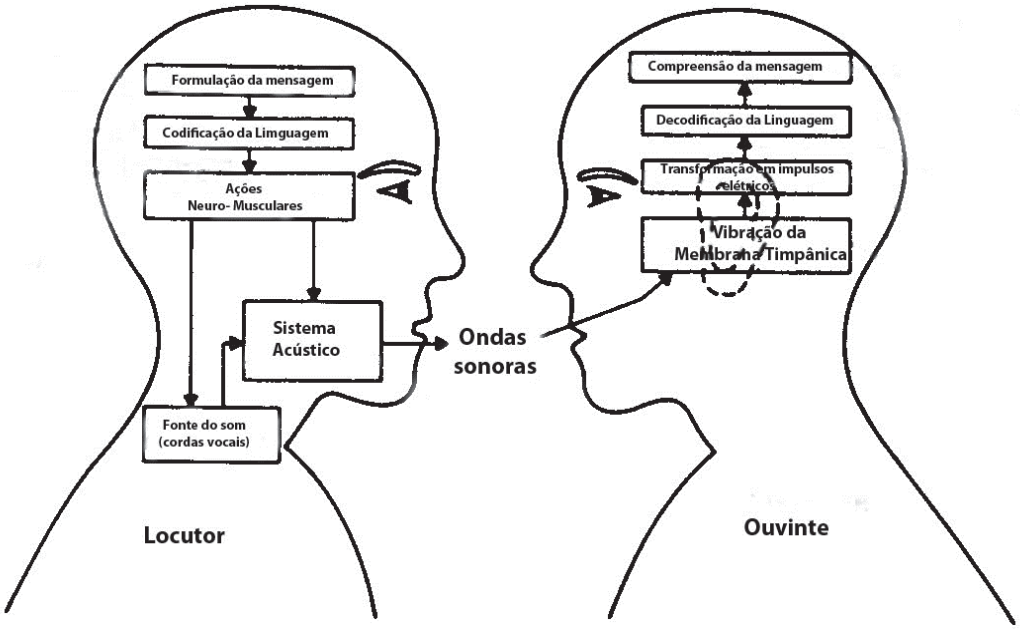


Figura 1.1: *Transmissão de uma mensagem. (RABINER, L.R.,1993)*

Também é pertinente entender a fisiologia do corpo humano envolvida no processo de fala, tanto na produção da mesma quanto na recepção. Analisaremos inicialmente os órgãos envolvidos na produção da fala. Como descrito anteriormente todo o processo se inicia no cérebro, a Figura 1.2 mostra as principais regiões do cérebro, onde a mensagem é formulada e codificada na língua a ser falada.

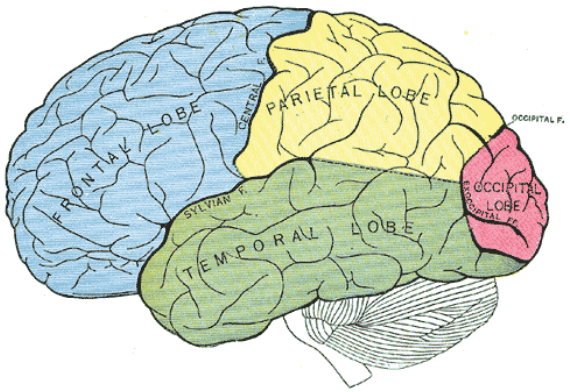


Figura 1.2: *Regiões do cérebro humano.*
(fonte: http://pt.wikipedia.org/wiki/Cérebro_humano)

O cérebro então manda um comando para que o diafragma contraia, pressionando assim o pulmão fazendo com que o ar suba pela traqueia. O ar passa pela abertura da prega vocal, localizada dentro da laringe, fazendo com que esta vibre, iniciando-se assim a produção

do som, o qual se propaga pela faringe. Dependendo de cada fonema que será necessário para a fala, por exemplo, o fonema "ãn" é nasal, portanto o véu palatino se abre para que o ar possa entrar na fossa nasal e sair pelo nariz. Em alguns fonemas o som é produzido tanto pela boca quanto pelo nariz. A língua é de extrema importância para os fonemas produzidos pela boca como, por exemplo, o fonema "tê", no qual a língua encosta no céu da boca, atrás dos dentes, bloqueando a passagem de ar por um instante e liberando-a impulsivamente. Tais órgãos envolvidos na boca (língua, mandíbula, dentes, véu palatino, lábios, etc.) são costumeiramente chamados de articuladores. Pode-se observar na Figura 1.3 a localização de cada um destes órgãos.

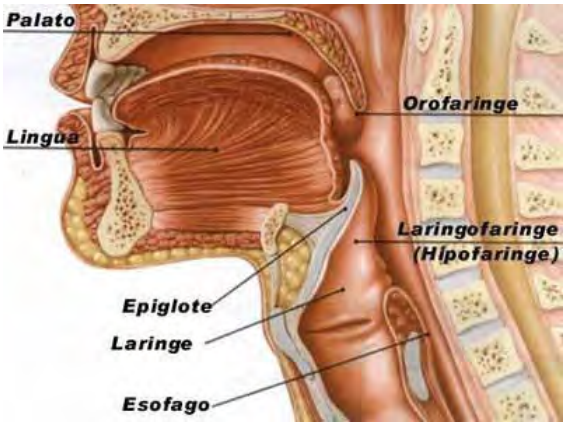


Figura 1.3: *Órgãos envolvidos na fala.*
(fonte: http://ocantodocanto.blogspot.com/2007/08/curiosidades.html)

O processo de recepção e percepção da fala se inicia com a chegada das ondas sonoras até a orelha externa e orelha interna do receptor. É nesta última que os sinais sonoros são transformados em impulsos nervosos e transmitidos ao cérebro. A Figura 1.4 mostra os detalhes da orelha interna (antigamente denominada ouvido).

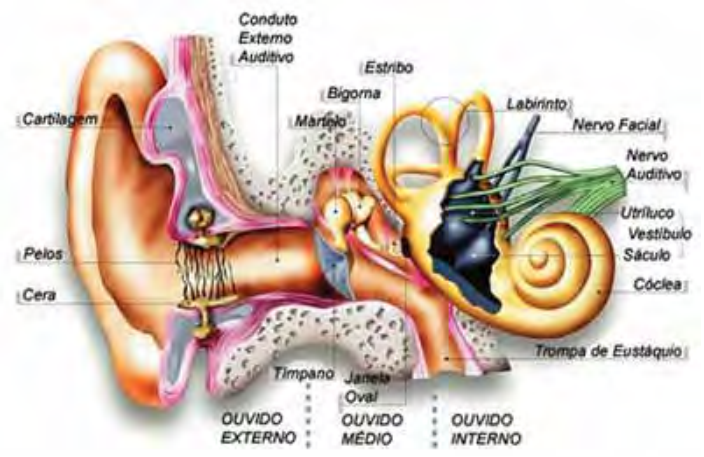


Figura 1.4: *Orelha externa e orelha interna.*
(fonte: http://fisiogerontologia.blogspot.com/2010/06/reabilitacao-vestibular-uma-revisao.html)

A orelha externa capta as ondas sonoras, as quais entram no conduto auditivo e fazem vibrar a membrana timpânica (o tímpano), esta por sua vez está mecanicamente ligada á três ossículos chamados: Martelo, Bigorna e Estribo. A vibração do tímpano é transferida para essa cadeia de ossículos que então transferem a vibração para a orelha interna. Esta última é composta pela cóclea e pelo labirinto. A vibração na orelha interna chega até as células ciliadas da cóclea que, por fim, é transformada em impulsos elétricos e enviada para o cérebro através do nervo auditivo. Este nervo conduz a informação até o Lobo Temporal que é responsável pela interpretação do sinal sonoro, e também a região de convergência dos lobos temporal, occipital e parietal, chamada de Área de Wernicke responsável pela compreensão da mensagem transmitida.

1.3 – Processamento de Voz.

Como apresentado de forma escrita no item 1.1, atualmente existem diversas técnicas capazes de realizar o reconhecimento automático de voz (RAV). Algumas das principais técnicas usadas para implementar sistemas de reconhecimento de voz em tecnologias atuais são: Alinhamento Temporal Dinâmico (*Dynamic Time Warping, DTW*), Modelos Escondidos de Markov (*Hidden Markov Model, HMM*) e a utilização de Redes Neurais Artificiais (*Artificial Neural Networks, ANN*). O reconhecimento automático de voz pode ser classificado da seguinte maneira:

1- *Reconhecimento de palavras isoladas:* sistema no qual o locutor tem que fazer uma pausa entre as palavras a serem pronunciadas. Este método oferece uma

grande chance de acerto, do algoritmo, de praticamente cem por cento para vocabulários pequenos e ambientes limpos (livres de interferências sonoras).

2- *Reconhecimento de palavras conectadas:* neste sistema a complexidade é maior que o precedente, onde as palavras são identificadas através de cada fonema que as compõem, permitindo assim o reconhecimento de sentenças sem pausas entre as palavras pronunciadas.
3- *Reconhecimento de fala contínua:* este sistema é o de maior complexidade e dificuldade, pois deve ser capaz de reconhecer a fala de forma natural tendo que lidar com problemas de fonemas parecidos e de coarticulação.

Quanto ao tipo de locutor os sistemas de reconhecimento automático de voz podem ser classificados em duas categorias:

1- *Dependentes do Locutor:* nos quais utilizam um banco de dados para reconhecer as palavras, ou seja, o sistema apenas reconhece a voz dos locutores para o qual foi treinado.
2- *Independentes do Locutor:* sistemas mais complexos, nos quais realiza o reconhecimento da fala de qualquer pessoa que as estejam pronunciando. Neste caso as chances de acerto são menores que o sistema anterior.

O sistema proposto para ser desenvolvido neste trabalho é de reconhecimento de palavras isoladas com pequeno vocabulário e independente do locutor.

Este trabalho tem como um de seus objetivos, desenvolver o sistema citado pra ser utilizado por pessoas com deficiência física, as quais poderão ter suas vidas facilitadas através de produtos que incorporem este sistema.

Tal sistema pode ser implementado de diversas formas, sendo a abordagem via *software* a mais comum no mundo, porém este tipo de abordagem tem diversas desvantagens como: serem altamente dependentes de microcomputadores ou de qualquer outro equipamento equivalente, fazendo assim com que o usuário fique restrito a estes tipos de equipamentos, perdendo então mobilidade e conforto. O desempenho desta abordagem fica também limitado pelo *hardware* do equipamento em que o *software* está instalado.

Neste trabalho será realizado o estudo de viabilidade de implementação do sistema de reconhecimento de voz em uma abordagem de *hardware*, a qual tem diversas vantagens em relação à abordagem anterior, tais como: maior velocidade de processamento de dados, maior

compactação, flexibilidade e mobilidade, implicando assim em um maior conforto para o usuário. O *hardware* a ser utilizado neste trabalho é baseado em um componente de lógica reconfigurável denominado FPGA.

Já alguns anos, vem-se desenvolvendo em diversos países, soluções para sistemas RAV em FPGA's, utilizando-se as mais diversas técnicas de reconhecimento de voz, para os diversos tipos de sistemas já citados.

A utilização de um SOC – *System On a Chip* baseados em FPGA – *Field Programmable Gate Arrays,* permite que se possa ter um alto desempenho em suas aplicações, devido à característica deste dispositivo de executar todas as suas tarefas de maneira concorrente, ou seja, paralela. Os fabricantes de CPU's para microcomputadores, recentemente viram que esta característica aumentava consideravelmente o desempenho dos mesmos, portanto através de múltiplos núcleos que operam simultaneamente eles puderam adquirir a principal característica da FPGA, o paralelismo, encontrado nos atuais microprocessadores. Porém esta característica não é encontrada nas CPU's de propósito geral, que são outro tipo de *hardware* bastante utilizado para a implementação de sistemas RAV, fazendo com que as FPGA's tenham uma grande vantagem de desempenho em relação a estes dispositivos.

1.4- Componente Reconfigurável

Durante as últimas décadas, circuitos integrados capazes de serem programados, apagados, e reprogramados, têm sido utilizados em diversas aplicações nos mais variados tipos de tecnologia existentes desde então. Tais circuitos conhecidos como dispositivos lógicos programáveis (*PLD – Programmable Logic Device*) são classificados como, componentes capazes de implementar uma grande variedade de funções lógicas, tanto combinacionais quanto sequenciais. As funções a serem implementadas são definidas pelo usuário, o qual utiliza uma ferramenta computacional de auxílio do projeto. Esse tipo de ferramenta geralmente é um *software*, e através de um computador o usuário pode definir as funções a serem programadas e realizar a programação do dispositivo. São diversas as vantagens que os PLD's têm em relação á outros dispositivos, dentre as quais pode-se citar: o alto desempenho, a facilidade de projeto, redução do tempo global de projeto, confiabilidade, flexibilidade e redução de custos.

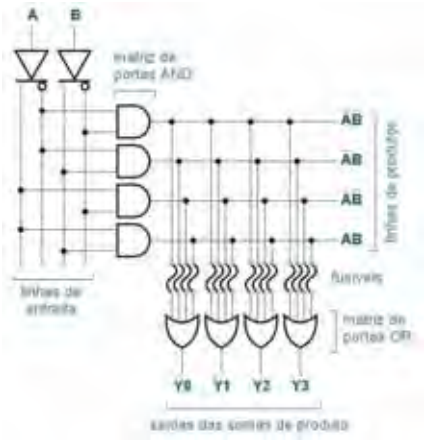


Figura 1.5: *Conceituação de PLD. (FILADELFO, F.R, 2003)*

A Figura 1.5 apresenta a ideia básica de um PLD, na qual com apenas as portas lógicas NOT, AND e OR, podemos realizar qualquer função booleana do tipo soma de produtos. Neste caso o usuário basta escolher qual arranjo quer utilizar (função booleana a ser implementada) e deste modo proceder a programação dos elos fusíveis necessários para a implementação da referida função. Atualmente as tecnologias dos elementos de programação são baseadas nos componentes anti-fusível, células SRAM, EPROM, EEPROM. Ao longo das últimas décadas, diversas foram também as arquiteturas de PLD's criadas. As arquiteturas mais básicas que surgiram inicialmente foram: a PROM - *Programmable Read Only Memory,* PLA – *Programmable Logic Array* e a PAL - *Programmable Array Logic*. Porém arquiteturas mais avançadas e complexas surgiram e são hoje objetos de muitos estudos e de muitas aplicações comerciais. As arquiteturas avançadas mais conhecidas são: CPLD – *Complex Programmable Logic Device* e a FPGA – *Field Programmable Gate Array*.

1.4.1- FPGA – Field Programmable Gate Array

Neste tópico será apresentada as características fundamentais de um circuito FPGA, de modo sucinto, maiores informações podem ser encontradas na referência FLOYD, T.L., 2007. Em uma FPGA têm-se três elementos básicos, os quais são o bloco lógico configurável (CLB), as interconexões e os blocos de entradas/ saídas (I/O). Os CLB's de uma FPGA são equiparáveis aos LAB's de um CPLD, porém são geralmente encontradas em maiores quantidades. Existe atualmente duas arquiteturas desenvolvidas para componentes FPGA sendo: (i) *fine graned* (granulação fina), na qual os CLB's são relativamente mais simples; (ii) *coarse graned* (granulação grossa), na qual os CLB's são maiores e mais complexos. Os blocos de I/O são responsáveis pela comunicação com o mundo externo da FPGA, e são selecionáveis

individualmente, podendo ser unidirecionais ou bidirecionais. Já as interconexões programáveis, realizam a interconexão entre os CLB's e os blocos de I/O. As FPGA's são reprogramáveis e utilizam tecnologia SRAM ou de antifusível nas conexões programáveis. A Figura 1.6 ilustra a estrutura básica de uma FPGA.

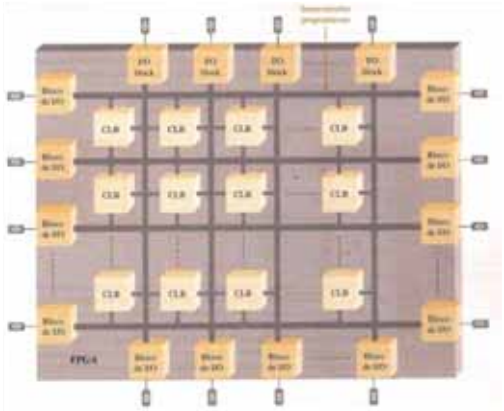
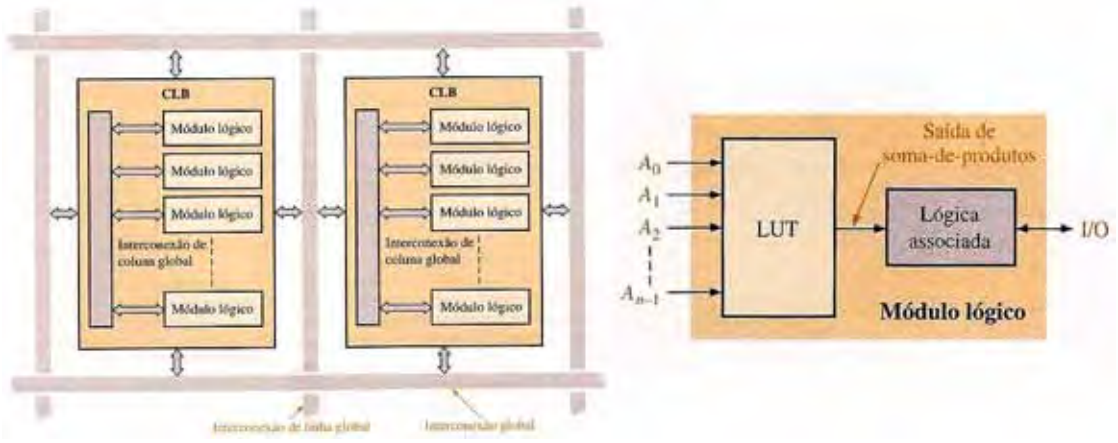


Figura 1.6: *Estrutura básica de uma FPGA. (FLOYD, T.L, 2007)*

Os CLB's são formados por múltiplos módulos lógicos menores e uma interconexão programável interna, a qual é utilizada para interconectar os módulos internos à um CLB. Os módulos lógicos por sua vez são constituídos por uma parte de lógica combinacional e uma parte de lógica registrada, assim pode-se configurar o CLB para apenas uma das lógicas ou uma combinação de ambas. A LUT presente no módulo lógico é uma memória programável usada para gerar funções lógicas combinacionais do tipo soma de produtos, já a lógica associada é formada de flip-flops que constituem a parte de lógica registrada do CLB. A Figura 1.7a mostra a constituição de um CLB e a Figura 1.7b mostra a estrutura interna de um módulo lógico.



(a) *(b)*

Figura 1.7: *(a) Estrutura interna de CLB; (b) Estrutura interna de um módulo lógico. (FLOYD, T.L, 2007)*

1.5- O sistema de reconhecimento automático de voz (RAV)

Durante o processo de reconhecimento de voz, diversas etapas devem ser realizadas, tais como: aquisição de voz, desenvolvimento da unidade de pré-processamento, parametrização da informação e desenvolvimento de uma rede de reconhecimento. Essa ideia básica do sistema RAV é ilustrado na Figura 1.8 em nível de diagrama de blocos.

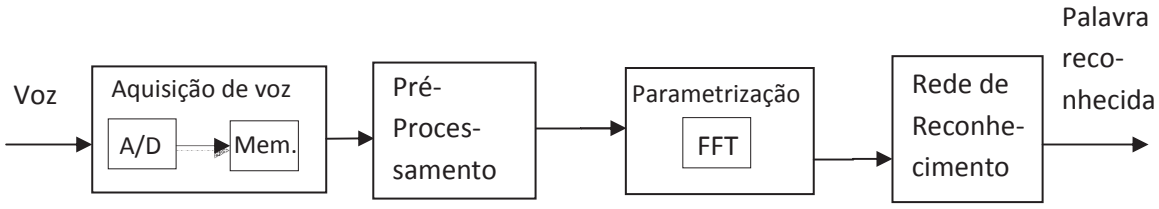


Figura 1.8: *Conceituação fundamental do sistema RAV.(O Autor)*

O primeiro passo do processo é adquirir a voz e armazená-la em uma memória. Para captar o som do locutor é utilizado um microfone, o qual transformará a energia das ondas sonoras em sinais elétricos. A aquisição da voz consiste em diversas etapas, como a amostragem no sinal, quantização do mesmo e codificação, sendo todas essas tarefas executadas pelo conversor analógico/digital. Após essa etapa, a voz adquirida é armazenada em uma memória, porém temos aqui a voz no domínio do tempo, na qual os sistemas que trabalham nesse domínio são muito complexos, portanto optamos por trabalhar no domínio da frequência. Na próxima etapa, também conhecido como etapa de pré-ênfase, três operações são realizadas no sinal: aplicação de um filtro de pré-ênfase, divisão em quadros e janelamento. Isto é necessá-

rio para que se possa ter um sinal cujo espectro seja equalizado, para que a rede de reconhecimento possa ter uma chance maior de acerto, pois teremos apenas a voz do locutor, no sinal entrando na rede. Na etapa de parametrização da voz, uma das operações realizadas no sinal é a Transformada Rápida de Fourier (FFT), a qual realiza a transformação da voz do domínio do tempo para o domínio da frequência, esta é uma das etapas mais difíceis de implementação em FPGA. A última e mais importante etapa do processo é que identifica a palavra dita pelo locutor. Nesta última parte utilizaremos um processo estocástico, ou seja, probabilístico, muito utilizado hoje nas soluções tanto de *software* quanto de *hardware,* tal processo é conhecido como Modelo Escondido de Markov (HMM – *Hidden Markov Model*).

Para realizar a implementação do sistema em uma FPGA, cada etapa do processo sofre uma sequência de ações como, especificação e descrição de cada módulo constituinte, de cada etapa, em linguagem VHDL. Para isso utilizamos a ferramenta Quartus II da Altera.

2-PROCESSAMENTO DIGITAL DE SINAIS

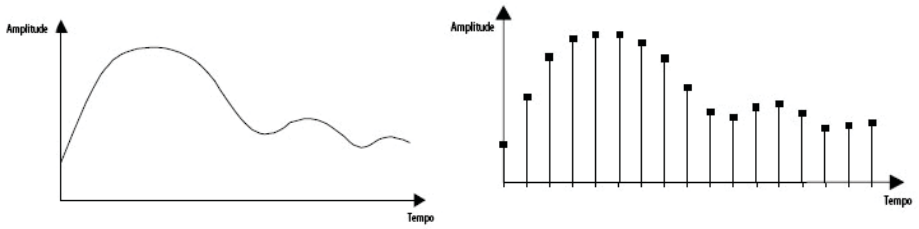
2.1- O sinal digital

Para apresentação da teoria referente ao processamento digital de sinais, este capítulo foi baseado nas referências DINIZ, P.S.R, 2004 e HAYES, M.H, 1999, onde podem ser encontradas informações mais aprofundadas sobre o tema. Como podemos realizar o processamento da voz humana, sendo que esta é um sinal analógico, ou seja, um sinal que varia continuamente no tempo? Ou ainda, como podemos analisar este mesmo sinal em um ambiente digital? A resposta para estas perguntas está na conversão do sinal de analógico (contínuo) para digital (discreto), pois assim qualquer sistema de *hardware* digital poderá realizar o processamento deste sinal de maneira fácil e rápida.

O sinal discreto pode ser definido como:

> *"Uma sequência indexada de números reais ou complexos, sendo assim uma função de uma variável de valor inteiro n" (HAYES, M.H, 1999).*

Um sinal analógico $x_a(t)$, ao passar por um conversor A/D (Analógico/Digital), é amostrado à uma frequência $\Omega_s=1/T_s$, portanto o sinal contínuo pode ser relacionado com o sinal discreto da seguinte maneira: $x(n)=x_a(nT_s)$, onde n é um número inteiro. Isso quer dizer que a função $x(n)$ é o valor instantâneo da função $x(t)$ para cada valor de n.



(a) (b)

Figura 2.1: *(a) Sinal contínuo, (b) Sinal discreto. (O Autor)*

Como pode ser observado na Figura 2.1, um sinal discreto pode também ser descrito como uma soma de impulsos unitários deslocados, cada um multiplicado por uma constante, ou seja, o impulso deslocado de k amostras é multiplicado por $x(k)$, matematicamente:

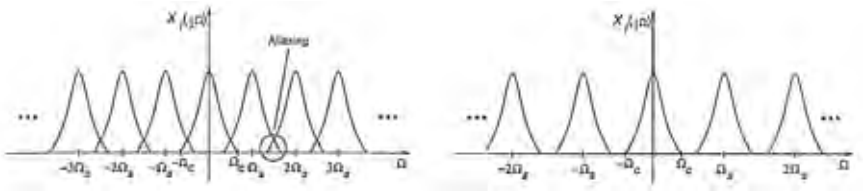
$$x(n) = \sum_{k=-\infty}^{\infty} x(k)\delta(n-k) \tag{2.1}$$

2.1.1 – Teorema da amostragem

Todo e qualquer sinal contínuo pode ser recuperado a partir de seu respectivo sinal discreto, contanto que o teorema da amostragem seja respeitado. A definição do Teorema da Amostragem é a seguinte:

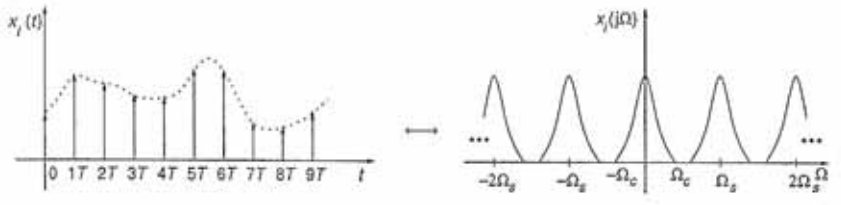
> *"Se um sinal $x_a(t)$ no tempo contínuo tem largura de faixa limitada, isto é, sua transformada de Fourier é tal que $X_a(j\Omega) = 0$ para $|\Omega| > \Omega_c$, então $x_a(t)$ pode ser completamente recuperado a partir do sinal no tempo discreto $x(n) = x_a(nT)$ se e somente se a freqüência de amostragem Ω_s satisfaz $\Omega_s > 2\Omega_c$" (DINIZ, P.S.R., 2004).*

Isso quer dizer que se desejamos recuperar o sinal amostrado, a frequência de amostragem deve ser maior que duas vezes a maior frequência presente no sinal, isto é feito para que não ocorra o *alising* fenômeno este que causa a perda de informação do sinal recuperado, como pode ser observado na Figura 2.2. A Figura 2.3 simplesmente mostra que todo sinal discreto tem uma respectiva resposta em frequência.



(a) (b)

Figura 2.2: *(a)Sinal com Ωs<2Ωc, (b) Sinal com Ωs>2Ωc. (DINIZ, P.S.R., 2004)*



(a) (b)

Figura 2.3: *(a) Sinal discreto, (b) Resposta em frequência do sinal. (DINIZ, P.S.R., 2004)*

2.1.2 – Quantização e codificação do sinal discreto.

Após a etapa de amostragem do sinal contínuo realiza-se a etapa de quantização. Um quantizador é um sistema não-linear e não-invertível, que transforma uma sequência de entrada *x(n)*, a qual tem um intervalo continuo de amplitudes, em uma sequência na qual cada valor de *x(n)* assume um dentre um número finito de valores possíveis. Os quantizadores podem ter níveis de quantização espaçados uniformemente ou não. Quando os intervalos estão espaçados uniformemente, são ditos *quantizadores lineares.* As equações 2.2 e 2.3 mostram respectivamente a resolução de um quantizador e o número de níveis de quantização, onde B é o número de bits do quantizador e X_{max} é a amplitude máxima do sinal de entrada *x(n).*

$$\Delta = \frac{X_{max}}{2^B} \tag{2.2}$$

$$L = 2^B \tag{2.3}$$

Todo quantizador tem associado a ele um erro de quantização, pois os valores são arredondados para cima se estiverem acima do nível de decisão e para baixo se estiverem abaixo do nível de decisão. O erro de quantização é limitado na metade da resolução do quantizador tanto negativamente como positivamente. As equações 2.4 e 2.5 mostram o erro de quantização e seu intervalo respectivamente.

$$e(n) = Q[x(n)] - x(n) \tag{2.4}$$

$$-\frac{\Delta}{2} < e(n) < \frac{\Delta}{2} \tag{2.5}$$

A Figura 2.4 ilustra todo o processo de quantização de um sinal, mostrando os níveis de decisão, o erro de quantização e o nível quantizado. Pode-se perceber que quanto maior for o número de bits do quantizador, mais preciso será o sinal discreto.

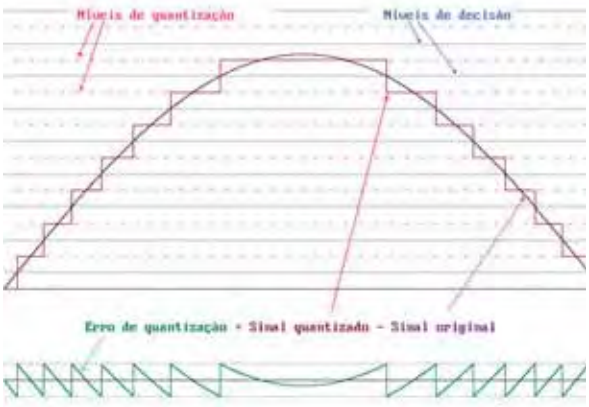


Figura 2.4: *Processo de quantização de um sinal.*
(fonte: http://www.qsl.net/py4zbz/teoria/quantiz.htm)

A próxima etapa é a codificação, onde para cada nível quantizado é atribuído uma palavra binária conforme o número de bits do codificador, o qual deve ser o mesmo do quantizador. Assim encerra-se toda a conversão de um sinal analógico para um sinal digital e todos esses processos estão presentes em um único dispositivo, o conversor A/D.

2.2- Transformadas de Fourier

A representação de Fourier desempenha um papel muito importante no processamento digital de sinais tanto de tempo contínuo quanto de tempo discreto. Através desta transformada podemos fazer com que um sinal mude de domínio, ou seja, passe do domínio do tempo para o domínio da frequência. Quando temos que aplicar alguma operação em um sinal discreto, por exemplo, um filtro, se trabalharmos no domínio do tempo a saída deste filtro será a convolução do sinal de entrada com a função do filtro. Convoluções são processos muito complicados de serem resolvidos, pois em sistemas contínuos são integrais complexas e em sistemas discretos são somatórias complexas muito longas. Ao se passar o sistema para o domínio da frequência, as convoluções se tornam simples multiplicações, facilitando-se assim os cálculos. Devido a isto as transformadas de Fourier são hoje amplamente utilizadas para diversos fins e aplicações em sistemas de processamento digital de sinais.

2.2.1- A Transformada de Fourier de tempo discreto (TFTD)

A resposta em frequência de um sistema linear invariante ao deslocamento (*LSI-Linear Shift Invariant*) é obtida através da multiplicação da entrada *x(n)* por uma exponencial

complexa $e^{-jn\omega}$, e somando em relação à *n*. A transformada de Fourier de tempo discreto é a própria resposta em frequência do sinal *x(n)*, portanto temos matematicamente:

$$X(e^{j\omega}) = \sum_{-\infty}^{\infty} x(n)e^{-jn\omega} \tag{2.6}$$

A condição para que a TFTD exista é que a somatória da equação 2.6 deva convergir, ou seja:

$$\sum_{-\infty}^{\infty} |x(n)| = S < \infty \tag{2.7}$$

Afim de, exemplificar a definição matemática da TFTD iremos calcular a TFTD do seguinte sinal:

$$x(n) = \alpha^n u(n) \quad |\alpha| < 1$$

Como para *n>0 ,u(n)=1*, temos através da equação 2.6:

$$X(e^{j\omega}) = \sum_{n=0}^{\infty} \alpha^n \, e^{-jn\omega} = \sum_{n=0}^{\infty} (\alpha e^{-j\omega})^n$$

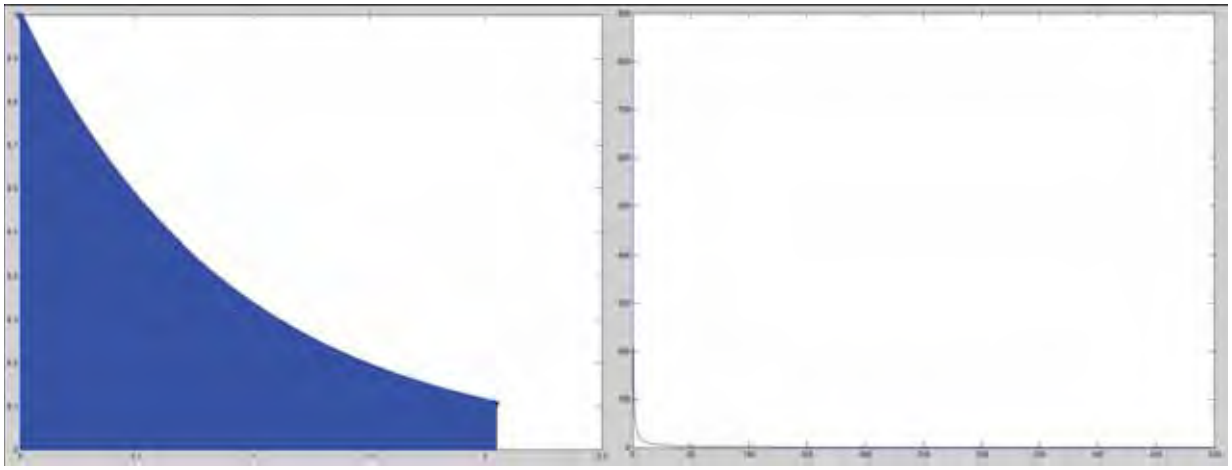
Usando a série geométrica:

$$\sum_{n=0}^{\infty} a^n = \frac{1}{1-a}, |a| < 1$$

Temos:

$$\boldsymbol{X(e^{j\omega}) = \frac{1}{1 - \alpha e^{-j\omega}}}$$

Podemos ver, pelo exemplo acima desenvolvido, o grande inconveniente da TFTD, apesar de estarmos trabalhando em um sinal discreto, a TFTD obtida deste sinal é dependente da variável contínua ω, portanto a resposta em frequência deste sinal será contínua. A Figura 2.5 ilustra graficamente, o exemplo acima para $\alpha = \frac{1}{3}$.



(a) (b)

Figura 2.5: *(a) Sinal discreto, (b) TFTD do sinal em Hz. (O Autor)*

Todo e qualquer sinal no domínio da frequência pode ser recuperado para o domínio do tempo através da transformada inversa de Fourier, mostrada na equação 2.8.

$$x(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} X(e^{j\omega})e^{jn\omega}d\omega \tag{2.8}$$

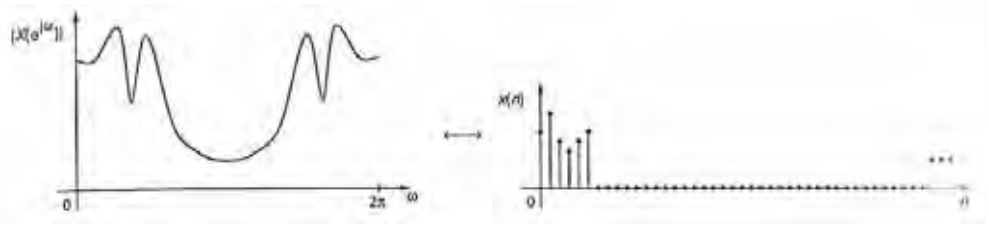
2.2.2- A transformada discreta de Fourier

Como foi visto no item 2.2.1 a TFTD é contínua, devido à variável contínua ω, tornando-se assim impossível se realizar a implementação desta transformada em qualquer tipo de aplicação digital. Para se implementar essa transformada em uma aplicação digital, precisamos ter uma variável discreta para representar a frequência. Podemos obter isso através da própria TFTD, bastando apenas amostrar de forma uniforme a variável contínua de frequência ω. Assim estamos realizando o mapeamento de um sinal que depende de uma variável discreta *n*, em uma transformada que depende de uma variável discreta de frequência *k*. Esse processo veio a ficar conhecido como Transformada Discreta de Fourier (DFT - *Discrete Fourier Transform*). As representações matemáticas da DFT e de sua inversa IDFT, são mostradas nas equações 2.9 e 2.10, respectivamente.

$$X\left(e^{j\frac{2\pi k}{N}}\right) = \sum_{n=0}^{N-1} x(n)e^{-j\frac{2\pi}{N}kn} \tag{2.9}$$

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X\left(e^{j\frac{2\pi k}{N}}\right) e^{j\frac{2\pi k}{N}n}, \quad para\ 0 \leq n \leq N-1 \tag{2.10}$$

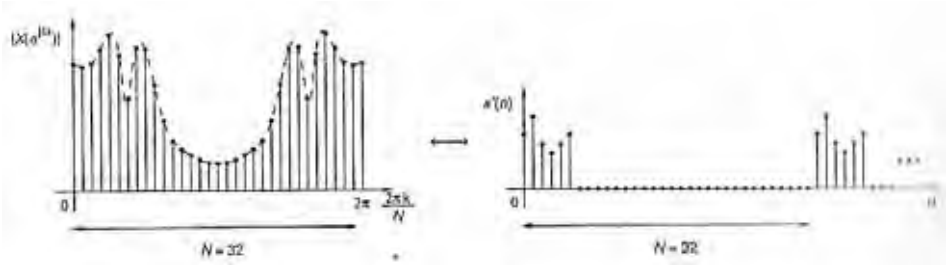
Onde a variável ω, foi amostrada em múltiplos inteiros *N* do período da transformada, que é 2π, uniformemente espaçadas entre 0 e 2π. As frequências agora serão: $\omega_k = 2\pi k/N$, onde $k \in Z$. As amostras da transformada de Fourier podem fornecer uma efetiva representação discreta na frequência para um sinal de comprimento finito no tempo discreto. Porém essa representação só é útil se o número *N* de amostras da transformada de Fourier é maior ou igual ao comprimento *L* do sinal original. Se esta condição não for respeitada o sinal original não poderá ser recuperado. Tal condição é equivalente ao Teorema da Amostragem aplicado à transformada de Fourier de tempo discreto.



(a) (b)

Figura 2.6: *(a) TFTD de x(n), (b) Sinal discreto de entrada. (DINIZ, P.S.R., 2004)*

As Figuras 2.6 e 2.7, mostram graficamente a diferença entre a TFTD e a DFT, e respectivamente o numero de amostras necessárias do sinal de entrada para se calcular cada uma das transformadas exemplificadas.



(a) (b)

Figura 2.7: *(a) Transformada Discreta de x(n), (b) Sinal de entrada com 32 amostras. (DINIZ, P.S.R., 2004)*

Através das equações 2.9 e 2.10, pode-se perceber que o número de multiplicações complexas para se calcular a DFT, é N^2 para uma amostra de tamanho *N*, incluindo possíveis multiplicações triviais por 1. Se retirarmos essas multiplicações triviais, o número de multiplicações complexas é de $(N-1)^2$ e o de adições complexas é $N(N-1)$.

2.2.3- A Transformada rápida de Fourier

Como para se calcular uma DFT, do modo convencional, ou seja, pela própria definição, são necessárias o quadrado do tamanho da DFT, de multiplicações complexas, sendo que operações complexas são complicadas de serem realizadas em aplicações digitais. Para se amenizar este problema, temos que realizar o menor número possível de operações complexas. Assim foram desenvolvidos algoritmos específicos para o calculo de DFT's, tais algoritmos, que hoje são diversos, são chamados de FFT, do inglês *Fast Fourier Transforms*, Transformadas Rápidas de Fourier, pois calculam uma DFT com bem menos operações complexas do que se precisaria se usássemos a definição. Explicaremos aqui o primeiro algoritmo, e mais simples, que foi desenvolvido por Cooley & Tukey em 1965. Tal algoritmo é chamado de *"Algoritmo de raiz 2 com decimação no tempo"*.

2.2.3.1 - Algoritmo de raiz 2 com decimação no tempo

Inicialmente iremos supor que o sinal de entrada $x(n)$ tenha um comprimento *N* e este comprimento é uma potência de dois, ou seja, $N=2^l$. Fazendo-se também a substituição: $W_N=e^{-j2\pi/N}$, substituindo-o na equação 2.9 e em seguida dividindo-se o somatório em duas partes, uma apenas com elementos de índices pares e outra com elementos de índice ímpar de $x(n)$, assim temos:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \tag{2.11}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x(2n) W_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) W_N^{(2n+1)k}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x(2n) W_N^{2nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) W_N^{2nk} \tag{2.12}$$

Se notarmos que para *N* par, temos:

$$W_N^{2nk} = e^{-j\frac{2\pi}{N}2nk} = e^{-j\frac{2\pi}{\frac{N}{2}}nk} = W_{\frac{N}{2}}^{nk} \tag{2.13}$$

Então a equação 2.12 se torna:

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(2n) W_{\frac{N}{2}}^{nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) W_{\frac{N}{2}}^{nk} \tag{2.14}$$

Portanto podemos ver quer uma DFT de tamanho *N*, pode ser calculada a partir de duas DFT's de tamanho *N/2*, além das multiplicações por W_N^k. Como cada nova DFT tem tamanho *N/2*, então teremos $(N/2)^2$ multiplicações complexas, mais as *N* multiplicações por W_N^k, pois há uma multiplicação de W_N^k para cada *k* entre *0* e *N-1*. Portanto para o cálculo da FFT, teremos:

$$2(\frac{N}{2})^2 + N = \frac{N^2}{2} + N \tag{2.15}$$

Como $N^2/2+N$ é menor que N^2 para $N>2$, então já podemos perceber que houve um decréscimo no número de multiplicações complexas realizadas quando comparadas ao cálculo usual da DFT.

Em relação ao número de adições complexas, temos que calcular duas DFT's de tamanho *N/2* e após a multiplicação por W_N^k, temos realizar as *N* adições das duas DFT's parciais, uma para cada *k* entre *0* e *N-1*.

$$2\left[\left(\frac{N}{2}\right)^2 - \frac{N}{2}\right] + N = \frac{N^2}{2} \tag{2.16}$$

Que é menor que as *N(N-1)* adições necessárias para se calcular a DFT pelo método convencional. Como *N* é uma potência de 2, é fácil ver que se o procedimento mostrado na equação 2.14 é aplicado recursivamente, a cada uma das DFT's resultantes, até que todas as DFT's remanescentes sejam de comprimento 2, podemos chegar à uma redução significativa

da complexidade. O procedimento completo é formalizado em um algoritmo escrevendo-se, primeiro:

$$X(k) = X_e(k) + W_N^k X_o(k) \tag{2.17}$$

Onde $X_e(k)$ e $X_o(k)$ são as DFT's de comprimento $N/2$ dos índices pares e ímpares, respectivamente, isto é:

$$\left.\begin{aligned} X_e(k) &= \sum_{n=0}^{\frac{N}{2}-1} x(2n) W_{\frac{N}{2}}^{nk} = \sum_{n=0}^{\frac{N}{2}-1} x_e(n) W_{\frac{N}{2}}^{nk} \\ X_o(k) &= \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) W_{\frac{N}{2}}^{nk} = \sum_{n=0}^{\frac{N}{2}-1} x_o(n) W_{\frac{N}{2}}^{nk} \end{aligned}\right\} \tag{2.18}$$

Agora separando-se novamente cada uma das DFT's da equação 2.18 em seus componentes pares e ímpares, obtemos:

$$\left.\begin{aligned} X_e(k) &= \sum_{n=0}^{\frac{N}{4}-1} x_e(2n) W_{\frac{N}{4}}^{nk} + W_{\frac{N}{2}}^{k} \sum_{n=0}^{\frac{N}{4}-1} x_e(2n+1) W_{\frac{N}{4}}^{nk} \\ X_o(k) &= \sum_{n=0}^{\frac{N}{4}-1} x_o(2n) W_{\frac{N}{4}}^{nk} + W_{\frac{N}{2}}^{k} \sum_{n=0}^{\frac{N}{4}-1} x_o(2n+1) W_{\frac{N}{4}}^{nk} \end{aligned}\right\} \tag{2.19}$$

De tal forma que:

$$\left.\begin{aligned} X_e(k) &= X_{ee}(k) + W_{\frac{N}{2}}^{k} X_{eo}(k) \\ X_o(k) &= X_{oe}(k) + W_{\frac{N}{2}}^{k} X_{oo}(k) \end{aligned}\right\} \tag{2.20}$$

Onde $X_{ee}(k)$, $X_{eo}(k)$, $X_{oe}(k)$, $X_{oo}(k)$ correspondem agora, as DFT's de tamanho $N/4$. Assim a aplicação recursiva do procedimento pode conduzir o cálculo de uma DFT de comprimento $N=2^l$, ao longo de l etapas, até o cálculo de 2^l DFT's de comprimento 1, porque cada etapa converte uma DFT de comprimento L em duas DFT's de comprimento $L/2$, mais uma multiplicação complexa e uma adição complexa. Portanto levando-se em conta todo o proces-

so do algoritmo, para se realizar a FFT de tamanho N temos que realizar $Nlog_2N$ multiplicações e adições complexas. Para o caso de, por exemplo, uma FFT de tamanho 1024, teremos, se realizarmos o algoritmo acima descrito de FFT, 100 vezes menos multiplicações complexas do que o método convencional (DINIZ, P.S.R., 2004)

Podemos agora elaborar uma representação gráfica para o procedimento explicado. Se notarmos a operação contida na equação 2.20, cada valor $X_{ie}(k)$ ou $X_{io}(k)$ é usado duas vezes. Isso ocorre porque, se $X_i(k)$ tem período L, então $X_{ie}(k)$ e $X_{io}(k)$ tem período $L/2$. Em outras palavras:

$$X_i(k) = X_{ie}(k) + W_L^k X_{io}(k) \tag{2.21}$$

$$X_i\left(k+\frac{L}{2}\right) = X_{ie}\left(k+\frac{L}{2}\right) + W_L^{k+\frac{L}{2}} X_{io}\left(k+\frac{L}{2}\right)$$

$$= X_{ie}(k) + W_L^{k+\frac{L}{2}} X_{io}(k) \tag{2.22}$$

Assim, se temos as DFT's de comprimento $L/2$, $X_{ie}(k)$ e $X_{io}(k)$, podemos calcular a DFT de comprimento L $X_i(k)$ aplicando as equações 2.22 e 2.21, para $k=0...(L/2-1)$. Essa operação está representada graficamente na Figura 2.8. Como o algoritmo da DFT é composto de repetições desse procedimento, o diagrama apresentado é chamado de célula básica do algoritmo. Devido à sua aparência ele também é chamado de célula borboleta.

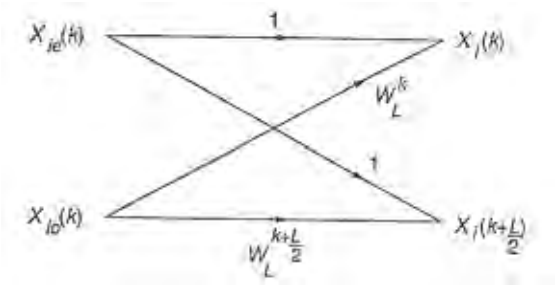


Figura 2.8: *Célula básica da FFT. (DINIZ, P.S.R., 2004)*

Ainda sim podemos diminuir o número de multiplicações complexas requeridas pelo algoritmo, notando a seguinte propriedade na equação 2.22.

$$W_L^{k+\frac{L}{2}} = W_L^k W_L^{\frac{L}{2}} = W_L^k W_2 = -W_L^k \tag{2.23}$$

Assim a equação 2.22 se torna:

$$X_i\left(k+\frac{L}{2}\right) = X_{ie}(k) - W_L^k X_{io}(k) \tag{2.24}$$

Temos então a célula básica mais eficiente do algoritmo de FFT com decimação no tempo, ilustrada na Figura 2.9.

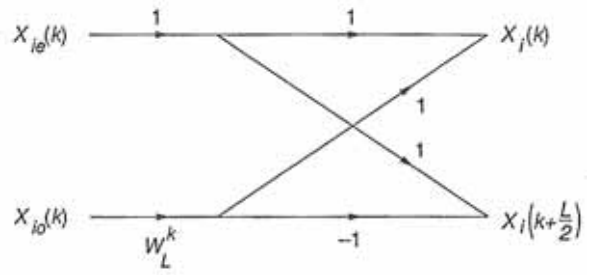


Figura 2.9: *Célula básica mais eficiente da FFT raiz 2. (DINIZ, P.S.R., 2004)*

A fim de exemplificar o algoritmo completo, a Figura 2.10 mostra a estrutura do algoritmo raiz 2 com decimação no tempo, para um sinal de entrada de tamanho oito, ou seja *N=8*, utilizando as células básicas mais eficientes. Com essa célula básica, em cada célula temos apenas uma multiplicação complexa ao invés de duas como na outra célula. Assim no total teremos $(N/2)log_2N$ multiplicações complexas, diminuindo ainda mais os cálculos.

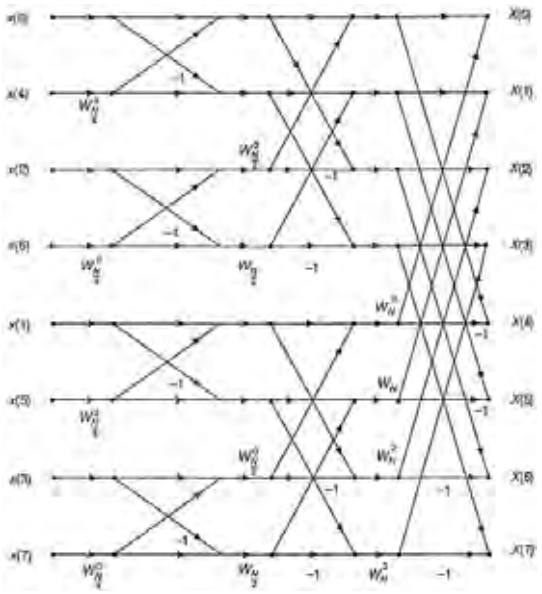


Figura 2.10: *Diagrama da FFT de oito pontos. (DINIZ, P.S.R., 2004)*

Podemos perceber pela Figura 2.10, que as posições do vetor de saída estão de maneira sequencial, porém as do vetor de entrada não. Isso se dá pois, os índices pares tem o bit menos significativo igual a 0, e os índices ímpares tem o bit menos significativo igual a 1. Assim se caminharmos da direita para a esquerda, na primeira DFT, a parte superior são os índices com LSB igual a 0 e as parte inferior os índices com LSB igual a 1. Na segunda DFT a primeira parte superior corresponde aos índices com o segundo LSB igual a 1 a a segunda parte superior aos índices com o segundo LSB igual a 0, e assim por diante. Se continuarmos nesse raciocínio chegaremos a conclusão que para que o vetor de saída tenha a sequencia correta, o vetor de entrada deve ser dado de forma que o índice do vetor corresponda ao índice da sequência com os bits na ordem inversa. Por exemplo, x(6)=x(110), deverá ocupar a posição (3)=(011) na entrada da FFT.

2.3- Filros digitais

A aplicação de filtros em qualquer tipo de sistema, tanto analógico quanto digital, tem diversas funções, podemos citar algumas delas, como: seleção de frequência para autofalantes, eliminar ruídos de determinada faixa de frequência, etc. Existem hoje diversas técnicas para a implementação de filtros digitais nas aplicações tanto de *software*, quanto de *hardware*. Aqui abordaremos apenas algumas dessas técnicas, as mais utilizadas, como a forma direta, a forma canônica alternativa, a forma paralela e a forma por amostragem na frequência. Os filtros digitais são divididos em duas categorias, sendo: os filtros não-recursivos (*FIR - Finite Impulse Response*) e os filtros recursivos (*IIR – Infinite Impulse Response*). Os filtros FIR são filtros cuja função de transferência segue a forma polinomial, já a função de transferência dos filtros IIR segue a forma racional polinomial.

2.3.1 – Filtros não recursivos *FIR - Finite Impulse Response*

A estrutura de um sistema com um filtro FIR é mostrada na Figura 2.11.

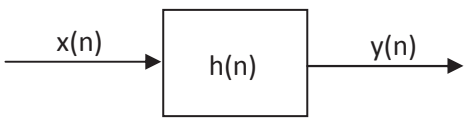


Figura 2.11: *Sistema de um filtro digital. (O Autor)*

A saída de um sistema digital é dada, pela convolução linear do sinal de entrada *x(n)* e a função de transferência do filtro *h(n)*, ou seja, matematicamente temos:

$$y(n) = \sum_{l=0}^{M} h(l)x(n-l) \tag{2.25}$$

Onde *M* é a ordem do filtro. Como todas as análises de filtros são feitas no domínio da freqüência, através da transformada Z podemos reescrever a equação 2.25, da seguinte forma:

$$H(z) = \frac{Y(z)}{X(z)} = \sum_{l=0}^{M} h(l)z^{-l} \tag{2.26}$$

2.3.1.1 - Forma direta

Tal forma tem este nome, pois os seus coeficientes multiplicadores são obtidos diretamente da função de transferência do filtro, aplicando-se assim a própria definição do filtro mostrada na equação 2.26. A Figura 2.12 ilustra graficamente a implementação de um filtro FIR na forma direta. Uma outra forma mais utilizada é a forma direta canônica alternativa, na qual se usa um menor número de atrasos possíveis obtendo-se assim uma vantagem, em relação a forma direta convencional, ilustrada na Figura 2.13.

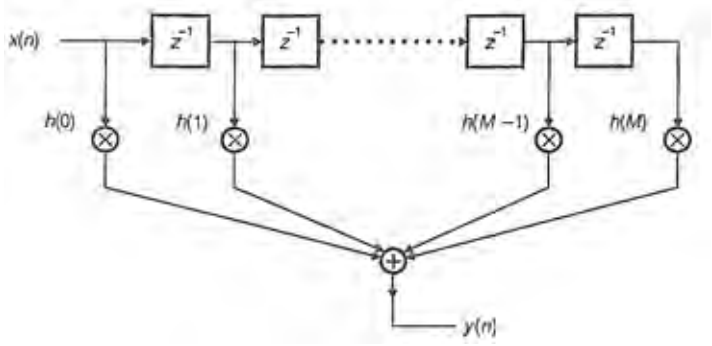


Figura 2.12: *Forma direta de um filtro FIR. (DINIZ, P.S.R., 2004)*

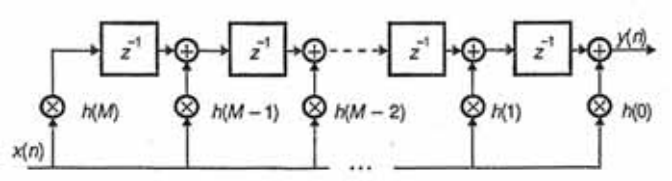


Figura 2.13: *Forma direta alternativa de um filtro FIR. (DINIZ, P.S.R., 2004)*

Para que estes métodos possam ser implementados devemos conhecer os coeficientes da função de transferência do filtro desejado. Para tanto devemos primeiro entender a resposta deste filtro à um impulso, e depois tentar através de algumas técnicas, a serem explicadas nos itens 2.3.1.2 e 2.3.1.3, chegar a resultados o mais próximos possíveis da resposta ideal.

Para um filtro passa-baixas ideal temos a seguinte resposta em frequência, em módulo, e a respectiva resposta ao impulso no domínio do tempo.

$$\left|H\left(e^{j\omega}\right)\right| = \begin{cases} 1, \ para\ |\omega| \leq |\omega_c| \\ 0, \ para\ \omega_c \leq |\omega| \leq \pi \end{cases} \tag{2.27}$$

$$h(n) = \frac{1}{2\pi}\int_{-\omega_c}^{\omega_c} e^{j\omega}\, d\omega = \begin{cases} \dfrac{\omega_c}{\pi}, \ para\ n = 0 \\ \dfrac{sen(\omega_c n)}{\pi n}, \ para\ n \neq 0 \end{cases} \tag{2.28}$$

De maneira similar, para um filtro rejeita-faixas, a resposta ideal a um impulso, em módulo, na frequência e no domínio do tempo é mostra nas equações 2.29 e 2.30, respectivamente.

$$\left|H\left(e^{j\omega}\right)\right| = \begin{cases} 1, \ para\ |\omega| \leq |\omega_c| \\ 0, \ para\ \omega_{c1} < |\omega| \leq \omega_{c2} \\ 1, \ para\ \omega_{c2} \leq |\omega| \leq \pi \end{cases} \tag{2.29}$$

$$\begin{aligned} h(n) &= \tfrac{1}{2\pi}\left[\int_{-\omega_{c1}}^{\omega_{c1}} e^{j\omega}\, d\omega + \int_{\omega_{c2}}^{\pi} e^{j\omega}\, d\omega + \int_{-\pi}^{-\omega_{c2}} e^{j\omega}\, d\omega\right] \\ &= \begin{cases} 1 + \frac{\omega_{c1}-\omega_{c2}}{\pi}, & para\ n = 0 \\ \frac{1}{\pi n}[sen(\omega_{c1}n) - seb(\omega_{c2}n)], & para\ n \neq 0 \end{cases} \end{aligned} \tag{2.30}$$

Analogamente às equações 2.27 a 2.30, as respostas ideais ao impulso para os filtros passa-altas e passa-faixas podem ser obtidas.

2.3.1.2 – Filtros FIR por amostragem na frequência

De maneira geral, o grande problema do projeto de filtros FIR é se encontrar uma resposta ao impulso de duração finita, cuja transformada de Fourier se aproxime suficientemente bem de uma dada resposta na frequência. Para isso basta notarmos a definição da DFT, onde esta corresponde às amostras da TFTD nas frequências, $\omega=2\pi k/N$. Portanto, para se projetar um filtro FIR de comprimento N, basta termos uma DFT que corresponda exatamente às amostras da resposta na frequência desejada de um filtro *h(n).* Assim para obtermos *h(n)* aplicamos a transformada inversa de Fourier descrita na equação 2.10. Para filtros FIR de fase linear, a Tabela 2.1, nos mostra as equações, para cada um dos quatro tipos de filtros, onde podemos obter os coeficientes do filtro desejado. Sendo os tipos: *Tipo I* – Ordem *M* par e resposta ao impulso simétrica, *Tipo II* – Ordem *M* ímpar e resposta ao impulso simétrica, *Tipo III* – Ordem *M* par e resposta ao impulso anti-simétrica e *Tipo IV* – Ordem *M* ímpar e resposta ao impulso anti-simétrica.

Filtro	Resposta ao impulso $h(n)$, para $n = 0, \ldots, M$	Restrição
Tipo I	$\frac{1}{M+1}\left[A(0)+2\sum_{k=1}^{\frac{M}{2}}(-1)^k A(k)\cos\frac{\pi k(1+2n)}{M+1}\right]$	
Tipo II	$\frac{1}{M+1}\left[A(0)+2\sum_{k=1}^{\frac{M-1}{2}}(-1)^k A(k)\cos\frac{\pi k(1+2n)}{M+1}\right]$	$A\left(\frac{M+1}{2}\right)=0$
Tipo III	$\frac{2}{M+1}\sum_{k=1}^{\frac{M}{2}}(-1)^{k+1}A(k)\,\mathrm{sen}\,\frac{\pi k(1+2n)}{M+1}$	$A(0)=0$
Tipo IV	$\frac{1}{M+1}\left[(-1)^{\frac{M+1}{2}+n}A\left(\frac{M+1}{2}\right)+2\sum_{k=1}^{\frac{M-1}{2}}(-1)^k A(k)\,\mathrm{sen}\,\frac{\pi k(1+2n)}{M+1}\right]$	$A(0)=0$

Tabela 2.1: *Resposta ao impulso para filtros FIR por amostragem na frequência. (DINIZ, P.S.R., 2004)*

Este método, porém, contém um grande inconveniente, há em sua resposta de módulo uma ondulação considerável nas proximidades da banda de passagem e da banda de rejeição. Isso fez com que este método não fosse o mais adequado para a aplicação no projeto de filtros. No entanto existe uma situação em que este método fornece resultados precisos, basta respeitarmos o seguinte teorema:

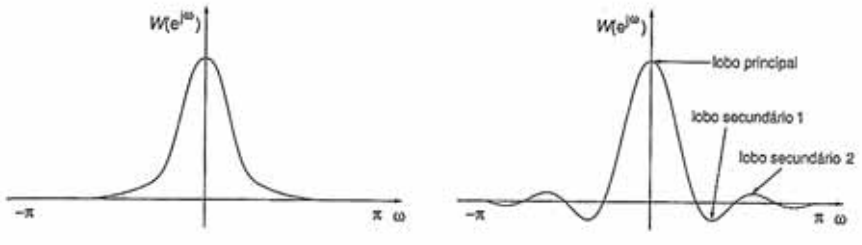
"Se a resposta em frequência desejada é uma soma finita de senóides complexas igualmente espaçadas na frequência, então o método da amostragem na frequência fornece resultados exatos, exceto por um termo de atraso de grupo constante, contanto que o comprimento da resposta ao impulso, N, satisfaça $N \geq N_1 - N_0 + 1$*" (DINIZ, P.S.R., 2004).*

2.3.1.3 – Filtros FIR com funções-janela

Como pela própria definição de filtros ideais, a resposta ao impulso destes filtros tem duração infinita, nos levando a filtros impossíveis de ser implementados, portanto precisamos fazer com que esta resposta de duração infinita, tenha uma duração finita. Porém se realizarmos apenas o truncamento da equação do filtro, teremos sérios problemas nas faixas de transição. Surgirão ondulações nas extremidades da faixa de passagem devido à convergência lenta da série de Fourier quando aproxima funções que apresentam descontinuidades. As ondulações de elevada amplitude que surgem nessas extremidades, são comumente chamadas de oscilações de Gibbs. Tais oscilações não podem ser diminuídas com o aumento da ordem *M* do filtro, limitando drasticamente as aplicações práticas deste tipo de equação.

No entanto, embora não possamos remover as oscilações de Gibbs, podemos controlar a sua amplitude multiplicando a resposta ao impulso *h(n)* por uma função-janela *w(n).* A função-janela tem como principal função amenizar as oscilações de Gibbs, introduzindo um mínimo de desvio em relação à resposta ideal do filtro. Assim os coeficientes da resposta ao impulso resultante *h'(n)* se tornam:

$$h'(n) = h(n)w(n) \tag{2.31}$$



(a) *(b)*

Figura 2.14: *(a) Resposta ideal na frequência (b) Resposta prática na frequência. (DINIZ, P.S.R., 2004)*

Como a função janela é um truncamento, então ela causa as oscilações de Gibbs também, como pode ser visto na Figura 2.14b, devido à existência dos lóbulos secundários na sua

resposta em frequência. A largura no lóbulo principal determina a largura da faixa de transição do filtro. Assim para que possamos ter uma resposta mais próxima possível da resposta ideal do filtro, temos que fazer com que a amplitude do lóbulo principal seja muito maior que a do lóbulo secundário e a energia têm que decair rapidamente à medida que $|\omega|$ aumenta de 0 a π.

Diversas são as funções janelas existentes, as mais importantes e mais utilizadas são: a janela retangular, janela de Hamming, janela de Hanning, janela de Blackman e janela de Kaiser. A Tabela 2.2 nos mostra cada uma destas funções-janela com exceção da janela de Kaiser.

Retangular	$w(n) = \begin{cases} 1 & 0 \le n \le N \\ 0 & \text{demais casos} \end{cases}$
Hanning	$w(n) = \begin{cases} 0{,}5 - 0{,}5\cos\left(\frac{2\pi n}{N}\right) & 0 \le n \le N \\ 0 & \text{demais casos} \end{cases}$
Hamming	$w(n) = \begin{cases} 0{,}54 - 0{,}46\cos\left(\frac{2\pi n}{N}\right) & 0 \le n \le N \\ 0 & \text{demais casos} \end{cases}$
Blackman	$w(n) = \begin{cases} 0{,}42 - 0{,}5\cos\left(\frac{2\pi n}{N}\right) + 0{,}08\cos\left(\frac{4\pi n}{N}\right) & 0 \le n \le N \\ 0 & \text{demais casos} \end{cases}$

Tabela 2.2: *principais funções-janela. (HAYES, M.H, 1999)*

Afim de exemplificação, a Figura 2.15 mostra o efeito de cada uma das janelas mostradas na Tabela 2.2, em um filtro rejeita-faixas.

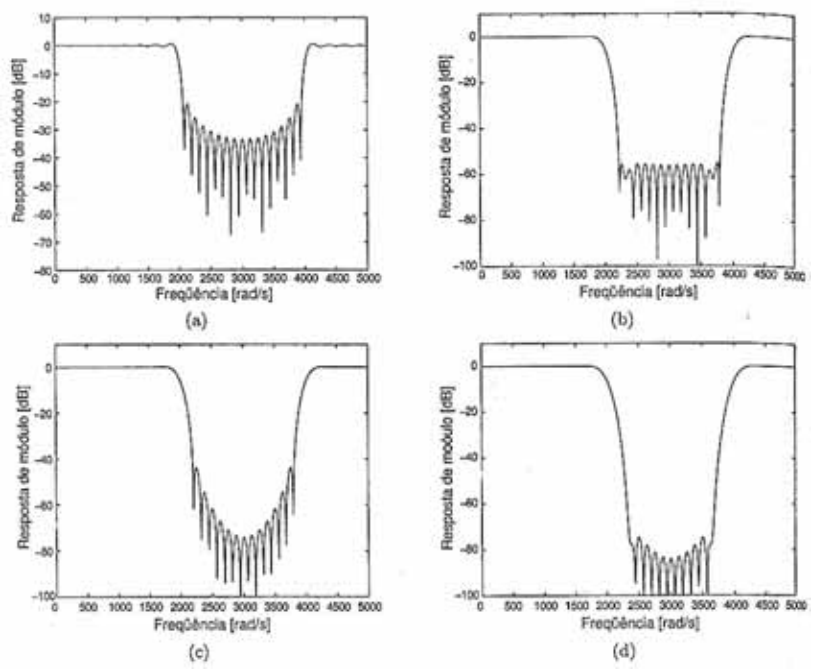


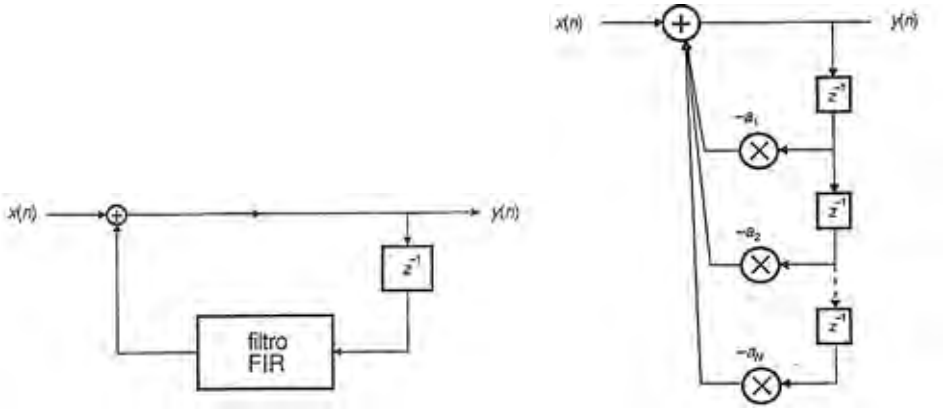
Figura 2.15: *Resposta de módulo usando janela (a) Retangular; (b) Hamming; (c) Hanning; (d) Blackman. (DINIZ, P.S.R., 2004)*

2.3.2 – Filtros recursivos IIR-Infinite Impulse Response

Filtros recursivos são filtros que tem uma função de transferência na forma polinomial racional, como descreve a equação 2.32.

$$H(z) = \frac{N(z)}{D(z)} = \frac{\sum_{i=0}^{M} b_i z^{-i}}{1 + \sum_{i=1}^{N} a_i z^{-i}} \tag{2.32}$$

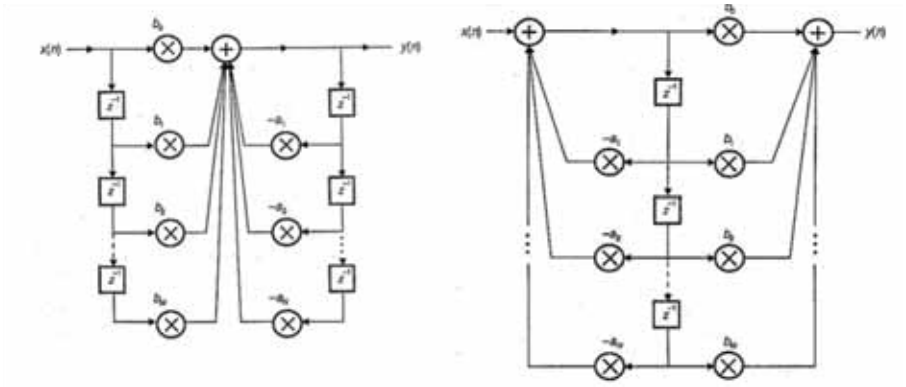
Pode-se considerar inicialmente que a função de transferência do filtro *H(z),* pode ser obtida da cascata de dois filtros independentes, *N(z)* e *1/D(z)*. O polinômio *N(z)* pode ser formado com a forma direta de um filtro FIR, já o polinômio *1/D(z)* resulta de uma realimentação positiva, onde na malha de realimentação temos um atraso e um filtro FIR, como mostra a Figura 2.16.



(a) *(b)*

Figura 2.16: *(a) Filtro 1/D(z) em diagrama de blocos; (b) Filtro 1/D(z) em detalhes. (DINIZ, P.S.R., 2004)*

Assim se juntarmos o filtro FIR na forma direta, para formar *N(z)* e o filtro mostrado na Figura 2.16, de forma que fiquem em cascata, teremos então um filtro do tipo IIR, com função de transferência *H(z).* A Figura 2.17 mostra a forma direta não canônica do filtro IIR e a forma direta canônica.



(a) (b)

Figura 2.17: *(a) Forma direta filtro IIR; (b) Forma direta canônica filtro IIR. (DINIZ, P.S.R., 2004)*

Uma outra forma de se realizar o filtro IIR é a forma paralela, a qual consiste em separar a equação típica do filtro IIR em frações parciais, na qual se utilizam equações de segunda ordem, ou seja, são feitos diversos filtros de segunda ordem e então colocados de maneira paralela. Diversas são as técnicas de implementação de um filtro IIR, basicamente estas técnicas consistem em se utilizar as equações de filtros analógicos, como: o Butterworth, Chebyshev e elíptica. Com as equações normalizadas destes filtros, faz-se então uma transformação do domínio do tempo contínuo para o domínio do tempo discreto, através de métodos como o da invariância ao impulso e da transformação bilinear. Assim após este processo

teremos a função de transferência do filtro IIR com seus respectivos coeficientes, ou seja, com seus pólos e zeros conhecidos. Os filtros IIR têm algumas vantagens em relação aos filtros FIR, por exemplo, os filtros IIR necessitam de menos multiplicações que os filtros FIR, todos os tipos de filtro IIR podem ser obtidos a partir de uma modificação de um filtro tipo passa-baixas. Essas vantagens podem ser relevantes em certos tipos de aplicações, porém a implementação dos filtros IIR é mais complexa do que a dos filtros FIR.

3 – O MODELO ESCONDIDO DE MARKOV E O PROCESSADOR LPC

Para apresentação da teoria referente ao Modelo Escondido de Markov, este capítulo foi baseado na referência RABINER, L, 1993, no qual informações mais aprofundadas sobre o tema podem ser encontradas. Existem hoje diversas técnicas, já dominadas, por aqueles que trabalham com sistemas de reconhecimento de voz. Tais técnicas podem ter abordagens tanto em padrões temporais do sinal de voz, ou abordagens mais amplas como as estatísticas. É nesta última que este trabalho está fundamentado, a abordagem através da caracterização das propriedades dos quadros de um determinado padrão, chamadas de abordagem por Modelos Escondidos de Markov (*HMM - Hidden Markov Models*). Tal abordagem diz que o sinal de voz pode ser caracterizado e parametrizado como um processo aleatório, e que os parâmetros deste processo estocástico podem ser bem definidos e determinados de uma maneira precisa.

Para se utilizar as *HMMs*, é necessário realizar etapas de pré-processamentos no sinal de voz como: uma etapa de pré-ênfase e uma etapa de extração de parâmetros, a serem discutidas também ao final deste capítulo.

3.1 – Cadeias de Markov - Processos de tempo discreto

Um processo de Markov de tempo discreto pode ser descrito como, um conjunto de *N* estados, igualmente espaçados entre si, discretos no tempo, onde o sistema transita de um estado para outro de acordo com o conjunto de probabilidades associadas com cada estado. A Figura 3.1 mostra um exemplo de uma cadeia de Markov de cinco estados.

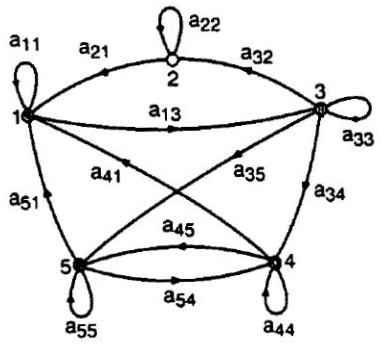


Figura 3.1: *Cadeia de Markov para N=5. (RABINER, L.R.,1993)*

Uma descrição probabilística completa da cadeia acima requer, em geral, a especificação do estado atual, bem como todos os seus estados antecedentes. Para o caso especial de um

processo de tempo discreto, a dependência probabilística é truncada em apenas um estado antecedente ao estado atual, como mostra a equação 3.1, onde *t* é o tempo discreto e q_t é o estado no instante *t*.

$$P[q_t = j | q_{t-1} = i, q_{t-2} = k, \dots] = P[q_t = j | q_{t-1} = i] \tag{3.1}$$

Além disso, nós consideramos apenas o lado direto da equação 3.1 como sendo independente do tempo, assim nos levando ao conjunto de probabilidades de transição como segue na equação 3.2.

$$a_{ij} = P[q_t = j | q_{t-1} = i], \quad 1 < i, j < N \tag{3.2}$$

Com as seguintes propriedades:

$$a_{ij} \geq 0 \quad \forall j, i \tag{3.3}$$

$$\sum_{j=1}^{N} a_{ij} = 1 \quad \forall i \tag{3.4}$$

O processo estocástico acima pode ser chamado de processo de Markov observável, pois a saída do processo é um conjunto de estados a cada instante de tempo, onde cada estado corresponde a um evento observável. Para fixar as ideias considere um modelo simples de três estados para o tempo. Assumiremos que o tempo é verificado uma vez por dia ao meio-dia. O tempo pode ser observado como sendo: Estado1, chuva; Estado2, nublado; Estado3, ensolarado.

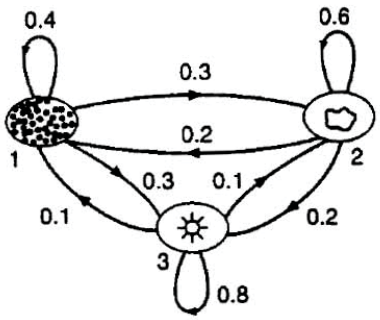


Figura 3.2: *Modelo de Markov para o tempo, N=3. (RABINER, L.R.,1993)*

Assumindo-se também que no dia *t* o tempo é caracterizado por apenas um dos estados acima citados, e que a matriz de probabilidade de transição de estados é como segue:

$$A = \{a_{ij}\} = \begin{bmatrix} 0.4 & 0.3 & 0.3 \\ 0.2 & 0.6 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{bmatrix}$$

Qual a probabilidade de que o tempo por oito dias consecutivos seja, "ensolarado, ensolarado, ensolarado, chuva, chuva, ensolarado, nublado, ensolarado", sendo que o primeiro dia é sempre ensolarado?

Podemos definir a observação como sendo:

O = (ensolarado, ensolarado, ensolarado, chuva, chuva, ensolarado, nublado, ensolarado)

= (3 , 3 , 3 , 1 , 1 , 3 , 2 , 3).

Portanto, queremos calcular a probabilidade da observação O, dado o modelo de Markov do sistema, ou seja, *P(O|Modelo).* Tal probabilidade pode ser calculada diretamente como:

$$\begin{aligned} P(O|Modelo) &= P[3,3,3,1,1,3,2,3|Modelo] \\ &= P[3]P[3|3]^2P[1|3]P[1|1]P[3|1]P[2|3]P[3|2] \\ &= \pi_3 {a_{33}}^2 a_{31}a_{11}a_{13}a_{32}a_{23} \\ &= 1.0(0.8)^2(0.1)(0.4)(0.3)(0.1)(0.2) \\ &= \mathbf{1.536x10^{-4}} \end{aligned}$$

Onde a probabilidade do estado inicial é dada pela notação:

$$\pi_i = P[q_1 = i], \qquad 1 \leq i \leq N \tag{3.5}$$

Qual a probabilidade de o tempo permanecer no mesmo estado, já conhecido, por exatamente *d* dias?

Podemos definir a observação como sendo:

O = (i, i, i, i, i, i, i, i, i, i, i, i, ... , i, j≠i)

Dias = (1,2,3,4,5,6,7,8,9,10,11,12,13, ... , d, d+1)

Dado o modelo podemos calcular a probabilidade como sendo:

$$
\begin{aligned}
P(O|Modelo) &= P[O, q_1 = i|Modelo]/P(q_1 = i) \\
&= \frac{P[i]P[i|i]^{d-1}P[j|i]}{P[i]} \\
&= \pi_i a_{ii}^{d-1}(1 - a_{ii})/\pi_i \\
&= \boldsymbol{a_{ii}^{d-1}(1 - a_{ii})} \\
&= p_i(d)
\end{aligned}
$$

A função de probabilidade distribuída, $p_i(d)$, de duração d no estado i, é uma distribuição exponencial característica da duração de um estado na cadeia de Markov. Assim podemos então, calcular o numero esperado de observações do mesmo estado, como:

$$
\begin{aligned}
\bar{d}_i &= \sum_{d=1}^{\infty} d p_i(d) \\
&= \sum_{d=1}^{\infty} d\, a_{ii}^{d-1}(1 - a_{ii}) = \boldsymbol{\frac{1}{1 - a_{ii}}}
\end{aligned}
$$

Portanto, temos como esperado para os dias ensolarados 1/0.2=5 dias, 2.5 dias para os dias nublados e 1.67 dias para os dias chuvosos. Assim para se saber qual a probabilidade basta substituir este valor calculado e as respectivas probabilidades do estado em questão na equação obtida anteriormente.

3.2 - HMM – *Hidden Markov Model*

Na seção 3.1, mostramos a ideia básica de uma cadeia de Markov, onde os estados são eventos observáveis e determinísticos. No entanto, nem sempre a saída de um estado qualquer não é aleatória. Assim este modelo se torna muito restrito para ser aplicado em diferentes tipos de problemas. As HMMs são uma extensão da cadeia de Markov, onde a observação é uma função probabilística do estado. Tal modelo é um modelo estocástico duplamente incorporado, com um processo estocástico subjacente que não é diretamente observável (é oculto), mas pode ser observado apenas através de outro conjunto de processos estocásticos que produzem a sequência de observações.

Vamos exemplificar o conceito de uma HMM da seguinte forma: imagine que você está em um quarto com uma cortina à sua frente, da qual não se consegue enxergar o que está do outro lado. Do outro lado há outra pessoa que está jogando moedas para o alto e verifican-

do o resultado. Tal pessoa não lhe dirá como as moedas estão sendo jogadas, mas apenas o resultado das mesmas. Em um dado momento a sequência de observações feita pela outra pessoa é a seguinte, “cara,cara,coroa,cora,coroa,cara,...,coroa”.

Dado o cenário acima, a principal pergunta é como iremos desenvolver uma HMM, um modelo, que irá explicar a sequência de observações das moedas? O primeiro problema a ser enfrentado é escolher quantos estados nosso modelo terá e o que cada estado corresponde em relação ao fenômeno. Podemos, por exemplo, assumir que a outra pessoa estava utilizando apenas uma moeda, neste caso o modelo teria dois estados, onde cada estado corresponderia a um lado da moeda. Assim neste caso teríamos um modelo de Markov observável, nos restando então apenas determinarmos os valores de cada parâmetro do modelo.

Um segundo modelo pode ser assumido, pode-se considerar que a outra pessoa estava utilizando duas moedas tendenciosas. Neste caso o modelo teria também dois estados e cada estado corresponderia a uma moeda. Assim cada estado seria caracterizado por uma probabilidade de distribuição entre cara e coroa, e as transições entre os estados seriam caracterizadas por uma matriz de transição de estados.

Também outro modelo ainda poderia ser assumido, poder-se-ia considerar que a outra pessoa estivesse jogando três moedas tendenciosas, e escolhendo uma entre as três, baseado em algum evento probabilístico.

Podendo-se escolher um entre os três modelos apresentados, uma pergunta surge, qual seria o modelo que melhor descreve o fenômeno ocorrido? É fácil perceber que o modelo com apenas uma única moeda, tem apenas um parâmetro a ser determinado, e que o modelo com duas moedas tem quatro parâmetros a serem determinados e o modelo com três moedas tem nove parâmetros a serem determinados. No entanto, quanto maior for o grau de liberdade, maior será a capacidade que a HMM terá em descrever ou modelar o evento ocorrido com as moedas, do que teria uma HMM menor. Porém isso é teoricamente verdade, pois existem algumas implicações práticas, que limitam fortemente o tamanho das HMM’s. Pode-se ter o caso ainda de se escolher o modelo de três moedas, mas apenas uma moeda está realmente sendo utilizada, assim estaríamos usando um sistema superdimensionado.

A Figura 3.3 mostra os três modelos exemplificados no problema descrito.

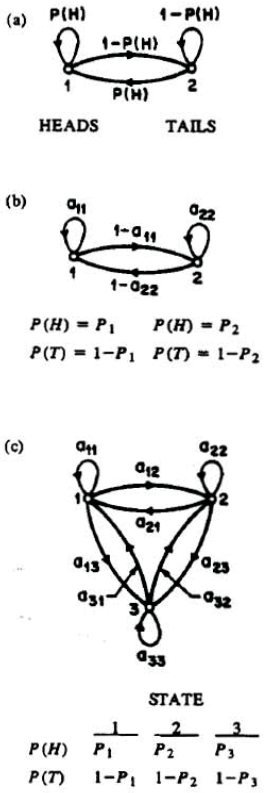


Figura 3.3: *Exemplos de modelos de HMM, (a)Modelo com uma moeda; (b)Modelo com duas moedas; (c)Modelo com três moedas. (RABINER, L.R.,1993)*

3.2.1 – Elementos de uma HMM

O exemplo anteriormente citado apresenta a ideia, do que é uma HMM e de como a mesma pode ser aplicada em alguns problemas simples. Porém quais são os elementos constituintes dessa HMM? A seguir iremos mostrar os principais elementos de uma HMM:

1. *N*, o número de estados presentes no modelo. Embora os estados sejam ocultos, em diversas aplicações práticas, estes estados tem alguma significância física atrelada ao estado ou ao conjunto de estados do modelo. De maneira geral os estados estão interconectados de uma maneira que qualquer estado pode ser alcançado de qualquer outro estado. No entanto, existem outros tipos de interconexões entre os estados que podem ser melhores para algumas aplicações de voz. Nós nomeamos cada estado como sendo um número inteiro, {1,2,3,...,*N*}.
2. *M,* número de observações distintas por estado. Os símbolos de observação correspondem a uma saída física do sistema que está sendo modelado. No exemplo das moedas os símbolos de observação eram apenas os lados da moe-

da, cara ou coroa. Usualmente denotamos os símbolos como $V=\{v_1,v_2,v_3,...,v_M\}$.

3. A probabilidade de distribuição dos estados $A = \{a_{ij}\}$ onde

$$a_{ij}=P[q_{t+1}=j|q_t=i], \quad 1 \leq i,j \leq N$$

Para o caso de cada estado poder alcançar todos os outros estados, temos que $a_{ij} \geq 0$. Para outros tipos de HMM's, temos um ou mais pares de (i,j) iguais a zero, $a_{ij}=0$.

4. A probabilidade de distribuição dos símbolos de observação é $B=\{b_j(k)\}$, no qual,

$$b_j(k)=P[o_t=v_k|q_t=j], \quad 1 \leq k \leq N$$

define a distribuição dos símbolos nos estados, j=1,2,3,...,*N*.

5. A probabilidade inicial de cada estado $\pi=\{\pi_i\}$, na qual:

$$\pi_i = P[q_1=i], \quad 1 \leq i \leq N$$

Portanto, de acordo com os itens acima, para caracterizarmos completamente uma HMM, necessitamos da especificação dos parâmetros do modelo, *N* e *M*, e da especificação dos símbolos de observação e de cada um dos conjuntos de probabilidades A, B e π. Por conveniência utilizamos a notação compactada:

$$\lambda=(A,B,\pi)$$

Tal notação é utilizada para indicar o conjunto completo de parâmetros do modelo. Esse conjunto define uma medida de probabilidade O, assim temos $P(O|\lambda)$, o qual iremos discutir logo a seguir.

3.2.2 – Gerando as observações da HMM

Dado os valores apropriados de N,M,A,B e π, a HMM pode ser usada como um gerador para dar a sequencia de observações,

$$O = (o_1,o_2,...,o_T)$$

Onde cada observação o_t é um dos símbolos de V, e T é o numero de observações na sequência. Para que a HMM gere a sequência de observações basta seguir o seguinte processo:

1. Escolha um estado inicial $q_1=i$ de acordo com a distribuição π.
2. Faça $t=1$.
3. Escolha $o_t = v_k$ de acordo com a distribuição de símbolos B.
4. Transite para o estado $q_{t+1} = j$ de acordo com a probabilidade de distribuição de transição de estados A.
5. Faça $t = t+1$; retorne para o passo 3 se t < T; se não termine o processo.

Este procedimento pode ser usado tanto como um gerador de observações, como um modelo para simular como uma dada sequência de observações foi gerada por uma apropriada HMM.

Vamos ilustrar este conceito com o mesmo problema das moedas, considerando agora um modelo de três estados com as seguintes probabilidades:

	State 1	State 2	State 3
$P(H)$	0.5	0.75	0.25
$P(T)$	0.5	0.25	0.75

Figura 3.4: *Probabilidades do exemplo das moedas modelo de três estados. (RABINER, L.R.,1993)*

E todas as probabilidades de transição de estados iguais a 1/3 e assumir a probabilidade do estado inicial igual a 1/3 também.

Observa-se a sequência:

$$O = (HHHHTHTTTT)$$

tem-se para H = cara e T = coroa

Qual sequência de estados é a mais provável? Qual a probabilidade desta sequência de observação e desta mesma sequência de estados? Qual a probabilidade da sequência de observação vir inteiramente do estado 1?

Se tivermos uma matriz de transição de estados igual a A, como ficariam as respostas para as perguntas anteriores?

$$A = \begin{bmatrix} 0.9 & 0.05 & 0.05 \\ 0.45 & 0.1 & 0.45 \\ 0.45 & 0.45 & 0.1 \end{bmatrix}$$

Dada a sequência, O = (HHHHTHTTTT) e como todas as transições de estados são equiprováveis, a sequência de estados mais provável, é aquela em que cada probabilidade de

cada observação individual é máxima. Portanto, para cada H, o estado mais apropriado é o estado 2, e para cada T o estado mais apropriado é o 3. Assim teremos a sequência:

q = (2222323333)

A probabilidade de O e q dado o modelo é:

$$P(O,q|\lambda) = (0.75)^{10}\left(\frac{1}{3}\right)^{10} = \mathbf{9.537x10^{-7}}$$

A probabilidade da sequência **q'** = (1111111111) é:

$$P(O,q'|\lambda) = (0.5)^{10}\left(\frac{1}{3}\right)^{10} = \mathbf{1.654x10^{-8}}$$

A razão entre as probabilidades dos dois casos é:

$$\frac{P(O,q|\lambda)}{P(O,q'|\lambda)} = \frac{9.537x10^{-7}}{1.654x10^{-8}} = \mathbf{57.66}$$

O que mostra, como esperado, que o primeiro caso é mais provável que o segundo.

Considerando agora o caso com a matriz de transição de estados A, a probabilidade para a primeira sequência de observações fica:

$$P(O,q|\lambda') = (0.75)^{10}\left(\frac{1}{3}\right)(0.1)^{6}(0.45)^{3} = \mathbf{1.710x10^{-9}}$$

Já para a segunda sequência de observações, temos:

$$P(O,q'|\lambda') = (0.5)^{10}\left(\frac{1}{3}\right)(0.9)^{9} = \mathbf{1.261x10^{-4}}$$

Assim temos a seguinte razão:

$$\frac{P(O,q|\lambda)}{P(O,q'|\lambda)} = \frac{1.710x10^{-9}}{1.261x10^{-4}} = \mathbf{1.356x10^{-5}}$$

Em outras palavras, por causa da não uniformidade das probabilidades da matriz de transição de estados, a segunda sequencia de observações é mais provável que a primeira neste caso. Assim dependendo da matriz de transição de estados a sequência mais provável pode mudar.

3.2.3 – Os três problemas básicos da HMM

Para que uma HMM possa ser utilizada em uma aplicação real, temos que resolver primeiramente os seus três problemas básicos, descritos a seguir:

1- Dado a sequência de observações O e o modelo λ (considerando todos os parâmetros já conhecidos), como podemos calcular eficientemente a probabilidade da observação $P(O|\lambda)$, dado o modelo?

2- Dado a sequência de observações O e o modelo λ, como podemos escolher a melhor sequência de estados para o modelo, ou seja, a sequência que melhor descreve o modelo?

3- Como podemos escolher os parâmetros A,B e π do modelo λ, que irá maximizar a probabilidade de observação $P(O|\lambda)$?

O primeiro problema se trata em como encontrar um modelo que melhor possa descrever a sequência de observações, ou seja, podemos ver este problema como sendo em quão bom um dado modelo pode ser para uma dada sequência de observações. A solução para o primeiro modelo nos permite escolher um entre os diversos, possíveis modelos, que melhor se adequam para uma dada sequência de observações.

O segundo problema é aquele, no qual nós tentamos descobrir a parte "escondida" do modelo, ou seja, achar a melhor sequência de estados. Devemos deixar claro que não existe uma sequência "correta" a ser encontrada. Porém devido a situações práticas, nós geralmente utilizamos critérios para a otimização da sequência da melhor maneira possível.

O terceiro problema é aquele em que tentamos aperfeiçoar os parâmetros do modelo, para melhor descrever a sequência de observações. A sequência de observações usadas para se ajustar os parâmetros do modelo, se chama "sequência de treinamento", porque a mesma "treina" a HMM. Este problema de treinamento é crucial para a maioria das aplicações de

HMM's, pois ele nos permite otimizar os parâmetros do modelo para certos dados de treinamento.

Vamos agora trazer estes problemas para o caso da aplicação relatada neste trabalho, o reconhecimento de palavras isoladas, independentes do locutor. Para cada palavra de um vocabulário de *W* palavras, queremos desenvolver, para cada uma delas, uma HMM de *N* estados. Nós representamos um sinal de voz, de uma dada palavra, como sendo uma sequência temporal de vetores espectrais codificados. Assumimos então que a codificação é feita usando-se um "*spectral codebook*" com *M* vetores espectrais únicos; portanto cada observação é o índice do vetor espectral o mais perto possível do sinal de voz original. Assim, para cada palavra presente neste vocabulário, temos uma sequência de treinamento, consistindo de um numero de repetições de sequências de índices de *codebook* da palavra, dita por diversos locutores diferentes. A primeira etapa a ser feita é criar os modelos para cada uma das palavras. Isso pode ser feito utilizando-se a solução para o terceiro problema, na qual iremos otimizar os parâmetros do modelo para cada uma das palavras. Para se desenvolver o entendimento físico, do significado dos estados do modelo, usamos a solução para o segundo problema, que consiste em segmentar cada sequência de treinamento, de cada uma das palavras, em estados, e então estudar as propriedades dos vetores espectrais, que nos levarão às observações ocorridas em cada estado. O principal objetivo aqui é refinar o modelo, para que a sua capacidade de reconhecimento de dada palavra melhore. Portanto, uma vez que o conjunto de *W* HMM's já estão otimizadas e devidamente desenvolvidas, o reconhecimento de uma palavra desconhecida pode ser feito, utilizando a solução para o primeiro problema, onde cada palavra terá uma pontuação, baseada na sequência de observações, e será então selecionada a palavra que tiver a maior pontuação, correspondendo esta a palavra dita pelo locutor.

3.2.4 – Solução para o Problema 1 – Avaliação das Probabilidades

Neste problema deseja-se calcular a probabilidade de ocorrer certa sequência de observações $O = (o_1,o_2,...,o_T)$, dado o modelo $P(O|\lambda)$. O modo mais direto de se fazer isso é enumerando cada possível sequência de observação de tamanho *T*. A probabilidade da sequência de observação *O*, dada a sequência de estados da equação 3.6, será:

$$q = (q_1, q_2, \dots, q_T) \tag{3.6}$$

$$P(O|q,\lambda) = \prod_{t=1}^{T} P(o_t|q_t,\lambda) \tag{3.7}$$

Assumindo que as observações são estatiscamente independentes, temos:

$$P(O|q,\lambda) = b_{q1}(o_1) \cdot b_{q2}(o_2) \cdot \ldots \cdot b_{qT}(o_T) \tag{3.8}$$

A probabilidade da sequência de estados *q*, pode ser escrita, como:

$$P(q|\lambda) = \pi_{q1} a_{q1q2} a_{q2q3} \ldots a_{qT-1qT} \tag{3.9}$$

A probabilidade de ocorrer a sequência de observações *O* e a sequência de estados *q*, é dada simplesmente pela multiplicação dos dois termos. Assim a probabilidade da sequência de observações, dado o modelo, é obtido assumindo essa probabilidade conjunta através de todas as sequências de estados possíveis. Portanto, temos então:

$$P(O|\lambda) = \sum_{q1,q2,q3,\ldots,qT} \pi_{q1} b_{q1}(o_1) a_{q1q2} b_{q2}(o_2) \ldots a_{qT-1qT} b_{qT}(o_T) \tag{3.10}$$

Podemos então observar que o cálculo desta probabilidade pela aplicação direta de sua definição, envolve na ordem de $2T*N^T$ cálculos, pois para cada $t=1,2,\ldots,T$ tem-se N possíveis estados que podem ser alcançados, e para cada sequencia de estados são requeridos em torno de $2T$ cálculos para cada termo da somatória da equação 3.10. Claramente vê-se que pela aplicação direta da definição da probabilidade teremos um esforço computacional muito grande, sendo assim necessário um procedimento mais eficiente para o cálculo desta probabilidade. Tal procedimento existe e é discutido nos itens a seguir.

3.2.4.1 – Procedimento *Forward*

Consideremos a variável $\alpha_t(i)$ definida por:

$$\alpha_t(i) = P\left(o_1 o_2 \ldots o_{t,} q_t = i \middle| \lambda\right) \tag{3.11}$$

a qual consiste na probabilidade da sequência de observações parciais, $o_1 o_2 \ldots o_{t,}$ de estado i no tempo t, dado o modelo λ. Podemos solucionar a equação 3.11 iterativamente, como segue:

1. Inicialização

$$\alpha_1(i) = \pi_i b_i(o_1) \qquad 1 \leq i \leq N.$$

2. Iteração

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij}\right] b_j(o_{t+1}) \quad \begin{matrix} 1 \leq t \leq T-1 \\ 1 \leq j \leq N \end{matrix}$$

3. Término

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$$

O algoritmo funciona da seguinte maneira, o passo 1 realiza a inicialização da variável como sendo a probabilidade conjunta do estado i com a observação o_1. O Passo 2 é o coração do algoritmo, neste um estado j, pode ser alcançado no tempo $t+1$ a partir de N possíveis estados, i, $1 \leq i \leq N$,no tempo t. Como $\alpha_t(i)$ é a probabilidade do evento conjunto no qual $o_1 o_2 \ldots o_{t,}$ são observados, e o estado no tempo t é i, o produto $\alpha_t(i) a_{ij}$ é a probabilidade do evento conjunto no qual $o_1 o_2 \ldots o_{t,}$ são observados, , e o estado no tempo t é j. Somando esse produto através de todos os N estados possíveis no tempo t, resulta na probabilidade do estado j no tempo $t+1$ acompanhado de todas as probabilidades parciais precedentes. Uma vez que o estado j é conhecido, $\alpha_{t+1}(j)$ pode ser obtido contabilizando a observação o_{t+1} no estado j, ou seja, multiplicando a somatória pela probabilidade $b_j(o_{t+1})$. Tal procedimento é realizado para todos os estados j, $1 \leq j \leq N$, em um dado tempo t, o cálculo é então iterado para $t=1,2,\ldots,T-1$. Por último o passo 3 nos dá o cálculo desejado de $P(O|\lambda)$ como sendo a somatória das probabilidades parciais $\alpha_T(i)$. Neste caso são requeridos na ordem de N^2T cálculos, para se obter a probabilidade, tendo portanto um esforço computacional muito inferior à aplicação direta da definição da probabilidade em questão.

3.2.4.2 – Procedimento *Backward*

De maneira similar podemos considerar a variável $\beta_t(i)$ definida por:

$$\beta_t(i) = P\left(o_{t+1}o_{t+2} \ldots o_t, q_t = i \middle| \lambda\right) \tag{3.12}$$

a qual consiste na probabilidade da sequência de observações parciais, do estado *t+1* até o fim, dado o estado *i* no tempo *t* e o modelo λ. Podemos solucionar a equação 3.12 iterativamente, como segue:

1. Inicialização

$$\beta_T(i) = 1 \qquad 1 \leq i \leq N.$$

2. Iteração

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j) \quad \begin{matrix} t = T-1, T-2, \ldots, 1 \\ 1 \leq i \leq N \end{matrix}$$

A inicialização do passo 1 define arbitrariamente todas as variáveis $\beta_T(i)$ como sendo 1, para todos os estados *i*. No passo 2 mostra que para termos estado no estado *i* no tempo *t*, e para ser contabilizada a sequência de observação do tempo *t+1,* devemos considerar todos os possíveis estados *j*, no tempo *t+1*, contabilizando a transição do estado *i* para o estado *j* (termo a_{ij}), bem como a observação o_{t+1} no estado *j* (termo $b_j(o_{t+1})$), para então ser contabilizado a sequencia de observação parcial do estado *j* (termo $\beta_{t+1}(j)$). Ambos os algoritmos descritos são fundamentais para se resolver os problemas 2 e 3 de uma *HMM*.

3.2.5 – Solução para o Problema 2 – Sequência de estados "ótima"

Diferentemente do problema 1, em que uma solução exata pode ser encontrada, o problema 2 tem diversas soluções possíveis, pois neste problema deve-se encontrar a sequência "ótima" de estados dada uma sequência de observação. A principal dificuldade deste problema está em se definir o que seria uma sequência "ótima" de estados, visto que diversos podem ser os critérios para isso. Uma solução que pode ser utilizada é, por exemplo, escolher os q_t estados individuais que são mais prováveis, a cada tempo *t*. Para se implementar esta solução podemos definir uma variável de probabilidade, que indica a probabilidade de estar no estado *i* no tempo *t*, dado o modelo λ e a sequência de observações *O*, como segue:

$$\gamma_t(i) = P(q_t = i|O, \lambda) \tag{3.13}$$

Podemos expressar a equação 3.13 de outra forma também:

$$\begin{aligned}\gamma_t(i) &= P(q_t = i|O,\lambda) \\ &= \frac{P(O, q_t = i|\lambda)}{P(O|\lambda)} \\ &= \frac{P(O, q_t = i|\lambda)}{\sum_{i=1}^{N} P(O, q_t = i|\lambda)}\end{aligned} \tag{3.14}$$

Como $P(O, q_t = i|\lambda)$ é igual a $\alpha_t(i)\beta_t(i)$ então a equação 3.14 fica:

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^{N} P(O, q_t = i|\lambda)} \tag{3.15}$$

Usando a variável $\gamma_t(i)$ podemos solucionar para um estado individualmente mais provável q_t^* no tempo *t*, como:

$$q_t^* = \arg\max[\,\gamma_t(i)], \qquad \begin{matrix} 1 \le t \le T \\ 1 \le i \le N \end{matrix} \tag{3.16}$$

A equação 3.16 maximiza o número de estados corretos esperados, escolhendo os mais prováveis, porém esta equação pode resultar em uma sequência de estados com alguns problemas. Caso a *HMM* tenha probabilidades de transição de estados nulas, a sequência de estados "ótima" pode ser uma sequência não válida. Isto se da, pois a equação 3.16 determina o estado mais provável a cada instante, sem se preocupar com a probabilidade de ocorrência da sequência de estados.

A solução mais utilizada para resolver este problema, é encontrar a melhor e única sequência de estados, que maximiza $P(q|O,\lambda)$. A técnica que realiza tal ação é baseada em métodos de programação dinâmicos, e é chamada de algoritmo de Viterbi.

3.2.5.1 – O Algoritmo de Viterbi

Para se encontrar a melhor e única sequência de estados, dada uma sequência de observações, precisamos definir a quantidade (probabilidade):

$$\delta_t(i) = \max_{q_1,q_2,\ldots,q_{t-1}} P[q_1 q_2 \ldots q_{t-1}, q_t = i, o_1 o_2 \ldots o_t|\lambda] \quad (3.17)$$

A equação 3.17 é a maior probabilidade ao longo de um único caminho, no tempo *t*, que vai desde as primeiras *t* observações até o estado i. Tem-se também que:

$$\delta_{t+1}(j) = [\max_i \delta_t(i)a_{ij}] \cdot b_j(o_{t+1}) \quad (3.18)$$

Para que possamos obter a sequência de estados, precisamos acompanhar o argumento que maximiza a equação 3.18, para cada *t* e *j*. Fazemos isso através do vetor $\psi_t(j)$.

1. Inicialização

$$\delta_1(i) = \pi_i b_i(o_1) \qquad 1 \le i \le N$$
$$\psi_1(j) = 0$$

2. Recursão

$$\delta_t(j) = [\max_{1\le i\le N} \delta_{t-1}(i)a_{ij}] \cdot b_j(o_t), \qquad \begin{matrix} 2 \le t \le T \\ 1 \le j \le N \end{matrix}$$
$$\psi_1(j) = \arg\max_{1\le i\le N}[\delta_{t-1}(i)a_{ij}] \qquad \begin{matrix} 2 \le t \le T \\ 1 \le j \le N \end{matrix}$$

3. Término

$$P^* = \max_{1\le i\le N}[\delta_T(i)]$$
$$q_T^* = \arg\max_{1\le i\le N}[\delta_T(i)]$$

Pode-se notar que o algoritmo de Viterbi é muito parecido com a implementação do procedimento *forward*, usado para solucionar o problema 1. A principal diferença está no passo 2, onde a maximização é feita através dos estados anteriores, a qual é usado no lugar da somatória no passo 2 do procedimento *forward*. Assim a estrutura do algoritmo Viterbi pode ser implementada da mesma forma que o procedimento *forward*.

3.2.6 –Solução para o Problema 3 – Estimação de Parâmetros

Este problema, é geralmente visto como sendo o mais complicado de se implementar, pois consiste em ajustar os parâmetros da *HMM* (A,B,π) de maneira satisfazer um determinado critério de otimização. Como não existe um método direto para se obter os parâmetros que maximizam a probabilidade de uma observação, utiliza-se o método iterativo de Baum-Welch, que determina os parâmetros da *HMM* mais prováveis localmente, ou seja, de acordo com uma certa sequência de observação. Este problema também é conhecido como problema de treinamento, pois os parâmetros da *HMM* são ajustados durante a etapa de treinamento da mesma.

Para descrevermos o procedimento proposto por Baum-Welch, iremos inicialmente determinar $\xi_t(i,j)$, como sendo a probabilidade de se estar no estado *i*, no tempo *t*, e no estado *j* no tempo *t+1*.

$$\xi_t(i,j) = P(q_t = i, q_{t+1} = j | O, \lambda) \tag{3.19}$$

Para satisfazer a as condições requeridas pela equação 3.19, iremos utilizar as definições das variáveis *forward* e *backward*, discutidas anteriormente.

$$\begin{aligned} \xi_t(i,j) &= \frac{P(q_t = i, q_{t+1} = j | O, \lambda)}{P(O|\lambda)} \\ &= \frac{\alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{P(O|\lambda)} \\ &= \frac{\alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)} \end{aligned} \tag{3.20}$$

Como a variável $\gamma_t(i)$ é a probabilidade de se estar em um estado *i*, no tempo *t*, podemos relaciona-la com a variável $\xi_t(i,j)$, como sendo:

$$\gamma_t(i) = \sum_{j=1}^{N} \xi_t(i,j) \tag{3.21}$$

A interpretação da equação 3.21 pode ser feita da seguinte maneira: se a variável $\gamma_t(i)$ ao longo do tempo *t*, nos indica o numero esperado de transições do estado *i*, em uma sequência de observação *O*, então a variável $\xi_t(i,j)$ nos indica o numero esperado de transições do estado *i* para o estado *j*, em uma sequência de observações *O*. Este problema também é conhecido como problema de treinamento, visto que é nesta etapa em que os parâmetros da *HMM* são estabelecidos.

Assim com esta definição conceitual e com a utilização das equações já discutidas, podemos obter um método para se reestimar os parâmetros da *HMM,* portanto obtem-se então:

$$\bar{\pi}_i = numero\ de\ vezes\ esperado\ para\ o\ estado\ i, \qquad (t=1) \Rightarrow \pi_1 = \gamma_1(i)$$

$$\bar{a}_{ij} = \frac{numero\ esperado\ de\ transições\ do\ estado\ i\ para\ o\ estado\ j}{numero\ esperado\ de\ transições\ do\ estado\ i}$$

$$= \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

$$\bar{b}_j(k) = \frac{numero\ esperado\ de\ vezes\ para\ o\ estado\ j\ e\ da\ observação\ \ v_k}{numero\ esperado\ de\ vezes\ parao\ estado\ j}$$

$$= \frac{\sum_{t=1}^{T} \gamma_t(i)}{t \in O_t = k}$$

Este algoritmo pode ser resumido da seguinte maneira:

1. Iniciam-se valores arbitrários para o modelo λ;
2. Utilizar o modelo λ, e a sequência *O* para calcular as equações 3.19 e 3.20;
3. Reestimar o modelo $\bar{\lambda}$, usando o algoritmo de Baum-Welch;
4. Se $P\left(O\middle|\bar{\lambda}\right) - P(O|\lambda) > \varepsilon$, então volte para o passo 2, caso contrário termina-se o algoritmo.
5. Repetir os passos 1 ao 4 com diversos valores iniciais de λ, de modo a obter um máximo local para $P(O|\lambda)$;

3.2.7 – Tipos de *HMMs*

Até agora em todas as discussões realizadas neste capitulo, usamos a estrutura mais clássica para a *HMM*, em tal estrutura todos os estados podem ser alcançados a partir de qualquer estado, esta estrutura é chamada de modelo totalmente conectado. Porém em há certos tipos de aplicações em que as propriedades de outros modelos de *HMM* são mais favoráveis para o tipo de sinal a ser usado. Por exemplo, a estrutura mostrada na Figura 3.5b, chamada de modelo *left-right*, é mais apropriada para sinais de voz, pois conforme o tempo aumenta, aumenta também o índice do estado, ou seja, o sistema vai da esquerda para a direita. Assim para sinais que variam com o passar do tempo, como sinais de voz, esta estrutura é melhor sucedida do que outras.

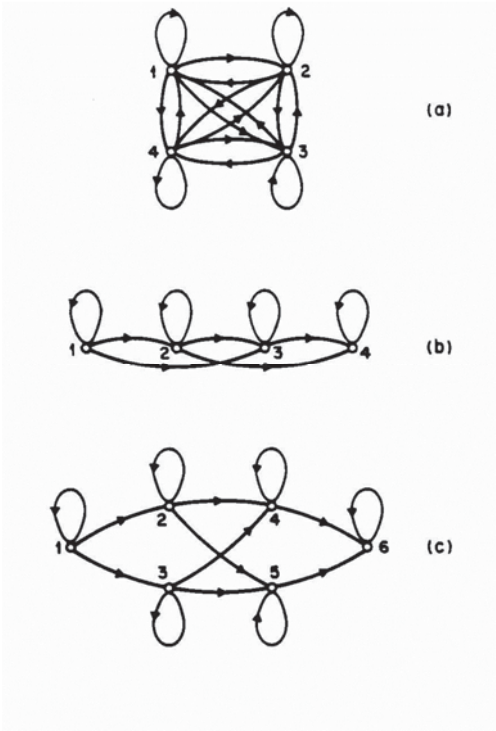


Figura 3.5: *(a) Modelo completamente conectado; (b) Modelo left-right; (c) Dois modelos left-right acoplados em cruzamento. (RABINER, L.R.,1993)*

As principais propriedades do modelo *left-right* são descritas nas equações a seguir:

$$a_{ij} = 0, \qquad j < i \tag{3.22}$$

$$\pi_i = \begin{cases} 0, & i \neq 1 \\ 1, & i = 1 \end{cases} \tag{3.23}$$

Muitos outros modelos de *HMM* podem ser utilizados, como o modelo exemplificado na Figura 3.5c, o qual consiste de dois modelos *left-right* concectados um ao outro em certo estado, este modelo pode oferecer algum tipo de flexibilidade dependendo do sistema. Deve-se ficar bem claro também que as imposições do modelo *left-right* descritas pelas equações 3.22 e 3.23, não tem efeito na reestimação dos parâmetros da *HMM*, visto que o conjunto de parâmetros iniciais nulos, permanecem nulos ao longo do processo de reestimação.

3.3 – Análise LPC

Existem diversas técnicas que podem realizar o pré-processamento do sinal de voz, como a *MFCC* e a *LPC*. Discutiremos agora processador *LPC*, pois este consiste de etapas que exigem menos processamento computacional do que a técnica de *MFCC*.

O modelo *LPC* diz que toda a amostra atual, de um sinal digital, pode ser aproximada para uma combinação linear das *p* amostras passadas do sinal digital, assim sendo temos,

$$s(n) \approx a_1 s(n-1) + a_2 s(n-2) + \cdots + a_p s(n-p) \tag{3.24}$$

onde os coeficientes $a_1, a_2, \ldots, a_p$ são assumidos como constantes ao longo do sinal de voz. Se adicionarmos uma excitação ao sinal e convertermos para o domínio-z, termos a seguinte função de transferência:

$$H(z) = \frac{S(z)}{GU(z)} = \frac{1}{1 - \sum_{i=1}^{p} a_i z^{-i}} = \frac{1}{A(z)} \tag{3.25}$$

A interpretação da equação 3.25, nos mostra como o modelo *LPC*, consegue descrever um sinal de voz. A excitação *u(n)*, em um sinal de voz, é um trem de pulsos quase periódico (para fala articulada) e um sinal aleatório (para fala não articulada). Assim, um sinal de voz, é produzido a partir do chaveamento entre um trem de pulsos, e um sinal aleatório. Ambos são multiplicados por um determinado ganho *G*, que então são usados como entrada de um filtro digital, o qual é controlado pelos parâmetros característicos do trato vocal do sinal de voz a ser produzido. Todos os parâmetros usados neste modelo variam lentamente com o tempo. A Figura 3.6 mostra o modelo *LPC* em diagrama de blocos.

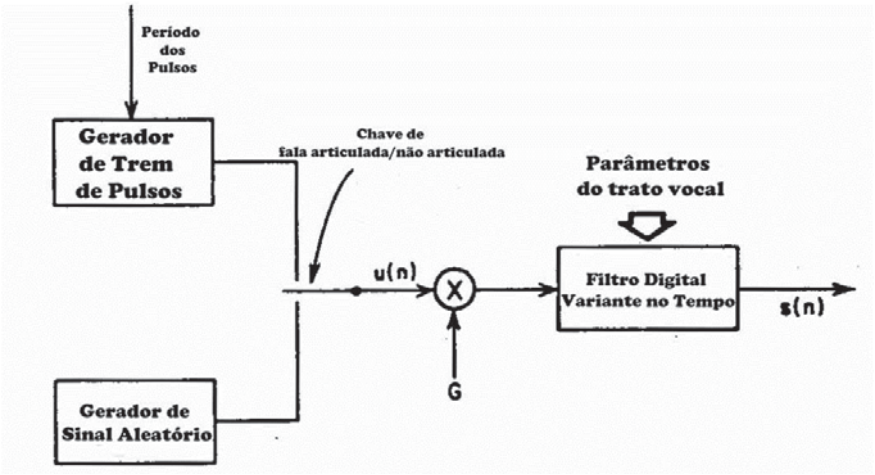


Figura 3.6: *Síntese de sinal de voz baseado no modelo LPC. (RABINER, L.R.,1993)*

Para se determinar os coeficientes *LPC*, dois são os métodos conhecidos, o da autocorrelação e o da covariância. Este último, porém, é pouco utilizado na análise de sinais de voz usando o modelo *LPC*. No método de autocorrelação, a função de autocorrelação é definida por:

$$r_n(i-k) = \sum_{m=0}^{N-1-(i-k)} s_n(m)s_n(m+i-k) \tag{3.26}$$

Como a função de autocorrelação é simétrica, ou seja, $r_n(-k)=r_n(k)$, as equações LPC podem ser expressas, por:

$$\sum_{k=1}^{p} r_n(|i-k|)\hat{a}_k = r_n(i), \qquad 1 \le i \le p \tag{3.27}$$

A equação 3.27 pode ser expressa na forma matricial como segue:

$$\begin{bmatrix} r_n(0) & r_n(1) & r_n(2) & \dots & r_n(p-1) \\ r_n(1) & r_n(0) & r_n(1) & \dots & r_n(p-2) \\ r_n(2) & r_n(1) & r_n(0) & \cdots & r_n(p-3) \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ r_n(p-1) & r_n(p-2) & r_n(p-3) & \cdots & r_n(0) \end{bmatrix} \begin{bmatrix} \hat{a}_1 \\ \hat{a}_2 \\ \hat{a}_3 \\ \vdots \\ \hat{a}_p \end{bmatrix} = \begin{bmatrix} r_n(1) \\ r_n(2) \\ r_n(3) \\ \vdots \\ r_n(p) \end{bmatrix} \tag{3.28}$$

A equação 3.28 é uma matriz *pxp* de Toeplitz, ou seja, simétrica com todos os elementos diagonais iguais. Esta matriz pode ser solucionada por diversas maneiras, uma bastante usada em sistemas RAV é o algoritmo de Durbin.

A influência no número de coeficientes *LPC* na análise espectral do sinal de voz, é mostrada na Figura 3.7.

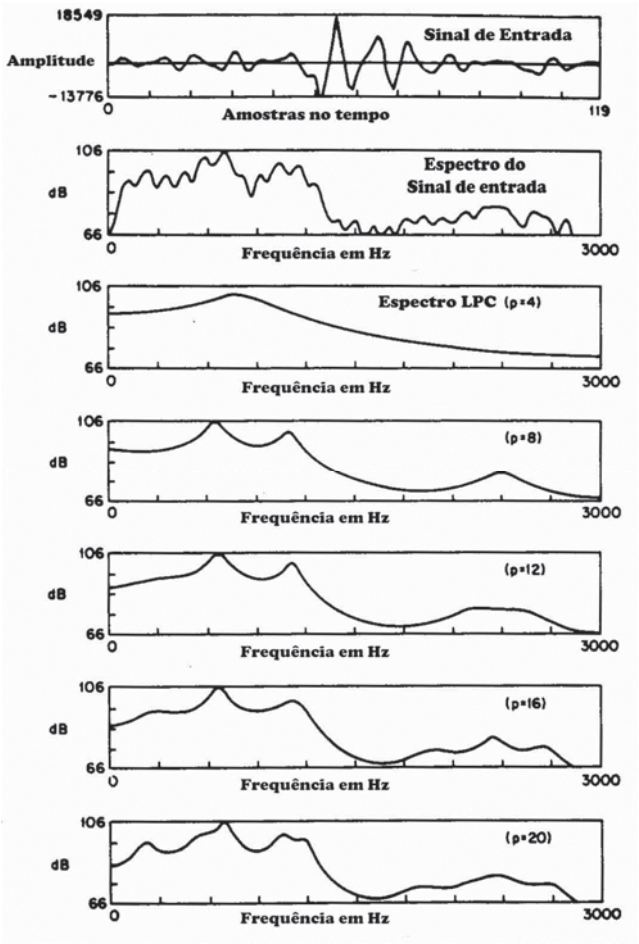


Figura 3.7: *Influência dos coeficientes LPC no espectro de frequência. (RABINER, L.R.,1993)*

Como se pode ver, quanto maior for o número de coeficientes *p* mais fiel o espectro de frequência do sinal predito pelo modelo *LPC* será do espectro do sinal de voz original. Porém estudos experimentais chegaram à conclusão que valores para *p* entre 8 e 12 são suficientes para se fazer uma análise de reconhecimento de voz, de maneira satisfatória, para a maioria das aplicações. Para valores de *p* acima desta faixa, o espectro de frequência do modelo *LPC* passa a ter mais componentes harmônicas, se aproximando ainda mais do espectro do sinal original, porém estas componentes não refletem em ressonâncias ou antirressonâncias relevantes do sinal de voz original.

3.3.1 – O processador *LPC*

Um diagrama de blocos de um processador *LPC* é mostrado na Figura 3.8. Tal processador é dividido em duas partes, a etapa responsável por realizar a pré-ênfase, e etapa responsável por realizar a extração de parâmetros.

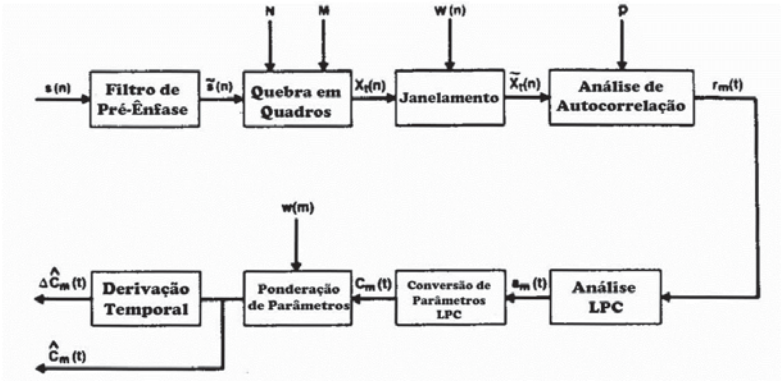


Figura 3.8: *Processador LPC. (RABINER, L.R.,1993)*

A etapa de pré-ênfase é constituída pelos blocos: Filtro de Pré-Ênfase, Quebra em Quadros e Janelamento. Sendo os demais blocos constituintes da etapa de extração de parâmetros.

1. **Filtro de Pré-Ênfase:** Consiste de um filtro digital de baixa ordem (filtro tipicamente de primeira ordem, do tipo *FIR*). Este filtro é um equalizador de espectro, cuja função é tornar o espectro mais igualitário e fazer com que o sinal seja menos susceptível à efeitos de precisão finita ao longo do processamento. A função de transferência do filtro, bem como sua equação de diferenças, são dadas pelas equações a seguir:

$$H(z) = 1 - az^{-1}, \qquad 0{,}9 \leq a \leq 1{,}0 \tag{3.29}$$

$$s(n) = s(n) - as(n-1) \tag{3.30}$$

   O valor mais comumente usado para a constante *a* é de 0,95. Para aplicações que utilizam ponto fixo, como a utilizada neste trabalho, utiliza-se *a=15/16=0,9375.*

2. **Quebra em Quadros:** Nesta etapa o sinal de voz é quebrado em quadros de *N* amostras, sendo que os quadros adjacentes são separados por *M* amostras. Geralmente utiliza-se *M=(1/3)N*, como pode ser visto na Figura 3.9, caso *M<<N* o espectro estimado

pelo modelo *LPC* será bastante suave de quadro para quadro. Porém se $M > N$ não haverá sobreposição dos quadros e, portanto, poderá haver a total perda de informação em alguma parte do sinal de voz. Valores típicos para os quadros são de 20-30ms do sinal de voz, assim se tivermos uma frequência de amostragem de 8kHz, teremos um quadro com *N=240* e *M=80*.

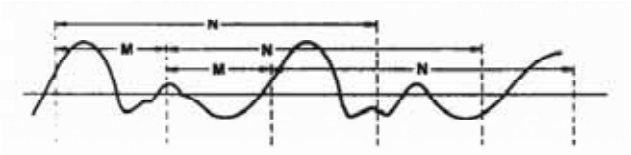


Figura 3.9: *Divisão do sinal de voz em quadros. (RABINER, L.R.,1993)*

3. **Janelamento:** Aqui cada quadro do sinal de voz é multiplicado por uma função-janela para se minimizar as descontinuidades do quadro, tanto no começo como no fim. Para isso a função-janela mais utilizada no método de autocorrelação *LPC*, é a função-janela de Hamming. A janela de Hamming, faz com que as bordas do quadro tendam a zero de maneira suave. A forma da janela de Hamming é descrita na equação 3.31.

$$w(n) = 0{,}54 - 0{,}46\cos\left(\frac{2\pi n}{N-1}\right), \qquad 0 \le n \le N-1 \tag{3.31}$$

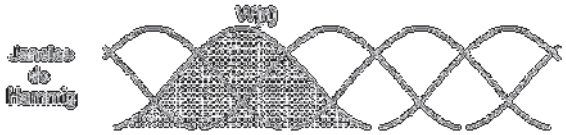


Figura 3.10: *Janelas de Hamming em cada quadro. (CIPRIANO, J.L.G. 2001)*

4. **Análise de Autocorrelação:** Para cada um dos quadros após a etapa de janelamento, é calculado, através da equação abaixo, os coeficientes de autocorrelação das amostras dos quadros. Onde *p* é a ordem da análise *LPC*.

$$r_l(m) = \sum_{n=0}^{N-1-m} \tilde{x}_l(n)\tilde{x}_l(n+m), \qquad m = 0,1,\dots,p \tag{3.32}$$

5. **Análise LPC:** Nesta etapa cada um dos coeficientes de autocorrelação é transformado em coeficientes *LPC*, ou em qualquer outro coeficiente de interesse para a

análise espectral, como por exemplo coeficientes de reflexão (PARCOR) ou coeficientes cepstrais. O método convencional para se transformar os coeficientes de autocorrelação em coeficientes *LPC*, é chamado de algoritmo de Durbin, descrito abaixo.

$$E^{(0)} = r(0)$$

$$k_i = \left\{ r(i) - \sum_{j=1}^{L-1} \alpha_j^{(i-1)} r(|i-j|) \right\} \Big/ E^{(i-1)}, \qquad 1 < i < p$$

$$\alpha_i^{(i)} = k_i$$

$$\alpha_i^{(i)} = \alpha_j^{(i-1)} - k_i \alpha_{i-j}^{(i-1)}$$

$$E^{(i)} = (1 - k_i^2) E^{(i-1)}$$

Como solução final após as *p* iterações, tem-se os seguintes coeficientes:

$$a_m = coeficientes\ LPC = \alpha_m^{(p)} \qquad 1 \le m \le p$$

$$k_m = coeficientes\ PARCOR$$

$$g_m = coeficientes\ de\ taxa\ de\ área\ logaritmica = \log\left(\frac{1-k_m}{1+k_m}\right)$$

6. **Conversão de parâmetros *LPC*:** Aqui novamente é realizada uma conversão de parâmetros. Os parâmetros *LPC* calculados anteriormente são convertidos em parâmetros cepstrais. Os coeficientes cepstrais são os coeficientes da transformada de Fourier da representação logarítmica de magnitude do espectro. Estes coeficientes se mostraram ser mais robustos e confiáveis para análise de sinais de voz, do que os outros coeficientes. O número de coeficientes cepstrais *Q*, é geralmente feito como *Q*>*p*, usualmente tomando $Q \cong (3/2)p$. O algoritmo de conversão é mostrado abaixo. O termo σ^2 é o ganho do modelo *LPC*.

$$c_0 = \ln \sigma^2$$

$$c_m = a_m + \sum_{k=1}^{m-1} \left(\frac{k}{m}\right) c_k a_{m-k,} \qquad 1 \le m \le p$$

$$c_m = \sum_{k=1}^{m-1} \left(\frac{k}{m}\right) c_k a_{m-k}, \qquad m > p$$

7. **Ponderação de Parâmetros:** Os coeficientes cepstrais de baixa ordem são sensíveis às inclinações gerais do espectro, e os de alta ordem são sensíveis à ruído. Assim tornou-se um padrão realizar uma ponderação dos coeficientes cepstrais através de uma função janela cônica, afim de minimizar essas sensibilidades. Considerando a definição dos coeficientes cepstrais, e diferenciando-a na frequência, tem-se:

$$\log\left|S\left(e^{j\omega}\right)\right| = \sum_{m=-\infty}^{\infty} c_m e^{-j\omega m} \tag{3.33}$$

$$\frac{\partial}{\partial \omega}\left[\log\left|S\left(e^{j\omega}\right)\right|\right] = \sum_{m=-\infty}^{\infty} (-jm) c_m e^{-j\omega m} \tag{3.34}$$

Como na forma diferencial cada inclinação fixa do espectro em log de magnitude se torna uma constante, e cada pico prominente em log de magnitude, por exemplo, sons de consoantes, se mantém conservado como um pico, então podemos usar o parâmetro $(-jm)$ como uma forma de ponderação.

$$\hat{c}_m = (-jm) c_m \tag{3.35}$$

Para se atingir maior robustez para valores de m muito grandes, e para truncar o cálculo infinito da equação 3.34, utilizamos o filtro passa-faixa no domínio cepstral, o qual trunca o cálculo e realiza a de-ênfase do coeficiente cepstral em torno de *m=1* e *m=Q* .

$$\hat{c}_m = w_m c_m \qquad 1 \le m \le Q \tag{3.36}$$

$$w_m = \left[1 + \frac{Q}{2}\sin\left(\frac{\pi m}{Q}\right)\right], \qquad 1 \le m \le Q \tag{3.37}$$

8. **Derivação Temporal:** Esta etapa é utilizada quando se deseja obter uma melhor representação espectral local do espectro de voz. Assim a representação cepstral pode ser extendida para a análise sobre a informação das derivadas cepstrais temporais, ambas as derivadas de primeira e segunda ordem incrementaram a performance de sistemas RAV. Os detalhes a respeito desta etapa podem ser encontrados em RABINER, L.R.,1993.

3.4 Quantização Vetorial

Após a análise *LPC*, teremos um conjunto de vetores v_l ,$l = 1,2,...,L$, onde cada vetor é um vetor de dimensão p. Comparando com o sinal antes da análise *LPC*, há uma significativa compressão da informação, por exemplo se usarmos uma taxa de amostragem de 10kHz, com 16 bits de quantização, teremos uma taxa de transferência de bit de 160kbps. Após a análise *LPC* se tivermos um sinal com 100 vetores espectrais por segundo, sendo cada um de dimensão $p=10$ e 16 bits de quantização teremos uma taxa de de transferência de bits de 16kbps, ou seja, a compressão da informação foi de 10:1. Se pensarmos que cada palavra é constituída por um conjunto de fonemas, então seria melhor se tivéssemos uma única representação espectral para cada quadro da informação de voz, ao invés de um vetor de dimensão p. É nisto em que se baseia a idéia básica por trás das técnicas de quantização vetorial, ou seja, representar um conjunto de fonemas através de um conjunto de códigos de palavras, chamado de *codebook*. Por exemplo, se desejarmos ter um *codebook* com 1024 representações necessitaríamos de 10 bits de quantização, e se considerarmos uma palavra com 100 vetores espectrais, teríamos uma taxa de transferência de bit de aproximadamente 1kbps, portanto a compressão agora seria de 16:1. Isto mostra que a quantização vetorial diminui a quantidade de informação a ser armazenada.

Algumas vantagens desta técnica podem ser citadas, como: redução da quantidade de informação armazenada; redução da quantidade de cálculos necessários para determinar a similaridade entre pares de vetores; representação discreta de sinais de voz, através da associação de um conjunto de índices à um conjunto de fonemas. As desvantagens relacionadas à quantização vetorial são: uma inerente distroção espectral na representação do vetor que está sendo analisado; a quantidade de memória necessária para o armazenamento dos vetores do *codebook* é não trivial, ou seja, quanto maior for o *codebook* (para se reduzir o erro de quantização), maior será a quantidade de memória necessária para armazenar as entradas do *codebook*.

3.4.1 – Elementos de implementação da quantização vetorial

Para se realizar a etapa de Quantização Vetorial, necessitamos implementar os blocos presentes na Figura 3.11.

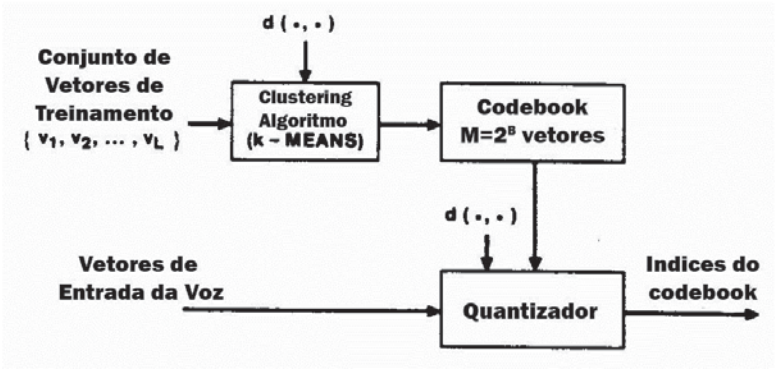


Figura 3.11: *Diagrama de Blocos da Quantização Vetorial. (RABINER, L.R.,1993)*

Inicialmente precisa-se de um grande conjunto de vetores espectrais a fim de compor um conjunto de treinamento. Como o tamanho do *codebook* é dado por $M=2^B$, então necessita-se de um conjunto de L vetores espectrais, sendo que $L>>M$ para se realizar o treinamento. Geralmente faz-se $L=10M$, para que assim o *codebook* formado possa representar a variabilidade espectral observada no conjunto de vetores de treinamento.

Em seguida deve-se também realizar o cálculo da similaridade, ou distância, entre os pares de vetores, para que possa ser possível realizar o agrupamento (*clustering*) do conjunto de vetores de treinamento, bem como associar ou classificar arbitrariamente vetores espectrais à uma única entrada de *codebook*.

Para que possamos gerar os M indices do *codebook* a partir de M grupos (*clusters*), calcula-se o centróide de cada grupo. Esse centróide é então o índice a ser usado no *codebook*.

O procedimento de classificação de um vetor espectral em um índice do *codebook* é na verdade um processo de quantização, ou seja, para um determinado vetor espectral de entrada, tem-se como saída um índice do *codebook* que mais se aproxima do calculado usando os vetores de entrada.

Assim ao final desta etapa teremos um único vetor de índices caracterizando o sinal de voz. Cada índice deste vetor é uma observação para a *HMM*, assim esse vetor é a entrada da *HMM*, a qual irá calcular a probabilidade de gerar a sequência de observação, ou seja, o vetor de índices. Caso essa probabilidade for baixa, então o vetor de índices não representa a

palavra para qual a *HMM* foi treinada, caso a probabilidade for alta então o vetor de índices representa a palavra para qual a *HMM* foi treinada, ou seja, a palavra foi reconhecida pelo sistema. Assim termina-se o processo de reconhecimento automático de voz.

4 – O SISTEMA DESENVOLVIDO

4.1 – O sistema completo

No item 1.5 foi discutido, em diagrama de blocos, as etapas necessárias para a realização de um sistema de reconhecimento automático de voz. Agora mostraremos em detalhes as etapas concretizadas, desde a sua especificação e descrição dos módulos constituintes do sistema em VHDL, até a síntese dos mesmos em um componente FPGA, mostrando em alguns casos simulações e os resultados obtidos em tempo real. A Figura 4.1 mostra as etapas realizadas, em destaque, e a Figura 4.2 sintetizadas em protótipo, na qual se realizou os testes para a utilização do bloco FFT.

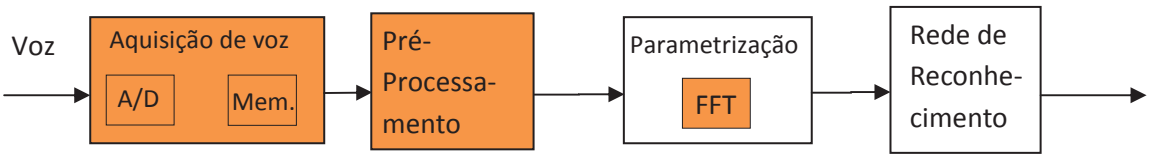


Figura 4.1: *Etapas desenvolvidas do sistema RAV. (O Autor)*

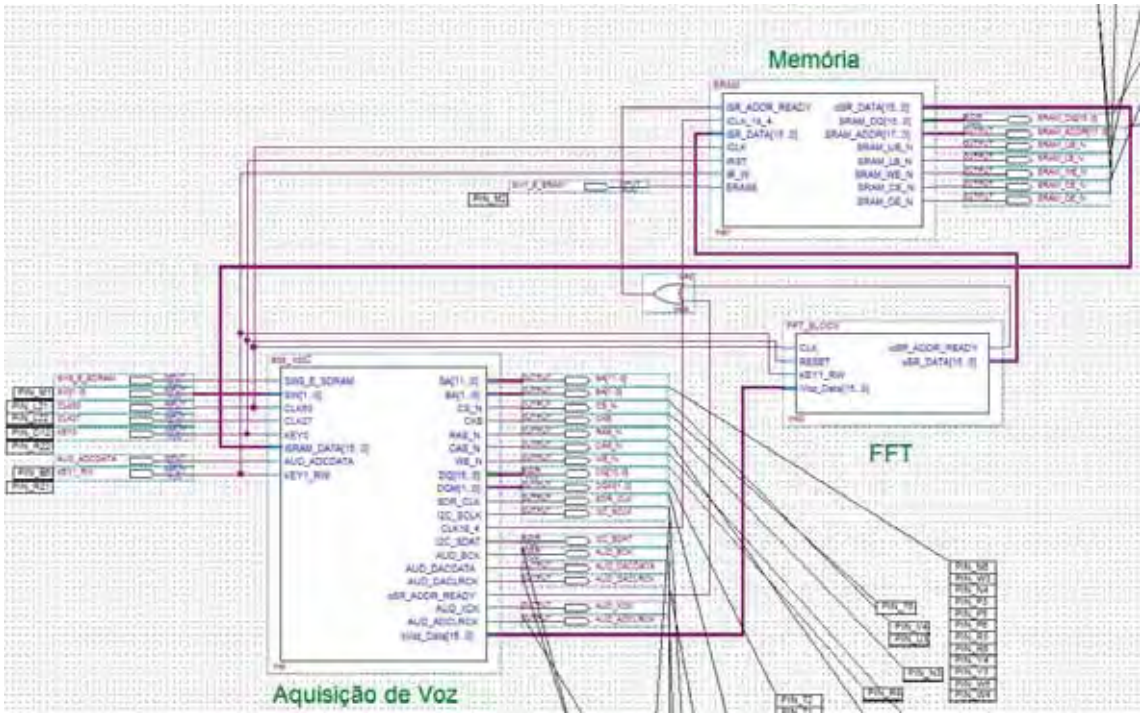


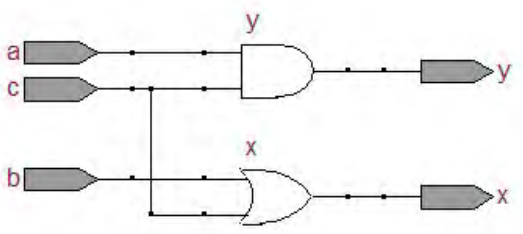
Figura 4.2: *Sistema com bloco FFT. (O Autor)*

4.1.1 – VHDL – VHSIC Hardware Description Language

Este subitem foi fundamentado na referência D'AMORE, R., 2005, onde informações mais aprofundadas e teoria de linguagem VHDL podem ser encontradas. A linguagem utilizada para descrever os módulos a serem sintetizados em um componente FPGA foi VHDL. Tal linguagem surgiu em um projeto do Departamento de Defesa dos Estados Unidos da América, denominado VHSIC – *Very High Speed Integrated Circuit*. Tal projeto consistia em desenvolver uma linguagem de descrição de circuitos padrões. O IEEE – *Institute of Electrical and Electronic Engineer,* padronizou a linguagem VHDL, em 1987, no padrão IEEE 1076-1987. Em 1993 uma nova versão deste padrão foi desenvolvida e aprovada pelo IEEE, o IEEE 1076-1993. Para se adicionar facilidades à linguagem dois padrões foram criados, o IEEE 1164 e o IEEE 1076.3. O primeiro define o pacote "Std_logic_1164" e o segundo os pacotes "Numeric_std" e "Numeric_bit". Em VHDL, um pacote ou "*package*", é um local para armazenamento de informações de uso comum, como tipos de dados, funções dentre outros. Atualmente a linguagem é periodicamente atualizada pelo IEEE.

A grande vantagem e principal característica da linguagem VHDL é que, com exceção de regiões específicas no código, todos os comandos são executados de maneira concorrente, ou seja, a mudança de estado em um sinal, leva à execução de todos os comandos sensíveis aquele sinal, da mesma forma que ocorre em um circuito digital. A linguagem permite também delimitar regiões de código sequencial, onde a execução dos comandos segue a ordem de sua inclusão no código. A Figura 4.3 mostra um exemplo da linguagem VHDL e o circuito sintetizado pela ferramenta. A região da "entidade" é onde se declaram todas as entradas e saídas do circuito a ser descrito, e na região da "arquitetura" é onde se descreve a lógica que o circuito irá realizar, ou seja, a "entidade" representa a interface e a "arquitetura" representa a funcionalidade do sistema digital proposto.

```
1  ENTITY porta IS
2      PORT (a,b,c : IN BIT;
3            x,y   : OUT BIT);
4  END porta;
5
6  ARCHITECTURE porta_or OF porta IS
7  BEGIN
8      y <= a AND c;
9      x <= c OR b;
10 END porta_or;
```



(a) *(b)*

Figura 4.3: *(a) Código em VHDL; (b) Circuito sintetizado, nível RTL. (O Autor)*

A ferramenta utilizada para a descrição do sistema é o Quartus II Web Edition v9.1sp2 da fabricante ALTERA. A ferramenta de desenvolvimento pode ser hoje adquirida gratuitamente na página, www.altera.com, da fabricante ALTERA. O ambiente de desenvolvimento Quartus II possibilita o desenvolvimento completo de um projeto digital tendo como alvo um componente programável tal como CPLD/FPGA. A ferramenta integra toda as etapas necessárias de projeto. A seguir são apresentadas e descritas as etapas fundamentais deste plano:

- **Entradas de projeto:** na forma de entrada esquemática, ou usando linguagem de programação para *hardware* tal como: VHDL, Verilog e XHDL.
- **Síntese:** compilação do projeto onde é gerado o arquivo binário de programação do componente alvo bem como o arquivo de fato que será usado no processo de simulação.
- **Simulação:** nesta etapa é necessário realizar a geração de um arquivo de vetores binário que serão usados para modelar o comportamento do sistema projetado.
- **Gravação:** nesta etapa o arquivo binário, já desenvolvido, é gravado no componente programável indicado previamente pelo usuário.

4.1.2 – Kit Altera DE-1

O kit de FPGA utilizado no projeto do sistema de reconhecimento automático de voz é o *Altera DE-1*, mostrado na Figura 4.4. Este kit foi disponibilizado pelo Departamento de Engenharia Elétrica da Universidade Estadual Paulista ”Júlio de Mesquita Filho” – Faculdade de Engenharia campus de Guaratinguetá.

Os principais componentes utilizados deste kit são:

- **FPGA Cyclone II:** neste componente será sintetizado o sistema RAV;
- **Memórias SRAM/SDRAM/FLASH:** serão utilizadas para armazenar a voz adquirida, os parâmetros extraídos da mesma, e os parâmetros de comparação;
- **Entrada Mic-In:** entrada para o microfone;
- **Saída Line-Out:** saída para se conectar um auto-falante;
- **CODEC de áudio:** usado para fazer a conversão do sinal de voz para o domínio digital;
- **Chaves e *push-bottons*:** usadas para testar e controlar os blocos desenvolvidos;
- **LEDs:** usados para verificação de algum teste realizado em certos blocos;
- **Cristais Osciladores:** os três cristais são utilizados como sinais de relógio;

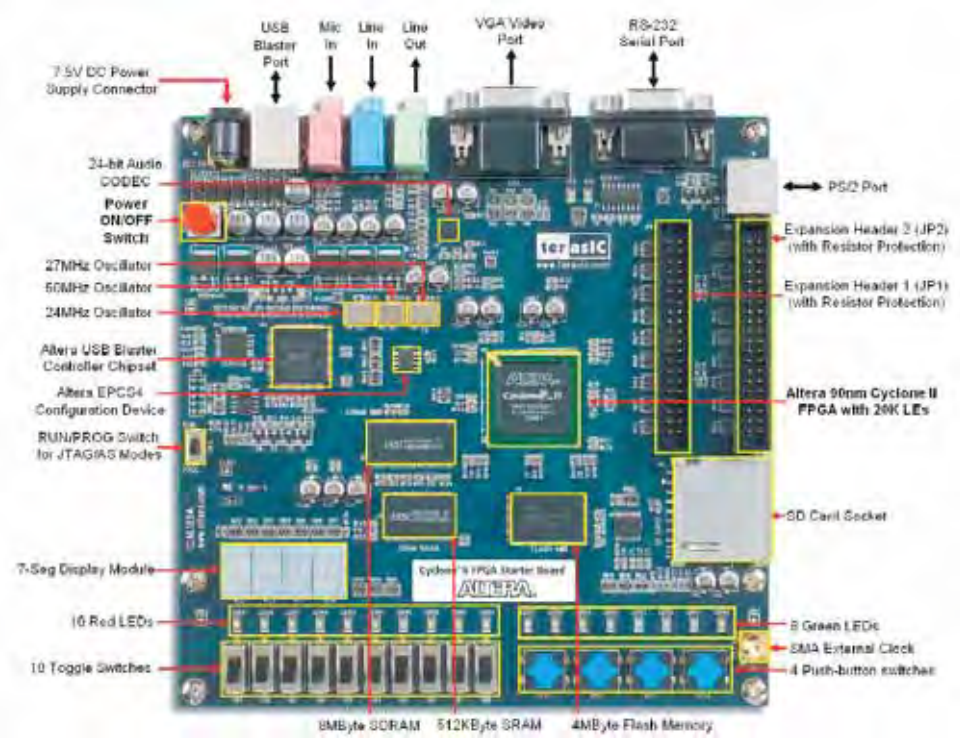


Figura 4.4: *Kit Altera Cyclone II Starter Board.*

Toda a comunicação entre a placa e a ferramenta de desenvolvimento, é feita através da porta USB, usando o driver *USB-Blaster,* fornecido pela própria ALTERA. Além de realizar a programação da FPGA, é através do cabo USB que podemos utilizar uma ferramenta extremamente poderosa do Quartus II, a *Signal Tap II Logic Analyzer*, a qual nos permite fazer a verificação dos sinais nos pinos e dos registradores internos sintetizados na FPGA em tempo real, como se fosse um analisador lógico.

4.2 – Bloco de aquisição de dados.

Este bloco, Figura 4.5, realiza todas as etapas para se adquirir o sinal de entrada de voz do locutor. As etapas de amostragem, quantização e codificação são realizadas pelo CODEC de áudio presente no kit de desenvolvimento, a ser discutido detalhadamente mais adiante. Para se gravar os dados que são gerados pelo CODEC nas memórias, foi preciso criar em linguagem de *hardware*, os blocos de comando que realizam o fluxo de dados para as memórias, gerando os sinais de controle necessários de acordo com o diagrama de tempo de cada uma. Cada um dos blocos constituintes deste sub-sistema será discutido posteriormente.

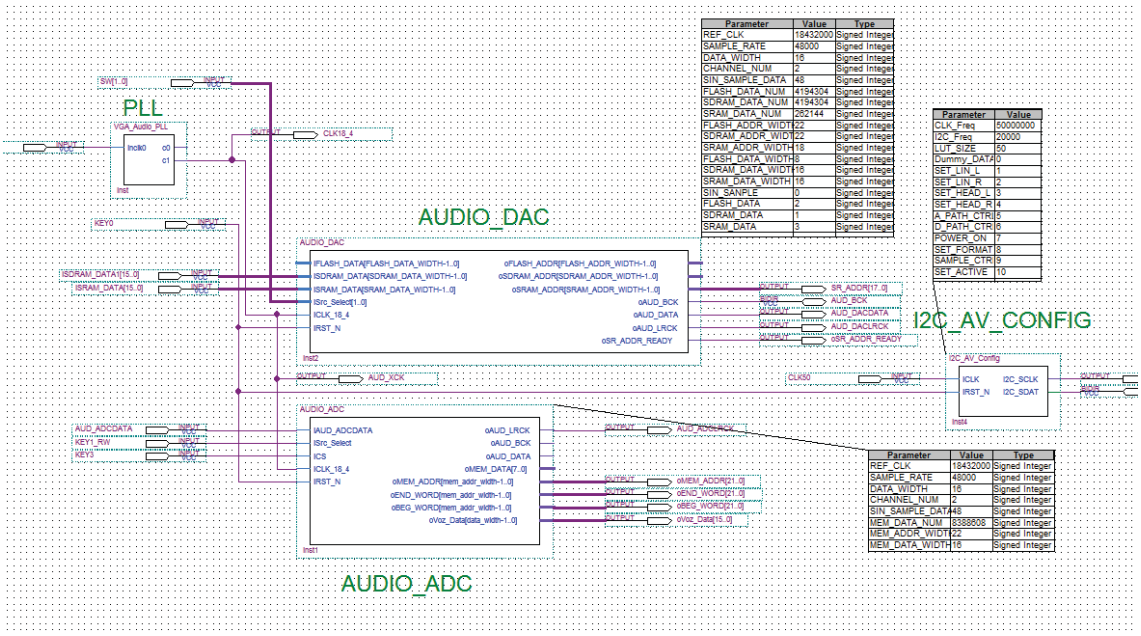


Figura 4.5: *Sub-Blocos constituintes do bloco aquisição de dados. (O Autor)*

4.2.1 – Memórias

O kit de desenvolvimento da ALTERA DE-1, vem com diversos exemplos demonstrativos dos vários recursos existentes no kit. No próprio manual do kit, encontrado no *website* da fabricante ALTERA, diz que nestes exemplos existem módulos de controle prontos de cada uma das memórias, e que estes módulos podem ser usados para que não precisemos perder tempo desenvolvendo-os em nossos projetos. Porém estes módulos estão todos escritos na linguagem de descrição de *hardware* Verilog. Como esta se trata de uma linguagem bem intuitiva, não se encontraram problemas em se entender o código, de cada um dos módulos prontos utilizados no projeto do sistema. Como o sistema está sendo descrito em VHDL para que possamos utilizar os módulos descritos em Verilog, utilizamos o recurso de Diagrama de Blocos da ferramenta Quartus II, onde neste recurso podemos compilar um projeto em que hajam blocos interligados que estão descritos em diferentes linguagens.

4.2.1.1 – Memória SDRAM

A memória SDRAM, do kit de desenvolvimento, contém 8MB de capacidade, 12 linhas de endereço com quatro bancos de endereços e 16 linhas de dados. Como o bloco pronto descrito em Verilog, possui diversas possibilidades de operação, de forma síncrona ou assíncrona, um outro bloco de controle foi desenvolvido em VHDL, para se comunicar com este

bloco pronto do kit. Os blocos estão interligados da forma que é mostrada na Figura 4.6, onde já podemos ver o recurso de Diagrama de Blocos discutido anteriormente, o qual está interligando o bloco “SDRAM_CMD”, descrito em VHDL, e o bloco “Multi_Sdram”, descrito em Verilog. A memória SDRAM foi utilizada para se armazenar a voz do locutor por ser uma memória volátil então não pode utilizá-la para guardar os parâmetros de comparação da rede de reconhecimento.

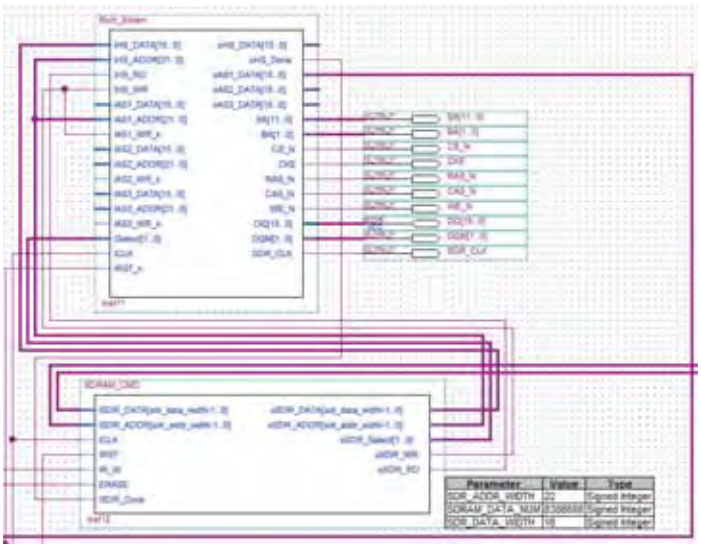


Figura 4.6: *Diagrama de Blocos do controle da SDRAM. (O Autor)*

Todos os sinais de controle e *handshaking* existentes para, se realizar uma operação de escrita ou leitura na SDRAM seguiram o diagrama de tempo presente no *datasheet* do circuito integrado da própria memória SDRAM. A Figura 4.7 mostra uma parte do código VHDL do bloco “SDRAM_CMD”, usado para gerenciar uma operação de escrita, caso em que guardamos o sinal de voz na SDRAM, e uma operação de leitura, caso em que queremos verificar o que foi armazenado na memória.

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3   USE ieee.std_logic_unsigned.ALL;
4   USE ieee.std_logic_arith.ALL;
5
6   LIBRARY work;
7
8   ENTITY SDRAM_CMD IS
9   GENERIC ( SDR_ADDR_WIDTH    : INTEGER := 22;
10            SDRAM_DATA_NUM    : INTEGER := 8388608;
11            SDR_DATA_WIDTH    : INTEGER := 16);
12  PORT (  iSDR_DATA    : IN STD_LOGIC_VECTOR (SDR_DATA_WIDTH-1 DOWNTO 0);
13          oSDR_DATA    : OUT STD_LOGIC_VECTOR (SDR_DATA_WIDTH-1 DOWNTO 0);
14          oSDR_ADDR    : OUT STD_LOGIC_VECTOR (SDR_ADDR_WIDTH-1 DOWNTO 0);
15          iSDR_ADDR    : IN STD_LOGIC_VECTOR (SDR_ADDR_WIDTH-1 DOWNTO 0);
16          iCLK,iRST    : IN STD_LOGIC;
17          iR_W         : IN STD_LOGIC;
18          ERASE        : IN STD_LOGIC;
19          oSDR_Select  : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
20          iSDR_Done    : IN STD_LOGIC;
21          oSDR_WR      : OUT STD_LOGIC;
22          oSDR_RD      : OUT STD_LOGIC);
23  END SDRAM_CMD;
24
25  ARCHITECTURE SDRAM1 OF SDRAM_CMD IS
26
27  SIGNAL mSDR_WRn          : STD_LOGIC;
28  SIGNAL mSDR_Start        : STD_LOGIC;
29  SIGNAL mSDR_ST           : INTEGER RANGE 0 TO 5;
30  SIGNAL SDRAM_Cont        : INTEGER RANGE 0 TO SDRAM_DATA_NUM;
31  SIGNAL SDRAM_Cont_Vector: STD_LOGIC_VECTOR (SDR_ADDR_WIDTH-1 DOWNTO 0);
32
33  BEGIN
34
35  --------- CICLO DE LEITURA E ESCRITA --------------
36  CICLOS : BLOCK
37  BEGIN
38      abc : PROCESS (iCLK, iRST)
91  END BLOCK CICLOS;
92  ------------------------------------------------------------
93  oSDR_WR <= (mSDR_WRn AND mSDR_Start);
94  oSDR_RD <= (NOT(mSDR_WRn) AND mSDR_Start);
95
96  END SDRAM1;
```

Figura 4.7: *Parte do código VHDL do bloco "SDRAM_CMD". (O Autor)*

4.2.1.2 – Memória SRAM

A memória SRAM do kit de desenvolvimento contém 256KB de capacidade, com 18 linhas de endereço e 16 linhas de dados. De maneira semelhante à memória SDRAM, dois blocos são utilizados para se realizar as operações de escrita e leitura na memória, Figura 4.8. O bloco "Multi_Sram" é o bloco pronto do kit, escrito em Verilog, que realiza as operações de escrita e leitura, já o bloco "SRAM_CMD", é o bloco intermediário de controle, escrito e desenvolvido em VHDL.

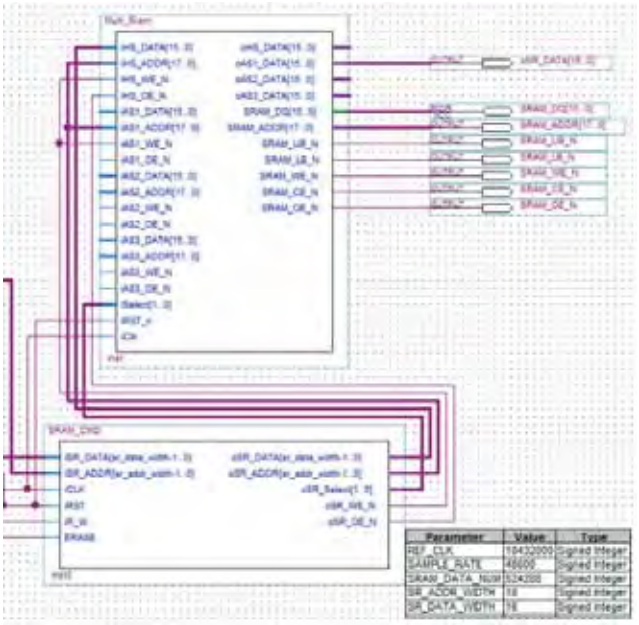


Figura 4.8: *Diagrama de Blocos de controle da SRAM. (O Autor)*

Como a memória SRAM também é volátil, então também não se pode utilizá-la para se armazenar os parâmetros de comparação da rede de reconhecimento. Portanto, como esta memória é volátil e possui baixa capacidade, a mesma foi escolhida para se armazenar os parâmetros extraídos do sinal de voz. A Figura 4.9 mostra um trecho do código VHDL do bloco "SRAM_CMD"

```
 1   LIBRARY ieee;
 2   USE ieee.std_logic_1164.all;
 3   USE ieee.std_logic_unsigned.ALL;
 4   USE ieee.std_logic_arith.ALL;
 5
 6   LIBRARY work;
 7
 8   ENTITY SRAM_CMD IS
 9   GENERIC (  REF_CLK          : INTEGER := 18432000;  --18.34MHz
10              SAMPLE_RATE      : INTEGER := 48000;     -- 48kHZ
11              SRAM_DATA_NUM    : INTEGER := 524288;    -- 512 Kbits
12              SR_ADDR_WIDTH    : INTEGER := 18;
13              SR_DATA_WIDTH    : INTEGER := 16);
14   PORT (  iSR_DATA         : IN STD_LOGIC_VECTOR (SR_DATA_WIDTH-1 DOWNTO 0);
15           oSR_DATA         : OUT STD_LOGIC_VECTOR (SR_DATA_WIDTH-1 DOWNTO 0);
16           oSR_ADDR         : OUT STD_LOGIC_VECTOR (SR_ADDR_WIDTH-1 DOWNTO 0);
17           iSR_ADDR         : IN STD_LOGIC_VECTOR (SR_ADDR_WIDTH-1 DOWNTO 0);
18           iCLK,iRST        : IN STD_LOGIC;
19           iR_W             : IN STD_LOGIC;
20           ERASE            : IN STD_LOGIC;
21           oSR_Select       : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
22           oSR_WE_N         : OUT STD_LOGIC;
23           oSR_OE_N         : OUT STD_LOGIC);
24   END SRAM_CMD;
25
26   ARCHITECTURE SRAM1 OF SRAM_CMD IS
27
28   SIGNAL DATA_Tmp          : STD_LOGIC_VECTOR (15 DOWNTO 0);
29   SIGNAL mSR_WRn           : STD_LOGIC;
30   SIGNAL mSR_Start         : STD_LOGIC;
31   SIGNAL mSR_ST            : INTEGER RANGE 0 TO 5;
32   SIGNAL f_SRAM            : STD_LOGIC;
33   SIGNAL SRAM_Cont         : INTEGER RANGE 0 TO SRAM_DATA_NUM;
34   SIGNAL SRAM_Cont_Vector: STD_LOGIC_VECTOR (SR_ADDR_WIDTH-1 DOWNTO 0);
35
36   BEGIN
37
38   --------- CICLO DE LEITURA E ESCRITA --------------
39   CICLOS : BLOCK
40   BEGIN
41       abc : PROCESS (iCLK, iRST)
100  END BLOCK CICLOS;
101  ------------------------------------------------------------
102  oSR_WE_N <= NOT(mSR_WRn AND mSR_Start);
103  oSR_OE_N <= NOT(NOT(mSR_WRn) AND mSR_Start);
104
105  END SRAM1;
```

Figura 4.9: *Parte do código VHDL do bloco "SRAM_CMD". (O Autor)*

4.2.1.3 – Memória FLASH

A memória FLASH presente no kit de desenvolvimento, contém 4MB de capacidade, 22 linhas de endereço e 8 linhas de dados. Assim como nas outras memórias, utilizamos o mesmo procedimento para se realizar o controle da memória FLASH, Figura 4.10. O bloco "Multi_Flash" é o bloco pronto, escrito em Verilog, que se comunica diretamente com a memória FLASH, e o bloco "FLASH" é o bloco de controle intermediário escrito e desenvolvido em VHDL.

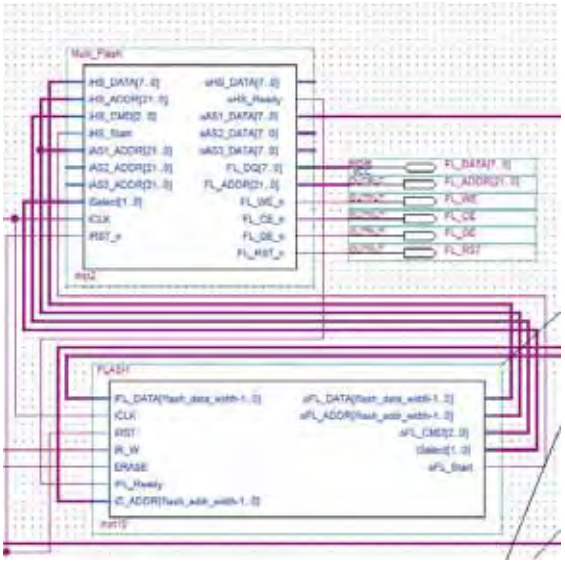


Figura 4.10: *Diagrama de Blocos de controle da memória FLASH. (O Autor)*

A principal característica da memória FLASH, é o fato da mesma ser não-volátil, ou seja, o seu conteúdo não é apagado quando se retira a sua alimentação. Assim por esse motivo esta memória foi escolhida para se armazenar os parâmetros de comparação da rede de reconhecimento, bem como a função-janela de Hamming. A Figura 4.11 mostra um trecho do código VHDL do bloco “FLASH”.

```
 1   LIBRARY ieee;
 2   USE ieee.std_logic_1164.all;
 3   USE ieee.std_logic_unsigned.ALL;
 4   USE ieee.std_logic_arith.ALL;
 5
 6   LIBRARY work;
 7
 8  ENTITY FLASH IS
 9  GENERIC ( FLASH_ADDR_WIDTH  : INTEGER := 22;
10            FLASH_DATA_WIDTH  : INTEGER := 8);
11  PORT (  iFL_DATA     : IN STD_LOGIC_VECTOR (FLASH_DATA_WIDTH-1 DOWNTO 0);
12          oFL_DATA     : OUT STD_LOGIC_VECTOR (FLASH_DATA_WIDTH-1 DOWNTO 0);
13          oFL_ADDR     : OUT STD_LOGIC_VECTOR (FLASH_ADDR_WIDTH-1 DOWNTO 0);
14          iCLK,iRST    : IN STD_LOGIC;
15          iR_W         : IN STD_LOGIC;
16          oFL_CMD      : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
17          ERASE        : IN STD_LOGIC;
18          iSelect      : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
19          iFL_Ready    : IN STD_LOGIC;
20          oFL_Start    : OUT STD_LOGIC;
21          iC_ADDR      : IN STD_LOGIC_VECTOR (FLASH_ADDR_WIDTH-1 DOWNTO 0));
22  END FLASH;
23
24 ARCHITECTURE FLASH1 OF FLASH IS
25
26  SIGNAL mFL_ST           : INTEGER RANGE 0 TO 5;
27
28 BEGIN
29  oFL_ADDR    <=  iC_ADDR;
30  --------- CICLO DE LEITURA, ESCRITA E APAGAMENTO ----------
31  CICLOS : BLOCK
32  BEGIN
33      abc : PROCESS (iCLK, iRST)
107 END BLOCK CICLOS;
108 ------------------------------------------------------------
109
110 END FLASH1;
```

Figura 4.11: *Código VHDL do bloco "FLASH". (O Autor)*

4.2.2 – Conversor D/A.

Esta etapa foi criada apenas para fins de verificação do que foi gravado nas memórias, visto que o conversor D/A não faz parte, necessariamente, de um sistema de reconhecimento automático de voz. O dispositivo CODEC de áudio presente na placa faz ambas as funções de conversor D/A e A/D, basta para isso controlarmos o fluxo de dados e os sinais de controle do conversor. Assim o bloco intitulado "AUDIO_DAC", apenas gera os sinais de controle e usa os dados armazenados em uma das memórias, através dos blocos de controle de memórias já discutidos, e transmite serialmente para o CODEC de áudio, o qual irá transformar a informação digital em níveis de tensão e aplicá-las no autofalante. Assim podemos verificar se a voz do locutor foi devidamente guardada na memória. O bloco "AUDIO_DAC" utilizado é um dos blocos prontos do kit da ALTERA, escrito em Verilog, Figura 4.12. Como esse bloco se

baseia em um sinal de relógio com frequência diferente das frequências dos cristais presentes na placa, houve então a necessidade de se usar um bloco PLL, o qual também foi adquirido do próprio kit.

Figura 4.12: *Diagrama de blocos do "AUDIO_DAC" com PLL. (O Autor)*

4.2.3 – CODEC de áudio de 24 bits

Este é o dispositivo responsável por fazer o interfaceamento entre o microfone e a FPGA, e entre a FPGA e o autofalante, ou seja, exerce a função tanto de conversor A/D como de conversor D/A. O CODEC presente no kit de desenvolvimento é específico para aplicações com áudio. Este CODEC pode ser programado, onde temos opções de escolher a frequência de amostragem, a resolução da quantização em até 24 bits, entre muitas outras características que estão inteiramente detalhadas no manual do componente. As opções escolhidas foram:

- 48000Hz de frequência de amostragem, esta frequência foi escolhida pois amostra o sinal com alta fidelidade, a qual é igual à usada em áudio de filmes em DVD;
- 16 bits para a quantização dos níveis de tensão, escolheu-se 16 bits pois as memórias que o kit possui são de 16 bits ou 8 bits;
- utilização da MIC IN e da LINE OUT.

Para que o CODEC realize uma conversão A/D ou D/A utiliza-se as linhas de controle, nas quais os sinais são gerados pelos blocos desenvolvidos na FPGA, já discutidos anteriormente. Mas antes disso precisamos realizar a programação do CODEC, a qual é feita através do protocolo de comunicação de circuitos em placas, o I^2C. O kit vem também com um bloco pronto que realiza a comunicação, portanto basta encontrarmos no código deste bloco os registradores do CODEC que se deseja alterar a programação. No manual do CODEC está toda a lista dos registradores bem como o que cada bit do registrador quer dizer, assim a Figura 4.13 mostra os bits do registrador que necessitava ser alterado.

0000100 Analogue Audio Path Control	0	MICBOOST	0	Microphone Input Level Boost 1 = Enable Boost 0 = Disable Boost
	1	MUTEMIC	1	Mic Input Mute to ADC 1 = Enable Mute 0 = Disable Mute
	2	INSEL	0	Microphone/Line Input Select to ADC 1 = Microphone Input Select to ADC 0 = Line Input Select to ADC

Figura 4.13: *Os três bits LSB que precisavam ser alterados do registrador AAPC. (Dataseet WM8731)*

Na linha 131 do código, em Verilog, do bloco do protocolo I^2C, encontra-se o registrador que necessita ser modificado. Como por *default* os quatro bits menos significativos deste registrador são 1010b = Ah, assim o A/D está configurado para usar o sinal do LINE IN e não do MIC IN, como necessitamos da entrada MIC IN, então modificamos esses quatro bits para 1101b = Dh. Assim temos o valor do registrador como mostra a Figura 4.14, os demais registradores não precisaram ser modificados, pois já estavam com a configuração que desejávamos.

```
122   always
123 begin
124       case(LUT_INDEX)
125       //  Audio Config Data
126       Dummy_DATA  :   LUT_DATA    <=  16'h0000;
127       SET_LIN_L   :   LUT_DATA    <=  16'h001A;
128       SET_LIN_R   :   LUT_DATA    <=  16'h021A;
129       SET_HEAD_L  :   LUT_DATA    <=  16'h047B;
130       SET_HEAD_R  :   LUT_DATA    <=  16'h067B;
131       A_PATH_CTRL :   LUT_DATA    <=  16'h08FD;
132       D_PATH_CTRL :   LUT_DATA    <=  16'h0A06;
133       POWER_ON    :   LUT_DATA    <=  16'h0C00;
134       SET_FORMAT  :   LUT_DATA    <=  16'h0E01;
135       SAMPLE_CTRL :   LUT_DATA    <=  16'h1002;
136       SET_ACTIVE  :   LUT_DATA    <=  16'h1201;
137       default     :   LUT_DATA    <=  16'h0000;
138       endcase
139   end
```

Figura 4.14: *Valores dos registradores do CODEC de áudio. (O Autor)*

Para se desenvolver os sinais de controle do CODEC, inicialmente verifica-se o tipo de ligação entre a FPGA e o CODEC, se esta é *slave* ou *máster*, a Figura 4.15 mostra a ligação, *slave*, presente no kit. Isto foi verificado observando-se as ligações na placa do kit, assim chegamos a conclusão que a ligação era do tipo *slave*, ou seja a FPGA é a responsável por gerar todos os sinais de relógio que o CODEC necessita.

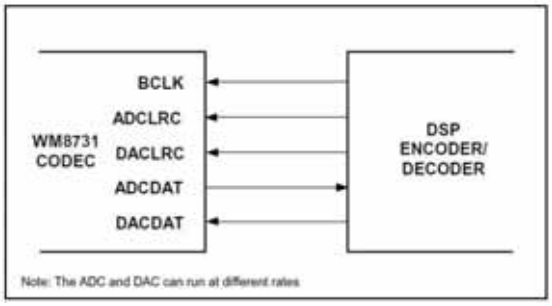


Figura 4.15: *Modo de operação slave. (Dataseet WM8731)*

A Figura 4.16, mostra o diagrama de tempo do CODEC tanto para transmissão quanto para a recepção de dados. Existem também diversos modos de se fazer a transmissão, o modo escolhido foi o *Left Justified Mode*, o mesmo utilizado pelo bloco D/A desenvolvido pela ALTERA. Este diagrama é muito importante pois os dados são enviados e recebidos serialmente, e são guardados na memória em pacotes de 16 ou 8 bits, dependendo da memória, se for SDRAM ou SRAM usa-se 16 bits, se for FLASH usa-se 8 bits.

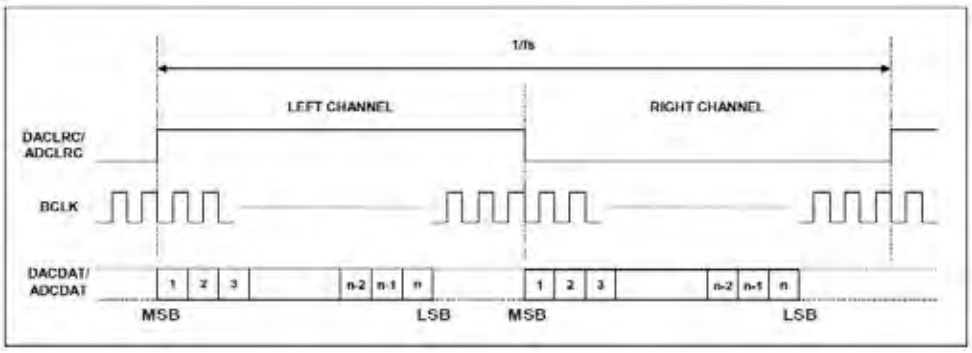


Figura 4.16: *Diagrama de tempo do Left Justified Mode. (Dataseet WM8731)*

4.2.4 – Conversor A/D

Essa é a primeira etapa a ser realizada em todo o processo do sistema RAV é nesta etapa que a voz do locutor é transformada através do microfone e do CODEC de áudio em um sinal digitalizado. Da mesma maneira que ocorre no conversor D/A, para que este processo ocorra, somente é necessário o controle do fluxo de dados e os sinais de controle do CODEC. O bloco "AUDIO_ADC", o qual pode ser visto na Figura 4.17, que gera os sinais de controle do CODEC e gerencia o fluxo de dados do CODEC para os blocos de controle da memória, foi inteiramente escrito em VHDL. A Figura 4.18 mostra um trecho do código em VHDL do bloco "AUDIO_ADC".

Apesar de ter-se escolhido a frequência de amostragem de 48kHz, esta frequência é demasiada para se realizar o processamento de um sinal de voz, consumindo muita memória e levando à um maior tempo de processamento. Assim diversos estudos empíricos chegaram a conclusão de que frequências de amostragem entre 8kHz e 12kHz são suficientes para se ter um resultado satisfatório do sistema RAV. Portanto, decidiu-se realizar uma etapa de *down-sampling* de fator 6, ou seja, reduzir a frequência de amostragem de 48kHz para 8kHz. Esta etapa foi descrita diretamente no código do bloco "AUDIO_ADC".

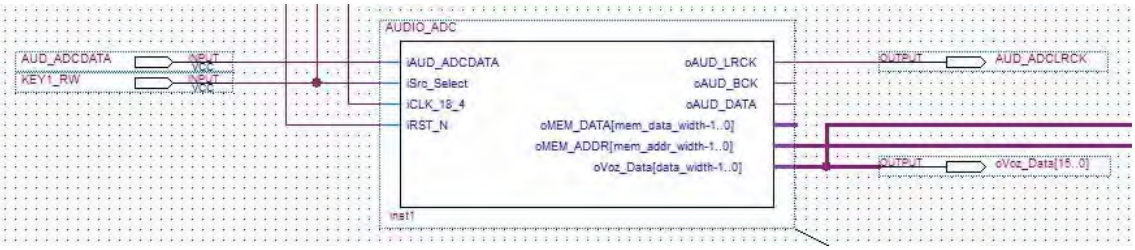


Figura 4.17: *Diagrama de blocos do "AUDIO_ADC". (O Autor)*

```
16              MEM_DATA_WIDTH    : INTEGER := 8);
17  PORT (  iAUD_ADCDATA: IN STD_LOGIC;
18          oAUD_LRCK   : OUT STD_LOGIC;
19          oAUD_BCK    : BUFFER STD_LOGIC;
20          oAUD_DATA   : OUT STD_LOGIC;
21          oMEM_DATA   : OUT STD_LOGIC_VECTOR (MEM_DATA_WIDTH-1 DOWNTO 0);
22          oMEM_ADDR   : BUFFER STD_LOGIC_VECTOR (MEM_ADDR_WIDTH-1 DOWNTO 0);
23          iSrc_Select : IN STD_LOGIC;
24          iCLK_18_4   : IN STD_LOGIC;
25          iRST_N      : IN STD_LOGIC;
26          oVoz_Data   : OUT STD_LOGIC_VECTOR (DATA_WIDTH-1 DOWNTO 0));
27  END AUDIO_ADC;
28
29  ARCHITECTURE AUDIO_ADC1 OF AUDIO_ADC IS
30
31  SIGNAL BCK_DIV          : INTEGER;
32  SIGNAL LRCK_1X_DIV      : INTEGER;
33  SIGNAL LRCK_2X_DIV      : INTEGER;
34  SIGNAL LRCK_4X_DIV      : INTEGER;
35  SIGNAL LRCK_1X          : STD_LOGIC;
36  SIGNAL LRCK_2X          : STD_LOGIC;
37  SIGNAL LRCK_4X          : STD_LOGIC;
38  SIGNAL MEM_Cont         : INTEGER RANGE 0 TO MEM_DATA_NUM;
39  SIGNAL MEM_Cont_Vector  : STD_LOGIC_VECTOR (MEM_ADDR_WIDTH-1 DOWNTO 0);
40  SIGNAL SEL_Cont         : INTEGER RANGE 0 TO 15;
41  SIGNAL R_W              : STD_LOGIC;
42  SIGNAL MEM_In_Tmp       : STD_LOGIC_VECTOR (DATA_WIDTH-1 DOWNTO 0);
43  SIGNAL MEM_In           : STD_LOGIC_VECTOR (7 DOWNTO 0);
44
45  BEGIN
46  ----------  AUD_LRCK Generator  --------------
47  AUD_LRCK : BLOCK
48  BEGIN
49      abc : PROCESS (iCLK_18_4, iRST_N)
82  END BLOCK AUD_LRCK;
83  oAUD_LRCK <= LRCK_1X;
84  ------------------------------------------
85
86  ----------  AUD_BCK Generator  -------------
87  AUD_BCK : BLOCK
88  BEGIN
89      abc: PROCESS (iCLK_18_4, iRST_N)
103 END BLOCK AUD_BCK;
104 ------------------------------------------
105
106 ---------   MEM_ADDR Generator  -----------
107 MEM_ADDR : BLOCK
108 BEGIN
109     abc : PROCESS (LRCK_4X, iRST_N)
121 END BLOCK MEM_ADDR;
```

Figura 4.18: *Parte do código VHDL do bloco "AUDIO_ADC". (O Autor)*

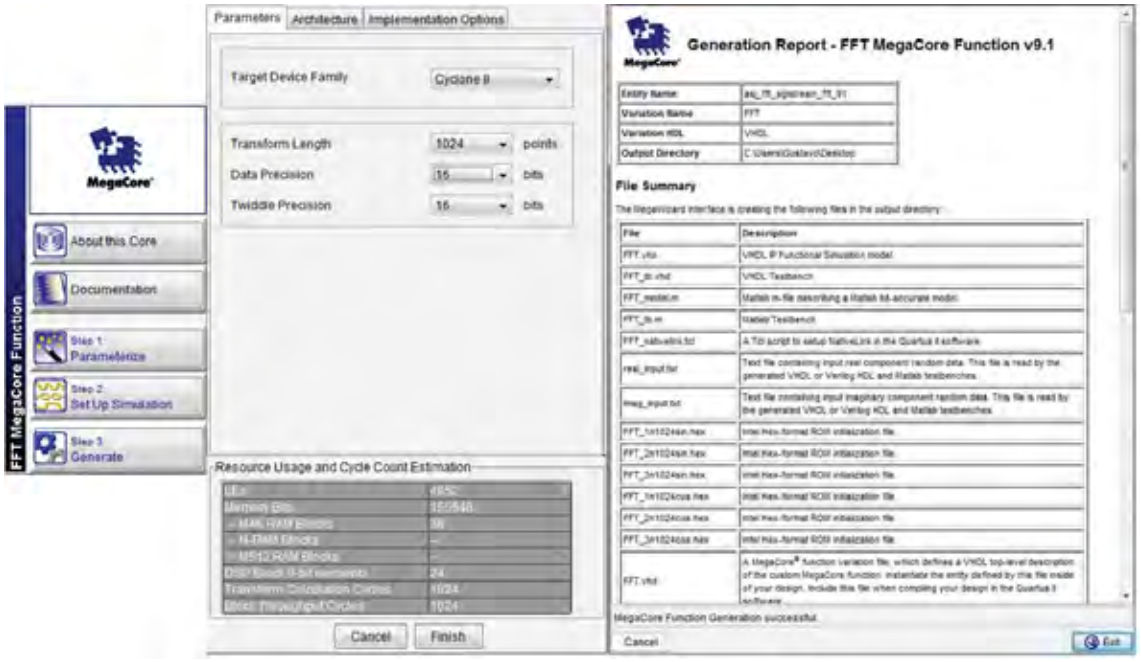
4.3 – Bloco de FFT

De acordo com o que foi visto no item 1.5, a FFT realiza a transformação do sinal do domínio do tempo para o domínio da frequência, pois a técnica de reconhecimento que iremos utilizar trabalha neste domínio. Criar um processador de FFT não é nada trivial, é um trabalho bem difícil e complexo. Como a rede de reconhecimento em si já é bem complexa de se desenvolver, então optamos por procurar uma solução que já existisse para a FFT, para que pudéssemos nos concentrar no desenvolvimento do restante do sistema. Vimos que a fabricante do próprio kit de desenvolvimento e da ferramenta de desenvolvimento, já tinha uma solução bem eficaz para um processador de FFT. Tal solução tem o nome de *IP – Intelectual Property*, presentes na *MegaCore Function Open Plus*, a ser detalhada a seguir.

4.3.1 – MegaCore Function Open Plus – FFT – Altera

A *MegaCore Function Open Plus* da Altera, está presente nas versões mais atuais da ferramenta de desenvolvimento Quartus II. A partir da versão 9.1, todas as IP's presente nesta ferramenta são gratuitas, para se realizar a especificação, simulação, síntese e verificação em placa da IP. Para estas propriedades intelectuais, deve-se adquirir uma licença, caso o fim da aplicação em que se esta utilizando-a seja comercial, ou seja, quando se esta desenvolvendo um produto a ser lançado no mercado. Como o fim deste trabalho é apenas para estudo de caso do sistema e fins de pesquisa, não nos caracterizamos na área comercial, portanto não temos problemas em ter de adquirir a licença da IP. Todas essas informações podem ser retiradas e verificadas na própria página da web da fabricante e nos manuais referente a cada IP. A IP da FFT tem uma única limitação quando se deseja verificar em placa seu desempenho. Como estamos usando-a no modo *"free"*, sem licença, então a placa precisa estar conectada ao software Quartus II para funcionar por período indeterminado, caso o cabo USB da placa seja retirado, a FFT irá operar por durante uma hora e depois irá travar. Isso é feito justamente para impedir a sua comercialização sem a licença. As Figuras 4.19 e 4.20 mostram as telas da ferramenta *MegaCore Function*, onde realizamos a escolha da IP e a parametrização da mesma junto com os arquivos gerados pela ferramenta.

Figura 4.19: *Tela da MegaFunction onde se escolhe a IP a ser usada, no caso, a FFT. (O Autor)*



(a) *(b)* *(c)*

Figura 4.20: *(a)Ferramenta FFT MegaCore; (b) Tela de parametrização; (c) Arquivos gerados. (O Autor)*

Um ponto bem interessante desta ferramenta é que a cada modificação que fazemos durante a parametrização, a mesma nos mostra o consumo de LE's e de Memory Bits que a FFT irá requerer, assim podemos saber se o componente irá suportar a IP ou não. Em nosso sistema parametrizamos a FFT da seguinte forma: comprimento de 1024 pontos e 16 bits de dados.

4.3.2 – Bloco de controle da FFT

O bloco de FFT criado pela *MegaCore*, realiza a transformada de acordo com as entradas fornecidas e os respectivos sinais de controle tanto de entrada quanto de saída. Para isso foi criado, em VHDL, um bloco de controle responsável por gerar os sinais de controle de entrada e monitorar os sinais de controle de saída, para que o fluxo de dados possa ser respeitado, ou seja, o mesmo envia o sinal no domínio do tempo para a FFT, e recebe da mesma o sinal no domínio da frequência e o envia para o bloco de controle da SRAM, onde a FFT do sinal de voz será armazenada. As Figuras 4.21 e 4.22 mostram respectivamente, a interligação dos blocos de controle e IP, e um trecho do código VHDL do bloco de controle desenvolvido.

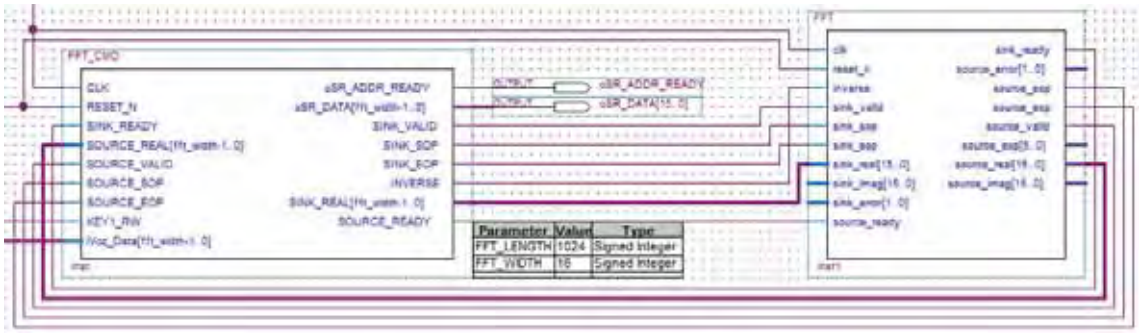


Figura 4.21: *Interligação entre os blocos de controle e da FFT. (O Autor)*

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3   USE ieee.std_logic_arith.all;
4   USE ieee.std_logic_unsigned.all;
5
6   LIBRARY work;
7
8   ENTITY FFT_CMD IS
9   GENERIC (FFT_LENGTH : INTEGER := 1024;      -- Tamanho da FFT
10            FFT_WIDTH  : INTEGER := 16);      -- Numero de bits da FFT
11  PORT     (CLK            : IN STD_LOGIC;
12            RESET_N         : IN STD_LOGIC;
13            oSR_ADDR_READY  : OUT STD_LOGIC;
14            oSR_DATA        : OUT STD_LOGIC_VECTOR (FFT_WIDTH-1 DOWNTO 0);
15            SINK_VALID      : OUT STD_LOGIC;
16            SINK_READY      : IN STD_LOGIC;
17            SINK_SOP        : OUT STD_LOGIC;
18            SINK_EOP        : OUT STD_LOGIC;
19            INVERSE         : OUT STD_LOGIC;
20            SINK_REAL       : OUT STD_LOGIC_VECTOR (FFT_WIDTH-1 DOWNTO 0);  --Entrada de dado Real da FFT
21            SOURCE_REAL     : IN STD_LOGIC_VECTOR (FFT_WIDTH-1 DOWNTO 0);   --Saida de dado Real da FFT
22            SOURCE_VALID    : IN STD_LOGIC;
23            SOURCE_READY    : OUT STD_LOGIC;
24            SOURCE_SOP      : IN STD_LOGIC;
25            SOURCE_EOP      : IN STD_LOGIC;
26            KEY1_RW         : IN STD_LOGIC;
27            iVoz_Data       : IN STD_LOGIC_VECTOR (FFT_WIDTH-1 DOWNTO 0));
28  END FFT_CMD;
29
30  ARCHITECTURE FFT_CMD1 OF FFT_CMD IS
31
32  SIGNAL CONT          : INTEGER RANGE 0 TO FFT_LENGTH;
33  SIGNAL CONT2         : INTEGER RANGE 0 TO 1;
34
35  ---------- Bloco de entrada de dados na FFT -----
36  BEGIN
77  ------------------------------------------------
78
79  ---------- Bloco de saida de dados da FFT -------
80  OUTPUT_DATA_FFT : BLOCK
81  BEGIN
82      abc: PROCESS (CLK,RESET_N)
106 END BLOCK OUTPUT_DATA_FFT;
107 ------------------------------------------------
108
109
110 END FFT_CMD1;
111
```

Figura 4.22: *Código VHDL do bloco de controle da FFT. (O Autor)*

4.3.3 – Bloco de Memória SRAM

Este bloco é o mesmo já discutido no item 4.2.1.2, com a adição de um divisor de frequência para ajustar a frequência do cristal usado à frequência do sinal amostrado. A função deste bloco é pegar os dados gerados pela FFT, repassados pelo respectivo bloco de controle, e guardá-los na memória SRAM. A Figura 4.23 mostra o bloco com divisor de frequência e a Figura 4.24 o código do divisor de frequência em VHDL.

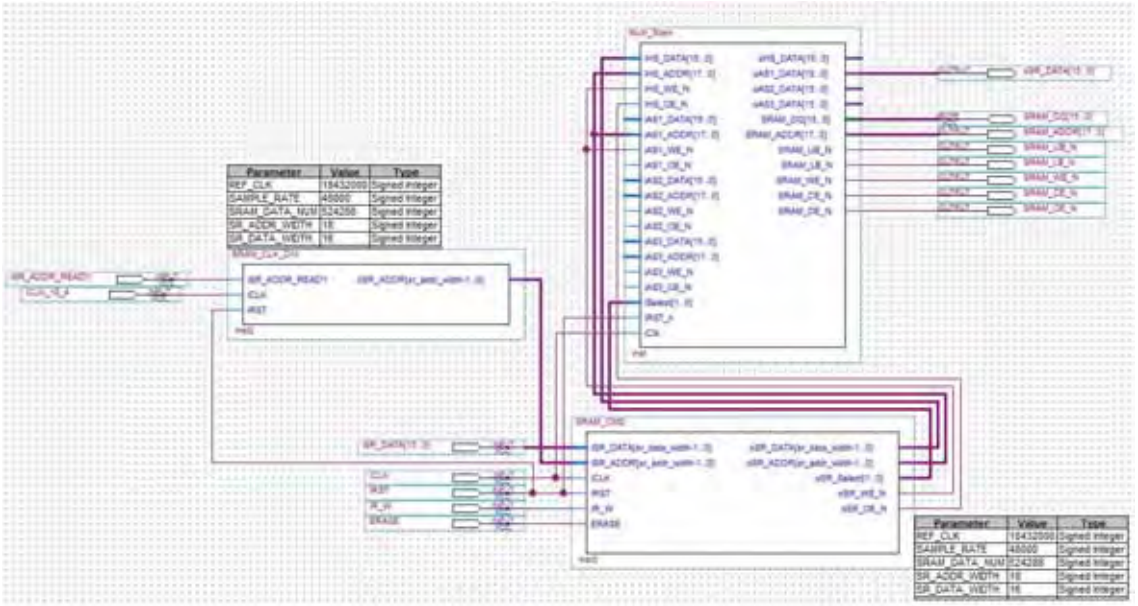


Figura 4.23: *Bloco de SRAM completo. (O Autor)*

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3   USE ieee.std_logic_unsigned.ALL;
4   USE ieee.std_logic_arith.ALL;
5
6   LIBRARY work;
7
8   ENTITY SRAM_CLK_DIV IS
9   GENERIC (  REF_CLK          : INTEGER := 18432000;  --18.34MHz
10             SAMPLE_RATE      : INTEGER := 48000;     -- 48kHZ
11             SRAM_DATA_NUM    : INTEGER := 524288;    -- 512 Kbits
12             SR_ADDR_WIDTH    : INTEGER := 18;
13             SR_DATA_WIDTH    : INTEGER := 16);
14  PORT (  oSR_ADDR        : OUT STD_LOGIC_VECTOR (SR_ADDR_WIDTH-1 DOWNTO 0);
15          iSR_ADDR_READY  : IN STD_LOGIC;
16          iCLK            : IN STD_LOGIC;
17          iRST            : IN STD_LOGIC);
18  END SRAM_CLK_DIV;
19
20  ARCHITECTURE SRAM_CLK OF SRAM_CLK_DIV IS
21
22  SIGNAL LRCK_1X_DIV      : INTEGER;
23  SIGNAL LRCK_2X_DIV      : INTEGER;
24  SIGNAL LRCK_4X_DIV      : INTEGER;
25  SIGNAL LRCK_1X          : STD_LOGIC;
26  SIGNAL LRCK_2X          : STD_LOGIC;
27  SIGNAL LRCK_4X          : STD_LOGIC;
28  SIGNAL SRAM_Cont        : INTEGER RANGE 0 TO SRAM_DATA_NUM;
29  SIGNAL SRAM_Cont_Vector : STD_LOGIC_VECTOR (SR_ADDR_WIDTH-1 DOWNTO 0);
30
31  BEGIN
32
33  ----------  AUD_LRCK Generator  --------------
34  AUD_LRCK : BLOCK
35  BEGIN
36      abc : PROCESS (iCLK, iRST)
69  END BLOCK AUD_LRCK;
70  ------------------------------------------------
71
72  ---------   SRAM ADDR Generator -----------
73  SRAM_ADDR : BLOCK
74  BEGIN
75      abc : PROCESS (LRCK_4X, iRST)
93  END BLOCK SRAM_ADDR;
94  ------------------------------------------------
95
96  END SRAM_CLK;
```

Figura 4.24: *Código VHDL do divisor de freqüência da SRAM. (O Autor)*

4.4 – Bloco de Pré-Processamento

O bloco de pré-processamento, mais conhecido como pré-ênfase, é constituído por três etapas: Filtro de Pré-Ênfase, Divisão em Quadros e Janelamento. A Figura 4.25 mostra o diagrama de blocos realizado da etapa de pré-ênfase. No sub-bloco ”PRE_FILTER”, realiza-se a implementação do filtro descrito na equação 3.30 para *a=15/16*. Este sub-bloco faz a leitura do sinal de voz armazenado na memória SDRAM, faz o cálculo do filtro e o resultado é passado para o próximo bloco.

O sub-bloco “QUAD_JAN” realiza as outras duas etapas, a divisão em quadros e o janelamento, em conjunto. Neste sub-bloco realiza-se uma contagem até 240, pois este é o tamanho de cada quadro, quando esta contagem é atingida um sinal é enviado para o sub-bloco “PRE_FILTER” para que este indique onde se inicia o próximo quadro. O resultado do

filtro é então multiplicado pela função-janela de Hamming, a qual foi amostrada para 240 amostras e armazenada na memória FLASH.

O resultado da multiplicação é então armazenado na memória SRAM de maneira sequencial, ou seja, de 0 a 240 está o primeiro quadro, de 241 a 482 está o segundo quadro e assim por diante. Para todos os cálculos realizados foram utilizados números e lógica de ponto fixo, convertidos para inteiros e, depois dos cálculos, convertidos para binário no formato sinal magnitude em complemento de 2 novamente. A Figura 4.26 mostra um trecho do código em VHDL de ambos os blocos desenvolvidos.

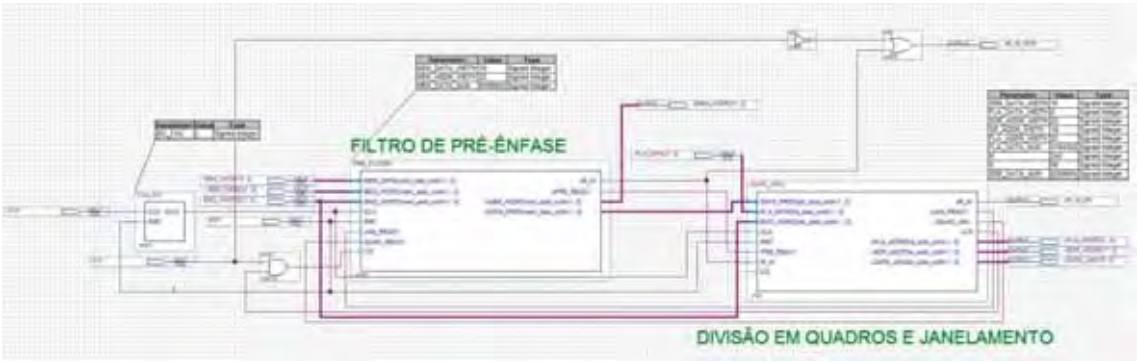


Figura 4.25: *Bloco de pré-ênfase. (O Autor)*

```
          iRST            : IN STD_LOGIC;
          iJAN_READY      : IN STD_LOGIC;
          iQUAD_READY     : IN STD_LOGIC;
          oR_W            : OUT STD_LOGIC;
          iCS             : IN STD_LOGIC;
          oPRE_READY      : OUT STD_LOGIC;
          oMEM_ADDR       : OUT STD_LOGIC_VECTOR (MEM_ADDR_WIDTH-1 DOWNTO 0);
          oDATA_PREF      : OUT STD_LOGIC_VECTOR (MEM_DATA_WIDTH-1 DOWNTO 0));
END PRE_FILTER4;

ARCHITECTURE FILTER OF PRE_FILTER4 IS

SIGNAL SN_1          : STD_LOGIC_VECTOR (MEM_DATA_WIDTH-1 DOWNTO 0);
SIGNAL SN            : STD_LOGIC_VECTOR (MEM_DATA_WIDTH-1 DOWNTO 0);
SIGNAL SN_INT        : INTEGER RANGE -32768 TO 32768;
SIGNAL SN_1_INT      : INTEGER RANGE -32768 TO 32768;
SIGNAL S_N_INT       : INTEGER RANGE -32768 TO 32768;
SIGNAL R_W           : STD_LOGIC;
SIGNAL MEM_ADDR_BEG  : STD_LOGIC_VECTOR (MEM_ADDR_WIDTH-1 DOWNTO 0);
SIGNAL MEM_ADDR      : STD_LOGIC_VECTOR (MEM_ADDR_WIDTH-1 DOWNTO 0);
SIGNAL MEM_ADDR_END  : STD_LOGIC_VECTOR (MEM_ADDR_WIDTH-1 DOWNTO 0);
SIGNAL MEM_ADDR_DIF  : STD_LOGIC_VECTOR (MEM_ADDR_WIDTH-1 DOWNTO 0);
SIGNAL CONT          : STD_LOGIC_VECTOR (MEM_ADDR_WIDTH-1 DOWNTO 0);
SIGNAL CONT2         : STD_LOGIC_VECTOR (MEM_ADDR_WIDTH-1 DOWNTO 0);
SIGNAL CONT3         : STD_LOGIC_VECTOR (MEM_ADDR_WIDTH-1 DOWNTO 0);
SIGNAL STATE         : INTEGER RANGE 0 TO 3;

BEGIN

MEM_ADDR_BEG <= iBEG_WORD;
MEM_ADDR <= MEM_ADDR_BEG + CONT2;
MEM_ADDR_END <= iEND_WORD;
SN_1_INT <= CONV_INTEGER(SN_1);
SN_INT <= CONV_INTEGER(SN);

---------- Aquisição das amostras na memória--------
VOZ : BLOCK
BEGIN
    abc : PROCESS (iRST, iCLK,iJAN_READY, STATE)

    cba : PROCESS (iRST,iQUAD_READY)
END BLOCK VOZ;
-------------------------------------------------

--oR_W <= S_W;
oDATA_PREF <= CONV_STD_LOGIC_VECTOR(S_N_INT, MEM_DATA_WIDTH) WHEN iCS = '1' ELSE "ZZZZZZZZZZZZZZZZ";
oMEM_ADDR <= (MEM_ADDR + CONT) WHEN iCS = '1' ELSE "ZZZZZZZZZZZZZZZZZZZZZZZZ";

END FILTER;
```

(a)

```
20          iEND_WORD      : IN STD_LOGIC_VECTOR (SDR_ADDR_WIDTH-1 DOWNTO 0);
21          iCLK           : IN STD_LOGIC;
22          iRST           : IN STD_LOGIC;
23          iPRE_READY     : IN STD_LOGIC;
24          iR_W           : IN STD_LOGIC;
25          iCS            : IN STD_LOGIC;
26          oR_W           : OUT STD_LOGIC;
27          oJAN_READY     : OUT STD_LOGIC;
28          oQUAD_JAN      : OUT STD_LOGIC;
29          oCS            : OUT STD_LOGIC;
30          oFLA_ADDR      : OUT STD_LOGIC_VECTOR (FLA_ADDR_WIDTH-1 DOWNTO 0);
31          oSDR_ADDR      : OUT STD_LOGIC_VECTOR (SR_ADDR_WIDTH-1 DOWNTO 0);
32          oDATA_JAN      : OUT STD_LOGIC_VECTOR (SDR_DATA_WIDTH-1 DOWNTO 0));
33  END QUAD_JAN3;
34
35  ARCHITECTURE JANELA OF QUAD_JAN3 IS
36
37  SIGNAL HN           : STD_LOGIC_VECTOR (FLA_DATA_WIDTH-1 DOWNTO 0);
38  SIGNAL SN           : STD_LOGIC_VECTOR (SDR_DATA_WIDTH-1 DOWNTO 0);
39  SIGNAL SN_INT       : INTEGER RANGE -32768 TO 32768;
40  SIGNAL HN_INT       : INTEGER RANGE -32768 TO 32768;
41  SIGNAL S_N_INT      : INTEGER RANGE -32768 TO 32768;
42  SIGNAL R_W          : STD_LOGIC;
43  SIGNAL CS           : STD_LOGIC;
44  SIGNAL SDR_ADDR_END : STD_LOGIC_VECTOR (SDR_ADDR_WIDTH-1 DOWNTO 0);
45  SIGNAL FLA_ADDR_BEG : STD_LOGIC_VECTOR (FLA_ADDR_WIDTH-1 DOWNTO 0);
46  SIGNAL MEM_ADDR_DIF : STD_LOGIC_VECTOR (SDR_ADDR_WIDTH-1 DOWNTO 0);
47  SIGNAL CONT         : STD_LOGIC_VECTOR (SDR_ADDR_WIDTH-1 DOWNTO 0);
48  SIGNAL CONT_INT     : INTEGER RANGE 0 TO 240;
49  SIGNAL CONT3        : STD_LOGIC_VECTOR (SR_ADDR_WIDTH-1 DOWNTO 0);
50  SIGNAL STATE        : INTEGER RANGE 0 TO 4;
51
52  BEGIN
53
54  SN_INT <= CONV_INTEGER(SN);
55  HN_INT <= CONV_INTEGER(HN);
56  CONT_INT <= CONV_INTEGER(CONT);
57
58  QUADRO : BLOCK
59  BEGIN
60      ABC : PROCESS (iRST,iCLK,iPRE_READY, STATE, iR_W)
118 END BLOCK QUADRO;
119
120 oFLA_ADDR <= (FLA_ADDR_BEG + CONT) WHEN (CS = '1') ELSE "ZZZZZZZZZZZZZZZZZZZZZZ";
121 oDATA_JAN <= CONV_STD_LOGIC_VECTOR(S_N_INT,SDR_DATA_WIDTH) WHEN (CS = '1') ELSE "ZZZZZZZZZZZZZZZZ";
122 --oDATA_JAN <= SN WHEN (CS = '1') ELSE "ZZZZZZZZZZZZZZZZ";
123 oSDR_ADDR <= (CONT3) WHEN (CS = '1') ELSE "ZZZZZZZZZZZZZZZZZZZZ";
124
125 END JANELA;
```

(b)

Figura 4.26: *Código VHDL.(a) bloco "PRE_FILTER";(b) bloco "QUAD_JAN".*

4.5 – Testes e Simulações

De modo a verificar cada um dos módulos desenvolvidos, utilizamos a ferramenta *SignalTap II Logic Analyzer*, disponível no Quartus II. Esta ferramenta é muito utilizada na fase de *debuging* dos módulos, pois nos permite verificar o comportamento dos sinais, tanto externos quanto dos registradores internos da FPGA.

Para se acessar o *SignalTap II Logic Analyzer*, basta clicar na opção *Tools*, localizado na barra de menus, e depois em *SignalTap II Logic Analyzer*, como mostra a Figura 4.27.

Figura 4.27: *SignalTap II Logic Analyzer. (O Autor)*

Na tela principal do *SignalTap II Logic Analyzer,* Figura 4.28, deve-se inicialmente clicar em *"Scan Chain"*, para que o *software* possa reconhecer a FPGA com a qual irá se comunicar. Depois devemos determinar qual será o *clock* de referência da ferramenta, em *"Clock"("Signal Configuration").* Pode-se então escolher quais serão os sinais a serem monitorados pela ferramenta, clicando duas vezes, na área branca abaixo de "*Node*". Após realizadas essas etapas, deve-se então, salvar este arquivo e associá-lo ao projeto, e em seguida compilar o projeto. Por último deve-se realizar a programação da FPGA, para por fim clicar em *"Autorun Analysis"* e então iniciar o processo de monitoramento.

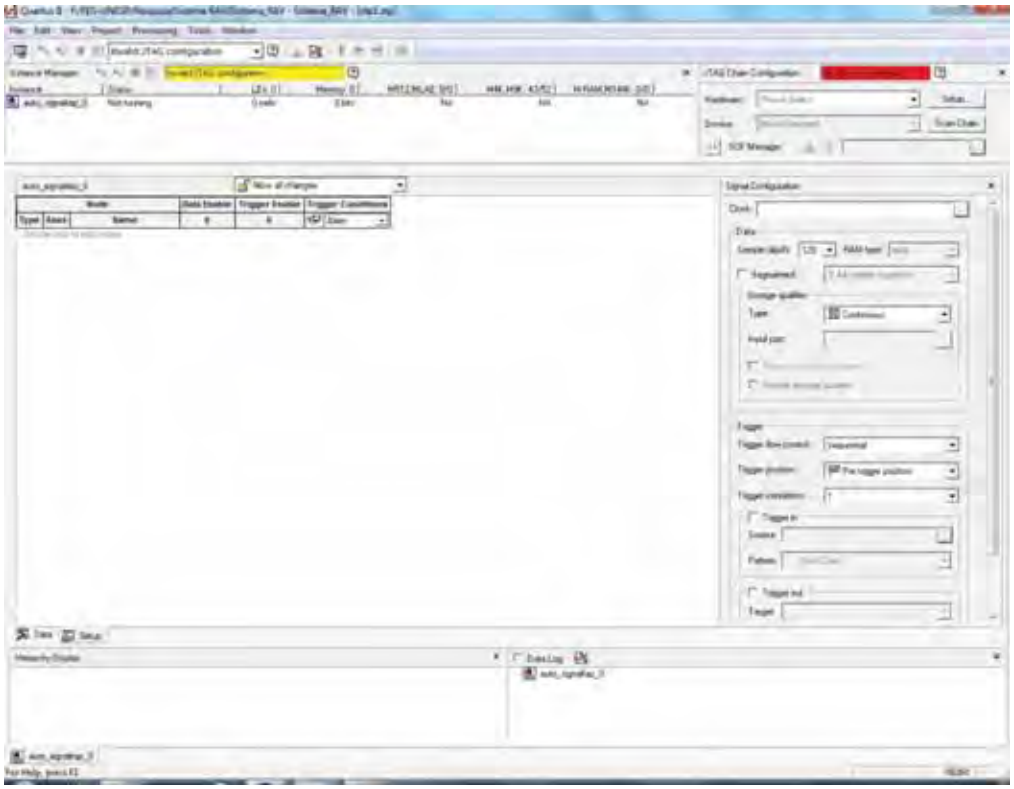


Figura 4.28: *Tela principal do SignalTap II Logic Analyzer. (O Autor)*

O sistema desenvolvido funciona da seguinte maneira: as chaves do tipo *switch* SW0 e SW1, selecionam a voz armazenada na SDRAM ou a FFT armazenada na SRAM, para serem enviadas aos autofalantes, caso SW0 for nível lógico alto e SW1 for nível lógico baixo seleciona-se a SDRAM, caso SW0 for nível lógico baixo e SW1 for nível lógico alto seleciona-se a SRAM e caso ambas forem nível lógico baixo, um sinal senoidal de 1kHz é emitido pelos autofalantes, apenas para verificar se o algoritmo está em funcionamento; as *switchs* SW7 e SW8 selecionam qual memória se deseja apagar, sendo a SDRAM se SW8 for nível lógico alto, e a SRAM se SW7 for nível lógico alto; o *push-botton* KEY0 é o *reset* do sistema; o *push-botton* KEY1 tem dupla função, caso as chaves SW7 e SW8 estiverem em nível lógico baixo, enquanto pressionado grava a voz adquirida pelo microfone na SDRAM e a FFT calculada na SRAM, caso contrário apaga a(s) memória(s) selecionada(s), se pressionado por no mínimo dois segundos.

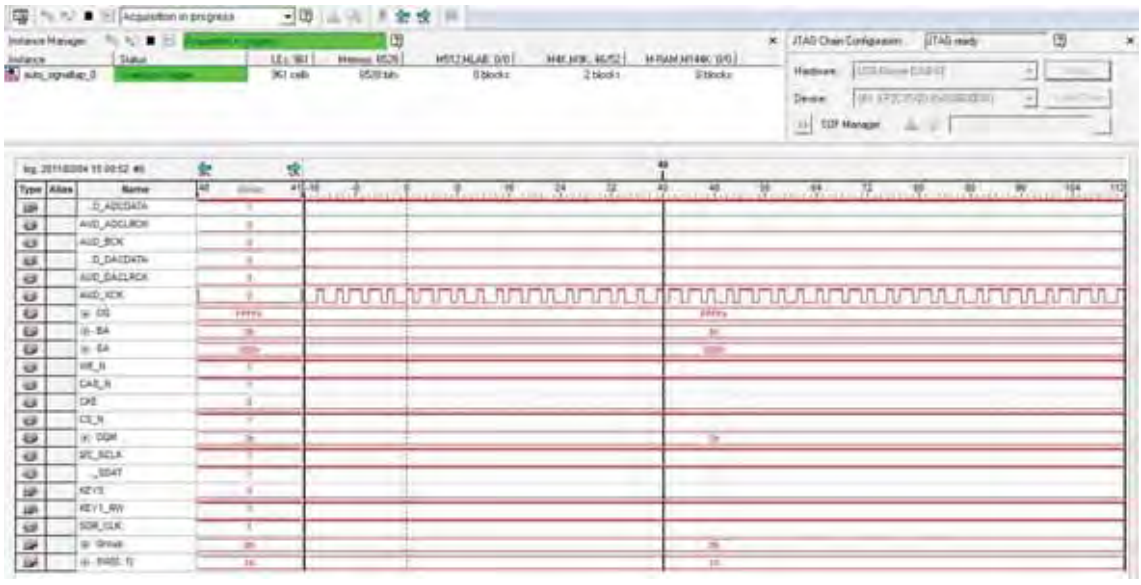


Figura 4.29: *Bloco de aquisição de dados em reset. (O Autor)*

A Figura 4.29 mostra a tela do *SignalTap II Logic Analyzer* apresentando o sistema em funcionamento. Pode-se ver que quando o sinal KEY0 está em nível lógico baixo, todas as demais saídas são forçadas para seus respectivos estados iniciais, ou seja, este sinal causa um *reset* no sistema. O sinal AUD_XCK é um sinal de relógio, portanto independente do sinal de *reset* ele oscilará constantemente.

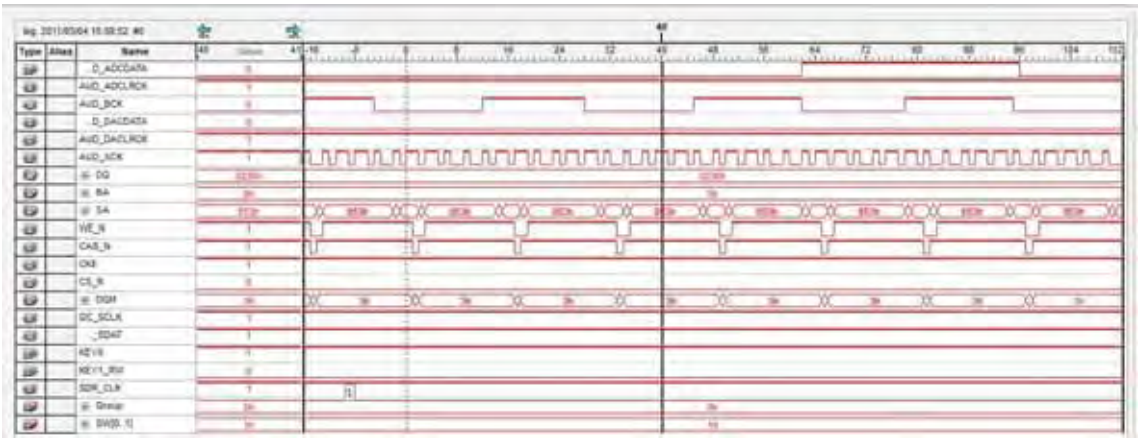


Figura 4.30: *Bloco de aquisição de dados em escrita na SDRAM. (O Autor)*

A Figura 4.30 mostra o processo de escrita da memória SDRAM. Quando o sinal KEY1_RW está em nível lógico baixo e os sinais SW7_E_SRAM e SW8_E_SDRAM (estas chaves estão representadas em conjunto em hexadecimal no sinal *Group*) também estão em nível lógico baixo, a voz adquirida pelo microfone é convertida em sinal digital e armazenada na memória SDRAM. O sinal DQ é bidirecional e neste momento o mesmo está enviando os dados para a memória SDRAM.

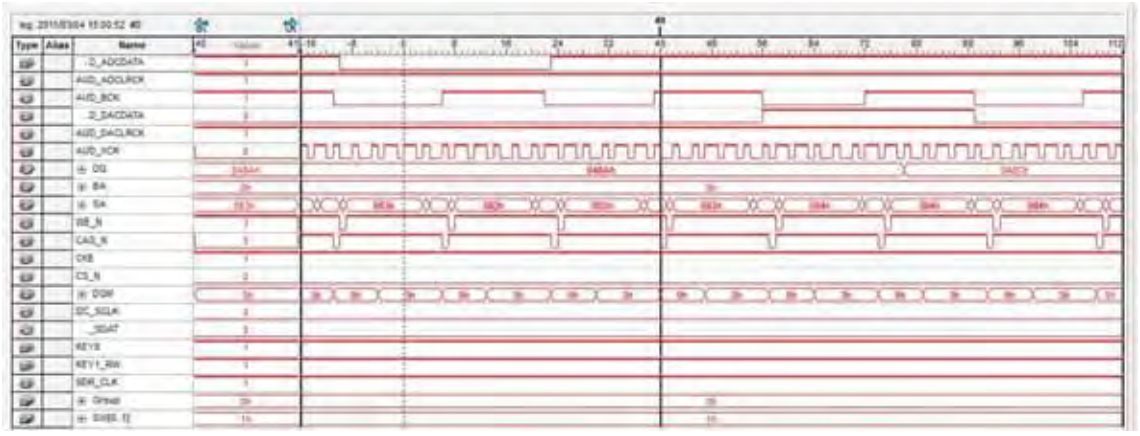


Figura 4.31: *Bloco de aquisição de dados em leitura da SDRAM. (O Autor)*

A Figura 4.31 mostra o processo de leitura da memória SDRAM, onde para que isso aconteça os sinais KEY1_RW deve estar em nível lógico alto e a chave SW0 deve estar em nível lógico alto também. Assim os dados da memória SDRAM são lidos de maneira sequencial, e enviados para o CODEC de áudio o qual converte o sinal digital para analógico, nos permitindo assim ouvir nos autofalantes o que foi armazenado na memória. Isto é feito para se verificar se o que foi armazenado na memoria é exatamente igual ao que foi dito pelo locutor no microfone.

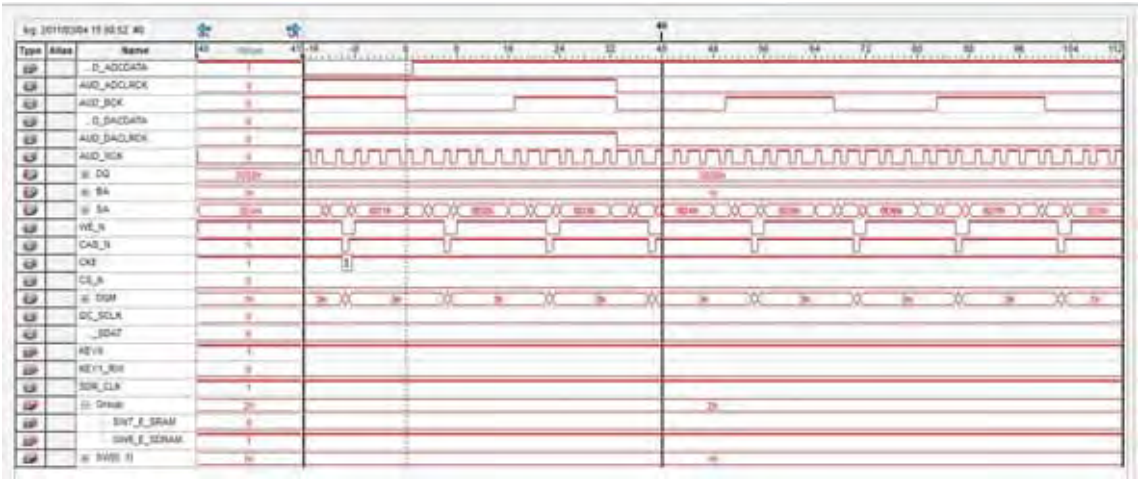


Figura 4.32: *Bloco de aquisição de dados apagando a SDRAM. (O Autor)*

A Figura 4.32 mostra que se o sinal SW8_E_SDRAM for nível lógico alto, e o sinal KEY1_RW for nível lógico baixo, o dado 0000h é escrito em todas as posições da memória SDRAM, apagando-se assim todo o conteúdo antes armazenado na memória.

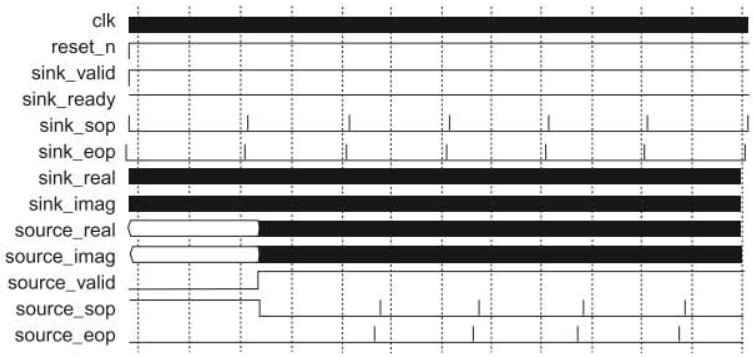


Figura 4.33: *Diagrama de tempo teórico do controle da FFT.*
(fonte: http://www.altera.com/literature/ug/ug_fft.pdf)

A Figura 4.33 mostra o diagrama de tempo de controle contido no guia do usuário da MegaFunction FFT da ALTERA, ou seja, para que a FFT possa funcionar corretamente, devemos gerar os sinais de acordo com o que esta apresentado nesta figura.

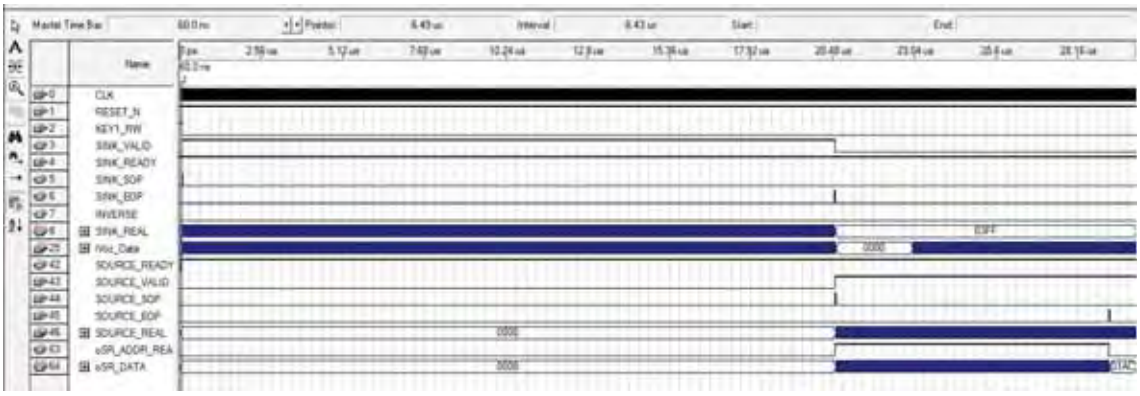


Figura 4.34: *Simulação do bloco de controle da FFT. (O Autor)*

A Figura 4.34 mostra a simulação feita do bloco de controle FFT_CMD, o qual foi desenvolvido para se gerar os sinais de controle da FFT. Pode-se ver que o resultado obtido da simulação corresponde ao mesmo comportamento esperado pela FFT mostrado na Figura 4.33, ou seja, o bloco desenvolvido gera corretamente os sinais de controle necessários, segundo esta simulação.

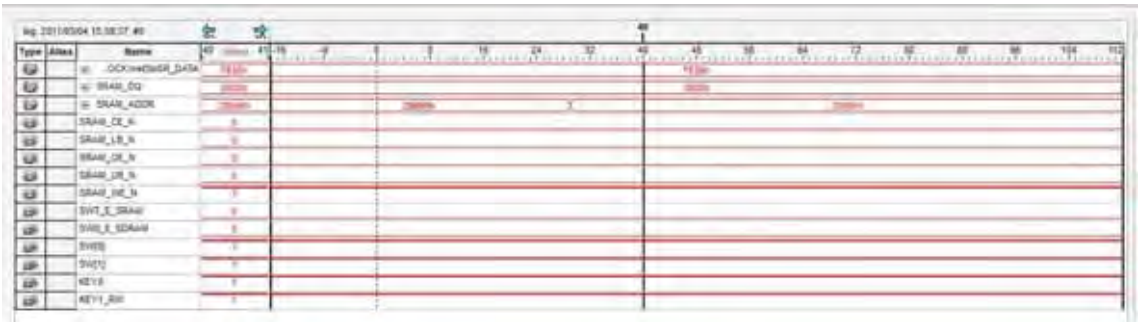


Figura 4.35: *Bloco de SRAM em leitura. (O Autor)*

Na Figura 4.35, pode-se ver a operação de leitura da SRAM, a qual está apagada, isto pode ser visto pelo sinal SRAM_DQ que está recebendo o dado 0000h, em um determinado endereço da SRAM, o qual difere do dado do barramento oSR_DATA, que está enviando os dados para este bloco de controle para serem armazenados na SRAM, assim confirma-se que a operação é de leitura e não de escrita. Veja também que para que o processo de leitura ocorra o sinal KEY1_RW deve estar em nível lógico alto.

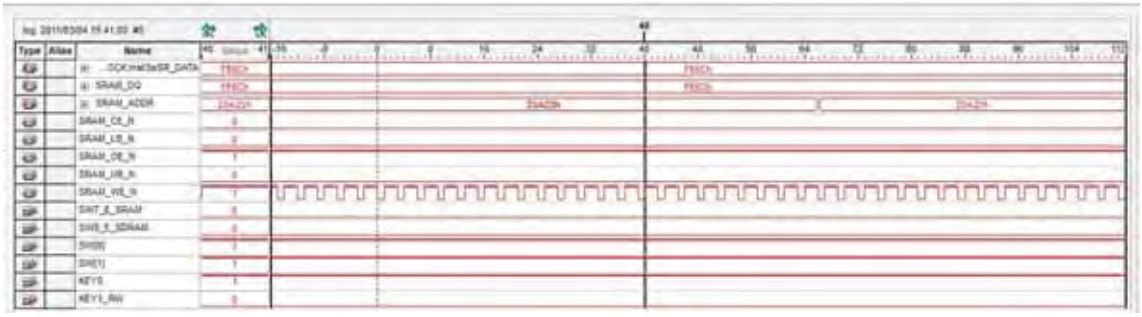


Figura 4.36: *Bloco de SRAM em escrita. (O Autor)*

Já na Figura 4.36, vemos a operação de escrita na SRAM, a qual está armazenando os dados calculados da FFT do sinal de voz. Esta operação ocorre quando o sinal KEY1_RW está em nível lógico baixo e o sinal SW7_E_SRAM está em nível lógico baixo também. Pode-se observar também que neste caso o dado do barramento oSR_DATA é exatamente igual ao dado do barramento SRAM_DQ, ou seja, o dado está sendo enviado para a SRAM, comprovando assim a operação de escrita na memória.

Figura 4.37: *Bloco de SRAM sendo apagado. (O Autor)*

Por fim na Figura 4.37 vemos a SRAM sendo apagada, ou seja, esta é uma operação de escrita, na qual se armazena o dado 0000h em todos os endereços da memória. Para que esta operação ocorra o sinal SW7_E_SRAM deve estar em nível lógico alto e o sinal KEY1_RW em nível lógico baixo, durante no mínimo dois segundos.

Figura 4.38: *Bloco de FFT em funcionamento. (O Autor)*

A Figura 4.38, mostra o funcionamento do bloco da FFT, o qual entra em operação assim que o sinal KEY1_RW for para nível lógico baixo. Neste momento se os sinais SW7_E_SRAM e SW8_E_SDRAM também forem nível lógico baixo, os dados de voz são enviados para o bloco de FFT, o qual após um tempo depois de receber o primeiro pacote de 1024 dados (este valor é o comprimento escolhido da FFT), começa a enviar os dados da FFT calculada para o bloco de controle da SRAM, o qual enviará para a memória SRAM, onde os dados serão armazenados.

Para se verificar o funcionamento do bloco de pré-ênfase, foi desenvolvido o sistema mostrado na Figura 4.39, onde a voz adquirida pelo bloco “aqs_voz” é armazenada na memória SDRAM. Após o armazenamento da voz, apenas para testes, a etapa de pré-ênfase é ativada colocando-se a chave SW6 em nível lógico alto. Nesta etapa o bloco de pré-ênfase faz a leitura da voz armazenada na memória SDRAM, realiza os cálculos pertinentes à esta etapa, fazendo a leitura da função janela de Hamming na memória FLASH, da qual foi previamente armazenada, e armazena o resultado na memória SRAM.

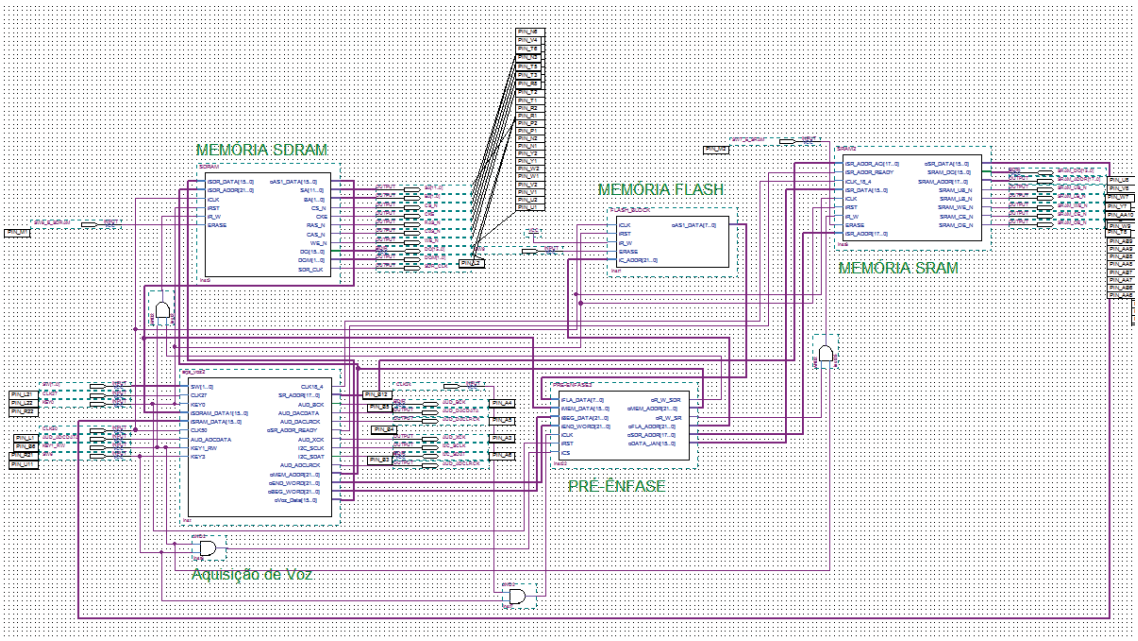


Figura 4.39: *Sistema com bloco de pré-ênfase. (O Autor)*

A Figura 4.40 mostra a etapa de pré-ênfase em funcionamento, onde podemos ver todos os sinais de controle, os barramentos de endereço e de dados das memórias, SDRAM, SRAM e FLASH, bem como sinais internos, como os registradores que armazenam as amostras atual e anterior, usadas no cálculo do filtro de pré-ênfase.

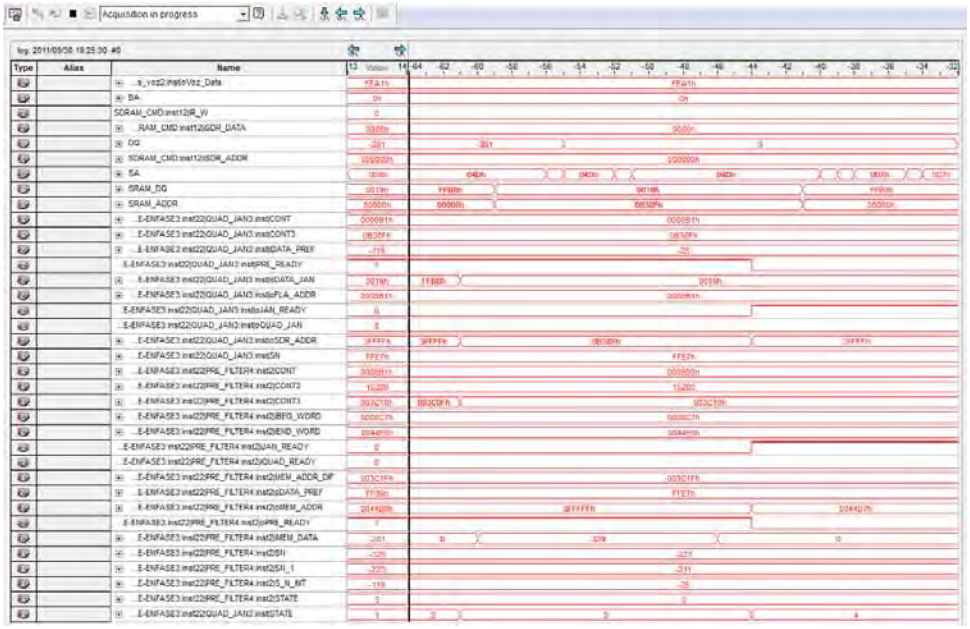


Figura 4.40: *Bloco de pré-ênfase em funcionamento. (O Autor)*

Para se verificar o resultado do bloco de pré-ênfase, foi utilizado um artifício para se transmitir os dados do kit para o computador, para então utilizarmos o *software MATLAB*,

para que assim fosse possível uma visualização gráfica da voz, antes do bloco e após o bloco. Para isso após o processamento da voz e o armazenamento da mesma, foi carregado na FPGA o *software* de demonstração do Kit, no qual era possível realizar um ciclo completo de leitura de cada memória e gravá-los no computador em um arquivo *.hex*, ou seja, puramente hexadecimal. Este arquivo então foi lido em um programa desenvolvido em *MATLAB*, e cada amostra em hexadecimal foi convertido para valores inteiros, e depois estes valores foram plotados em um gráfico, como mostra as Figuras 4.41 e 4.42 abaixo.

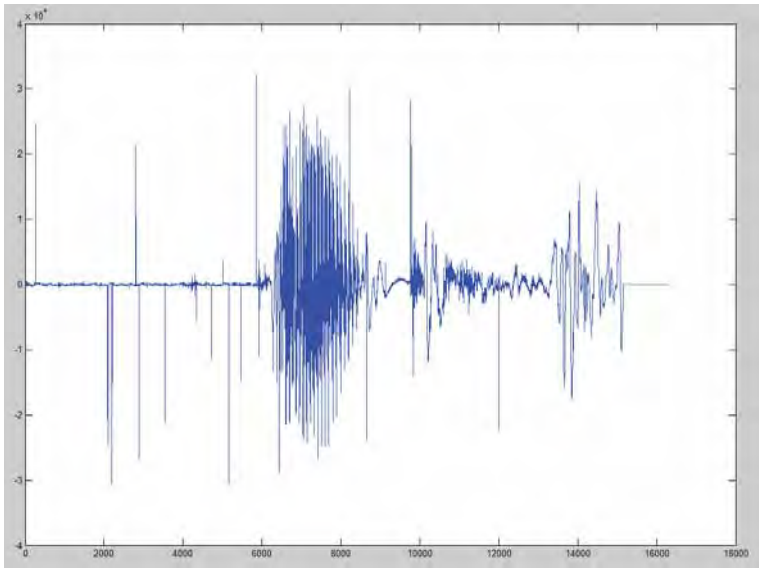


Figura 4.41: *Sinal de Voz adquirido. (O Autor)*

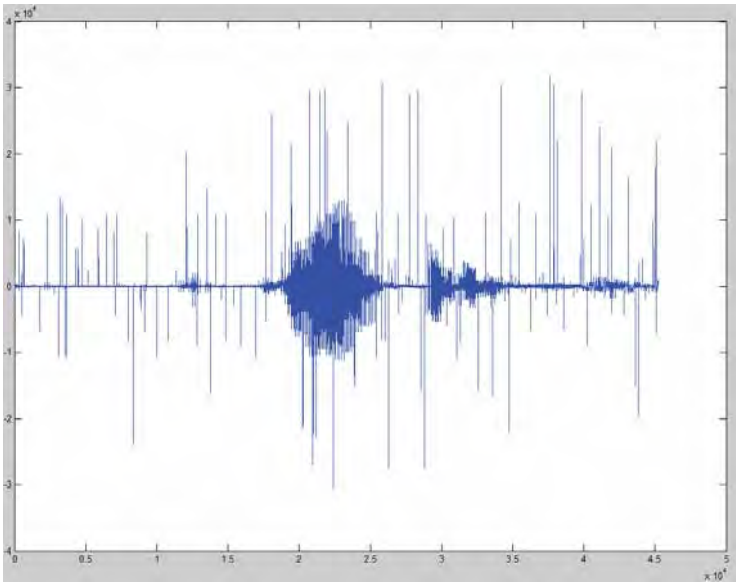


Figura 4.42: *Sinal após a etapa de pré-ênfase. (O Autor)*

Pelos gráficos obtidos, vemos claramente que o sinal após a etapa de pré-ênfase é bem maior que o sinal antes da mesma etapa, pois como explicado no item 4.4 cada quadro foi

armazenado sequencialmente. Pode-se ver também que há certo ruído no sinal adquirido originalmente, este ruído é devido à captação, assim ele também serviu para verificar o funcionamento do bloco de pré-ênfase, pois o mesmo aparece repetidas vezes com diferentes amplitudes, mostrando a sobreposição na etapa de divisão de quadros e janelamento.

5 – CONCLUSÕES E CONSIDERAÇÕES FINAIS

Durante este trabalho, todas as etapas previstas no cronograma de atividades foram realizadas e implementadas. Tais etapas consistiram no estudo do processamento digital de sinais, bem como da teoria das Transformadas de Fourier e dos diversos tipos de filtros digitais e suas implementações. Estudou-se também os conceitos básicos da técnica estocástica de reconhecimento de voz, a HMM, onde foram apresentados os principais problemas da HMM a serem resolvidos, para podermos realizar o reconhecimento da palavra dita pelo locutor, suas soluções e métodos de implementação, bem como técnicas de extração de parâmetros.

Neste trabalho também foram desenvolvidos os blocos em VHDL, referentes ao processo de aquisição da voz e armazenamento da mesma nas memórias presentes no Kit de desenvolvimento e o bloco de pré-processamento. Tais blocos foram devidamente verificados e simulados usando as ferramentas presentes no *software* Quartus II. Podemos ver a partir das informações exibidas no item 4.4, que o processo de aquisição da voz e cálculo da FFT é realizado de maneira simultânea, no sistema de teste da FFT. Esta característica das FPGA's nos mostra que há certo ganho de desempenho do sistema, já que as tarefas são executadas em paralelo, em relação às aplicações que utilizam dispositivos que executem tarefas de maneira exclusivamente sequencial. No sistema com o bloco de pré-ênfase o cálculo do filtro digital é realizado em um único ciclo de *clock*, tendo assim também um ganho de desempenho em relação às mesmas aplicações em questão.

A ferramenta *SignalTap II Logic Analyzer* foi de extrema importância para a solução de *troubleshootings* durante as etapas de descrição e implementação de cada bloco constituinte do sistema. Sem tal ferramenta não seria possível visualizar os registradores internos sintetizados, os quais algumas vezes eram a causa do problema a ser solucionado. Esta ferramenta, porém tem uma desvantagem, a mesma consome muitas macro células da FPGA, quanto mais sinais forem monitorados mais macro células serão consumidas. Assim há um limite para o uso desta ferramenta, se o sistema está consumindo um numero de macro células perto do limite do componente, então poucos sinais poderão ser monitorados.

O sistema desenvolvido consome 6.386 elementos lógicos, os quais correspondem a 34% da capacidade total do componente, e 155.904 bits de memória, correspondendo a 65% da capacidade total do componente, no sistema de teste da FFT. Já no sistema com o bloco de pré-ênfase há um consumo total de 1.506 elementos lógicos, correspondendo à 8% da capacidade lógica do componente, dos quais 277 elementos lógicos correspondem à etapa de pré-ênfase e 589 correspondem à etapa de aquisição de voz.

Assim pode-se concluir que além do sistema ter um ganho de desempenho considerável em relação à outras aplicações, o sistema desenvolvido consome uma quantidade pequena de elementos lógicos do componente, com exceção da FFT, pois foi utilizada uma solução já existente de um fabricante e otimizada, a qual consome grande quantidade de elementos lógicos.

REFERÊNCIAS BIBLIOGRÁFICAS

AMUDHA, V. E VENKATARAMANI, B. E VINOTHKUMAR, R. E RAVISHANKAR, S., ***Software/Hardware Co-Design of HMM Based Isolated Digit Recognition System***, JOURNAL OF COMPUTERS, VOL. 4, NO. 2, 2009.

BAESE, U.M., ***Digital Signal Processing with Field Programmable Gate Arrays 3rd Ed.***, Springer, 2007.

CHU, PONG P., ***RTL Hardware Design Using VHDL: Coding for Efficiency, Portability and Scalability***, John Wiley & Sons, 2006.

CIPRIANO, J.L.G., **Desenvolvimento de Arquitetura Para Sistemas de Reconhecimento Automático de Voz Baseados em Modelos Ocultos de Markov**, Porto Alegre, 2001.

CONCEJERO, C.G. E RODELLAR, V. E MARQUINA, A.A. E ICAYA, E.M. E VILDA, P.G., ***Designing an Independent Speaker Isolated Speech Recognition System on an FPGA***, IEEE, Madrid, 2006.

D'AMORE, R., **VHDL: Descrição e Síntese de Circuitos Digitais**, LTC, 2005.

Datasheet do circuito integrado WM8731_WM8731L

DINIZ, P.S.R. E SILVA, E.A.B. E NETTO, S.L., **Processamento Digital de Sinais: Projeto e Análise de Sistemas**, Bookman, 2004.

ELMISERY, F.A. E KHALIL, A.H. E SALAMA, A.E. E HAMMED, H.F., ***A FPGA-Based HMM For A Discrete Arabic Speech Recognition System***, IEEE, Cairo, 2003.

FILADELFO, F.R, **Desenvolvimento do Cirtuito Para um Kit Didático de Interface com um computador para gravação e simulação de projetos em Dispositivos Lógicos Pragramáveis**, TG – FEG/UNESP, 2003.

FLOYD, T.L., **Sistemas Digitais Fundamentos e Aplicações 9ª ed.**, Bookman, 2007.

HAYES, M.H., ***Digital Signal Processing***, MacGraw-Hill, 1999.

HAYKIN, S. E VEEN, B.V., **Sinais e Sistemas**, Bookman, 2001.

http://fisiogerontologia.blogspot.com/2010/06/reabilitacao-vestibular-uma-revisao.html

http://ocantodocanto.blogspot.com/2007/08/curiosidades.html

http://pt.wikipedia.org/wiki/Cérebro_humano

http://www.altera.com/literature/ug/ug_fft.pdf

http://www.qsl.net/py4zbz/teoria/quantiz.htm

KE, S., et all., ***A HMM speech recognition system based on FPGA, Congress on Image and Signal Processing****, IEEE*, 2008.

MIURA, K., et. all., ***A Low Memory Bandwidth Gaussian Mixture Model (GMM) Processor for 20,000-Word Real-Time Speech Recognition FPGA System***, IEEE, 2008.

RABINER, L.R., AND JUNG, B.H., ***"An introduction to Hidden Markov Model"***, IEEE Assp. Magazine, PP.4-16, Jan 1986.

RABINER, L. E JUANG, B.H., ***Fundamentals of Speech Recognition***, Prentice Hall, 1993.

RABINER, L. R. ***"A tutorial on hidden Markov models and selected applications in speech recognition",*** Proceedings of the IEEE.v.77.n.2, p.257-86, Feb.1989.

SUNG, W., et all., ***Architectural design and implementation of an FPGA softcore based speech recognition system***, IEEE International Workshop on System-on-Chip for Real-Time Aplications, Cairo, 2006.

VARGAS, F.L., et all., ***A fpga based Viterbi algorithm implementation for speech recognition systems***, ICASSP 2001, IEEE International Conference on Acoustics, Speech and Signal Processing.