UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

DIONÍSIO DE CARVALHO

Estudo e desenvolvimento de blocos para processamento *hardwired* em aparelhos de auxílio auditivo com DSP

São Paulo
2013

DIONÍSIO DE CARVALHO

Estudo e desenvolvimento de blocos para processamento *hardwired* em aparelhos de auxílio auditivo com DSP

Dissertação apresentada: Escola de Engenharia de São Carlos da Universidade de São Paulo como pre-requisito para a obtenção do título de Mestre em Ciências, Programa de Engenharia Elétrica.

Área de concentração: Telecomunicações

Orientador: Prof. Dr. João Navarro Soares Junior

Versão Corrigida[1]

São Carlos
2013

[1] Trata-se da versão corrigida da dissertação. A versão original se encontra disponível na EESP/USP que aloja o Programa de Pós-Graduação de Engenharia Elétrica.

AUTORIZO A REPRODUÇÃO TOTAL OU PARCIAL DESTE TRABALHO, POR QUALQUER MEIO CONVENCIONAL OU ELETRÔNICO, PARA FINS DE ESTUDO E PESQUISA, DESDE QUE CITADA A FONTE.

Carvalho, Dionisio
C257e Estudo e desenvolvimento de blocos para processamento hardwired em aparelhos de auxílio auditivo com dsp / Dionisio Carvalho; orientador João Navarro Soares Junior. São Carlos, 2013.

Dissertação (Mestrado) - Programa de Pós-Graduação em Engenharia Elétrica e Área de Concentração em Telecomunicações -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2013.

1. Processamento Digital. 2. Hardwired. 3. Aparelho auditivo. 4. Microeletrônica. 5. Baixo Consumo. I. Título.

FOLHA DE JULGAMENTO

Candidato: Engenheiro **DIONÍSIO DE CARVALHO.**

Título da dissertação: "Estudo e desenvolvimento de blocos para processamento *hardwired* em aparelhos de auxílio auditivo com DSP".

Data da defesa: 22/11/2013

Comissão Julgadora:	**Resultado:**
Prof. Dr. **João Navarro Soares Junior (Orientador)** (Escola de Engenharia de São Carlos/EESC)	APROVADO
Prof. Dr. **Maximiliam Luppe** (Escola de Engenharia de São Carlos/EESC)	Aprovado
Prof. Dr. **Tales Cleber Pimenta** (Universidade Federal de Itajubá/UNIFEI)	Aprovado

Coordenador do Programa de Pós-Graduação em Engenharia Elétrica e Presidente da Comissão de Pós-Graduação:
Prof. Titular **Denis Vinicius Coury**

AGRADECIMENTOS

À minha família, amigos e minha namorada Cintia Silva, que sempre me incentivaram e me apoiaram. Em especial ao Cesar Penteado e Gustavo Cerezo que me ajudaram com a elaboração da dissertação.

Ao meu orientador João Navarro, pelas incansáveis correções e pelas aulas de português.

Ao LSITEC que permitiu o inicio de tudo me dispensando do trabalho por dias ao longo dos anos, e ao LSI que permitiu o uso das ferramentas utilizadas. Em especial ao Fernando Chavez, Hugo Portas, Hugo Daniel, Angélica dos Anjos, Heiner Alarcon, Julio Saldana, Jeferson Soldeira, Armando Ayala, Raul Acosta, Carlos Cavalcanti e ao Paulo Roberto.

À FIT que também me liberou de horas de trabalho para a conclusão do meu mestrado.

Acima de tudo, à minha mãe, Heloisa Helena, ao meu pai, Marcos, e ao meu segundo pai, Delfim, que me deixou a maior saudade que conheço.

RESUMO

CARVALHO, D. (2013**). Estudo e desenvolvimento de blocos para processamento *hardwired* em aparelhos de auxílio auditivo com DSP.** 122 p. Dissertação (Mestrado em Ciências) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2013.

A vida de milhões de pessoas é afetada por problemas de deficiência auditiva, incapacitando-as de ouvirem os sons naturalmente. O uso de aparelhos de auxílio auditivo minimiza o efeito das deficiências, pois possibilita tratamento dos sinais auditivos através de sofisticados algoritmos que eliminam ruídos e amplificam os sinais de interesse. Este trabalho propõem a especificação de um sistema integrado, otimizado em termos de consumo de potência, para realizar o processamento de sinais digitais em aparelhos de auxílio auditivo digital. Foram desenvolvidos dois blocos para processamento *hardwired*, que substituem o processamento realizado por *software*, cuja finalidade é filtrar os sinais sonoros digitalizados com menor consumo. Um dos blocos, um filtro FIR de até 128 coeficientes, pode ser utilizado como filtro do tipo passa baixa ou passa altas frequências. O outro bloco, para executar o algoritmo ALE, é utilizado para eliminar ruídos periódicos. Os blocos desenvolvidos e implementados foram compilados e simulados para comprovar a funcionalidade. Os resultados das simulações mostraram que eles atendem as especificações de funcionalidade. Os blocos foram também sintetizados em uma tecnologia CMOS de 0,35 μm, três níveis de metal, para assim se ter as estimativas de área do circuito e de consumo de potência. A área do *layout* final foi de 14 mm². O consumo de potência estimado é de 0,30 mW para frequência de *clock* de 300 kHz (o que permite que um filtro FIR processe uma amostra a cada 240 μs, no pior caso, e o ALE, uma a cada 36 μs), e de 5,06 mW para frequência de *clock* de 5,0 MHz (filtro FIR processa uma amostra a cada 14,4 μs e o ALE, uma a cada 2,2 μs). As estimativas de consumo foram feitas considerando os dois blocos operando simultaneamente e com tensão de alimentação de 1,8 V. Para todo o sistema integrado proposto, obtive-se, com um cenário específico, o consumo de potência de 1,1 mW, considerando dois Filtros Configuráveis, um Filtro ALE e um DSP.

Palavras-chave: Processamento digital. Hardwired. Aparelho auditivo. Microeletrônica. Baixo consumo.

ABSTRACT

CARVALHO, D. (2013). **Study and development of blocks for hardwired processing in hearing aid devices with DSP.** 122 p. Dissertação (Mestrado em Ciências) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2013.

The live of millions of people are affected by hearing problems, disabling them from hearing the sounds naturally. The use of hearing aids devices minimizes the effect of deficiencies, since it allows processing of auditory signals through sophisticated algorithms that eliminate noise and amplify the signals of interest. This work proposes the specification of an integrated system, optimized in terms of power consumption, to perform digital signal processing in digital hearing aid devices. Were developed two blocks of hardwired processing, replacing software processing, whose purposes are to filter the digitized audio signals with lower consumption. One of the blocks, an FIR filter up to 128 coefficients can be used as a low pass or high pass filter. The other block, to run the ALE algorithm, is used to eliminate periodic noises. The blocks developed and implemented were compiled and simulated to demonstrate their functionality. The simulation results show that they meet the specifications of functionality. The blocks were also synthesized in a 0.35 μm CMOS technolog, three metal levels, in order to have estimatives of circuit area and power consumption. The area of the final layout was 14,0 mm². The estimated power consumption is 0.30 mW for clock frequency of 300 kHz (which allows a FIR filter to process one sample every 240 μs in the worst case, and ALE, one every 36 μs), and 5.06 mW for clock frequency of 5.0 MHz (FIR filter processing one sample every 14.4 μs, and ALE, one every 2.2 μs). Consumption estimates were made considering the two blocks operating simultaneously and supply voltage of 1.8 V. For all the proposed integrated system, it was found, for a specific scenario, the power consumption of 1.1 mW, considering two configurable filters, one filter ALE and one DSP.

Keywords: Digital processing. Hardwired. Hearing aid. Microelectronics. Low consumption.

Lista de Figuras

Figura 1: Área de Audição....3
Figura 2: Consumo de potência ao longo do tempo....5
Figura 3: Porcentagem de aparelhos digitais versus o ano....11
Figura 4: Estilos de AAA....11
Figura 5: Microfone Direcional....15
Figura 6: Resposta do microfone direcional....16
Figura 7: *Array* microfone omniderional....17
Figura 8: Sinal de um segmento de fala....18
Figura 9: Envelope de um segmento de ruído gaussiano branco....18
Figura 10: Aparelho AAA com realimentação acústica....21
Figura 11: Codec de Entrada....25
Figura 12: Fluxo de sinal para processamento digital....27
Figura 13: Codec de Saída....28
Figura 14: Plataforma TI....29
Figura 15: GE3280- Diagrama de Blocos –Plataforma da Gennum....32
Figura 16: EZAIRO- Diagrama de Blocos –Plataforma....34
Figura 17: Números de DSP's vendidos versus no ano....37
Figura 18: Potência consumida versus capacidade de processamento para as três diferentes formas de implementação de sistemas para processamento de sinais....39
Figura 19: *Cluster* genérico do GC5003....48
Figura 20: Sistema proposto para processamento digital de um AAA....52
Figura 21: Especificação das variáveis para se projetar filtro FIR....55
Figura 22: Resposta de um filtro passa baixas no domínio da frequência. Especificações: F_{pass} = 4,0 kHz e F_{stop} = 6,0 kHz, A_{pass} = 0,01 dB e A_{stop} = 80 dB....57
Figura 23: Resposta impulsiva do filtro passa baixas. Especificações: F_{pass} = 4,0 kHz e F_{stop} = 6,0 kHz, A_{pass} = 0,01 dB e A_{stop} = 80 dB....57
Figura 24: Arquitetura do filtro configurável....59
Figura 25: Interface do filtro configurável....60
Figura 26: Estrutura do Filtro ALE....63
Figura 27: Sinal de entrada, sinal de entrada mais ruído em 1,0 kHz e saída do filtro ALE implementado em Matlab....65

Figura 28: Sinal de erro do filtro ALE implementado em Matlab 65
Figura 29: Arquitetura do filtro ALE implementado em Verilog 66
Figura 30: Interface do filtro ALE 68
Figura 31: Máquina de estado do filtro ALE que controla os processos do filtro 69
Figura 32: Simulação de dois sinais entrando no filtro (Modelsim). Sinal do canal 1 de 3,5 kHz e do canal 2 de 4,0 kHz 75
Figura 33: Simulação de sinais entrando no filtro: Sinal do canal 1 de 200 Hz e do canal 2 de 7,0 KHz (Modelsim) 76
Figura 34: Diferença da resposta em frequência do filtro 8 com a representação dos coeficientes com 32 *bits* (linha pontilhada) e 16 *bits* (linha contínua) 77
Figura 35: Sinais de entrada dos Filtros ALE. Sinal Xi criado pelo Aledemo (puro) e Xin (puro + ruido) (Modelsim) 79
Figura 36: Sinais de saída dos Filtros ALE (Modelsim) 79
Figura 37: Diferença entre as saídas das instancias do Filtro ALE (Modelsim) 80
Figura 38: Simulação do Filtro ALE com um sinal senoidal composto (Modelsim) 81
Figura 39: Diagrama gerado da síntese do Filtro Configurável e do Filtro ALE (RLT Compiler) 84
Figura 40: Esquemático da máquina de estado do Filtro ALE (RTL compiler) 85
Figura 41: Posicionamento das células do *netlist* Top_filtro (Soc Encounter) 86
Figura 42: Células posicionadas para o roteamento (SoC Encounter) 87
Figura 43: *Layout* final dos coprocessadores, Filtro Configurável e Filtro ALE (SoC Encounter) 88
Figura 44: Proposta de *Pipeline* para processamento em um AAA 90

Lista de Tabelas

Tabela 1: Resumo das características das plataformas descritas 34
Tabela 2: Comparação entre arquiteturas Superscalar e VLIW 41
Tabela 3: Valores de desempenho do núcleo Icyflex4 sintetizado com duas unidades de VPU 46
Tabela 4: Valores típicos de consumo do *layout* do núcleo Coolflux 47
Tabela 5: Relação entre número de coeficientes e número de ciclos de *clock* para o Filtro Configurável 73
Tabela 6: Especificação de filtros FIR passa baixas e passa altas frequências, projetados no FDATool 74
Tabela 7: Valores máximos e mínimos dos sinais do canal 1 e canal 2 aplicados ao filtro 3 da tabela 6 75
Tabela 8: Valores máximos e mínimos dos sinais do canal 2 do filtro 8 76
Tabela 9: Valores máximos e mínimos dos sinais Xi e Xin 80
Tabela 10: Valores máximos e mínimos da saída do Filtro ALE 81
Tabela 11: Resultados das Sínteses 83
Tabela 12: Estimativa de consumo de potência para diferentes frequências de *clock* aplicados aos coprocessadores implementados 89
Tabela 13: Fluxo das amostras no *Pipeline* para processamento em um AAA. Os estágios de codec, PB e DSP trabalham com um conjunto de duas amostras, uma de cada canal de entrada 90
Tabela 14: Resultados de área e consumo para o sistema proposto configurado com e sem *pipeline* 93

Sumário

Lista de Figuras ..... **VII**

Lista de Tabelas ..... **IX**

Capítulo 1 - INTRODUÇÃO **1**

1.1- CONTEXTO **4**

1.2- OBJETIVO **7**

1.3- JUSTIFICATIVA **7**

1.4- METODOLOGIA **8**

1.5- DESCRIÇÕES DOS CAPÍTULOS **8**

Capítulo 2 - APARELHO DE AUXÍLIO AUDITIVO E DSP **9**

2.1-APARELHO DE AUXÍLIO AUDITIVO **9**

2.1.1 O processamento digital – Algoritmos *12*

2.1.2 Considerações do Processamento Digital *23*

2.1.3 Fluxo do Sinal *24*

2.1.4 Plataformas para desenvolvimento de AAA's Digitais *28*

2.2- DIGITAL SIGNAL PROCESSOR – DSP **35**

2.2.1 GPP, ASIP e ASIC *37*

2.2.2 DSP – Arquiteturas *39*

2.2.3 Memória *42*

2.2.4 DSP – Estudos de Caso *43*

2.3- AAA'S E DSP'S **49**

Capítulo 3 - PROPOSTA DE SISTEMA – COPROCESSADORES **51**

3.1 SISTEMA – COPROCESSADORES **51**

3.2 DSP E COMUNICAÇÃO **53**

3.3 FILTRO CONFIGURÁVEL – FIR **54**

3.3.1 Matlab – FDATool *56*

3.3.2 Arquitetura do filtro configurável implementado *58*

3.3.3 Interface e Operação *60*

3.3.4 Considerações Finais *61*

3.4 FILTRO ALE **62**

3.4.1 Least Mean Square – LMS *63*

3.4.2 Matlab – ALEDEMO *64*

3.4.3 Arquitetura do filtro ALE implementado *66*
3.4.4 Interface e Operação..... *67*

Capítulo 4 - SIMULAÇÕES E RESULTADOS **71**

4.1 FILTRO CONFIGURÁVEL **72**
4.1.1 Resposta Impulsiva – Filtros Passa Baixa..... *73*
4.1.2 Sinais Aplicados e Respostas *74*
4.1.3 Análise dos Resultados..... *77*
4.2 FILTRO ALE **78**
4.2.1 Sinais Aplicados e Respostas *78*
4.2.2 Análise dos Resultados..... *82*
4.3 SÍNTESE DOS COPROCESSADORES - *BACKEND* **83**
4.3.1 RTL Compiler..... *83*
4.3.2 SoC Encounter *85*
4.3.3 Análise do consumo de potência *89*

CONCLUSÕES E TRABALHOS FUTUROS **95**

5.1 CONCLUSÕES..... **95**
5.2 TRABALHOS FUTUROS **97**

REFERÊNCIAS BIBLIOGRÁFICAS **99**

APÊNDICE A – Código verilog filtro configurável **103**

APÊNDICE B – Código Verilog filtro ALE **109**

APÊNDICE C – Códigos Matlab..... **121**

Capítulo 1 - INTRODUÇÃO

O desenvolvimento da indústria de microeletrônica tem como possível desdobramento a melhoria da qualidade de vida daqueles que dela se beneficiam e sua evolução segue com a utilização de Circuitos Integrados (CI) com transistores CMOS *(Complementary metal–oxide–semiconductor)* cada vez menores. Hoje em dia, os transistores estão com dimensões nanométricas, mais rápidos e com tensões de alimentação cada vez menores, o que possibilita equacionar o problema de unir eficiência de consumo com poder de processamento de um sistema. Um sistema é composto por um ou mais CI's com diferentes funcionalidades. O poder de processamento de um sistema esta associado à sua frequência de operação, conhecida como frequência de *clock*, sob a qual o sistema realiza as operações básicas, como multiplicação e soma. Para atender a demanda de maiores funcionalidades, procura-se aumentar a frequência do *clock* o máximo possível e com isso aumentar o número de operações realizadas num determinado intervalo de tempo. Um dos fatores limitantes para o aumento da frequência de *clock* é o consumo associado aos blocos que implementam as funcionalidades do sistema e ao circuito de distribuição do *clock*. Visando uma alta eficiência energética, atualmente, utilizam-se diversos meios que garantam que o sistema desenvolvido atenda aos requisitos de funcionalidade, mas operando sob um *clock* que não irá fazer do consumo de potência um problema.

A tecnologia CMOS é predominante no desenvolvimento de CI's devido a sua alta densidade de integração e baixo consumo de potência, o que possibilita projetos altamente complexos com dimensões muito reduzidas, conhecidos como VLSI *(Very Large Scale Integration)*. Os VLSI's permitem o desenvolvimento de sistema alimentado à bateria com maior autonomia e maior portabilidade, isso sem sacrificar a capacidade de processamento de dados. Para uma ideia do grau de integração que se atingiu atualmente, a Intel lançou em 2012 um processador de 434 mm² de área com algo em torno de 2,27 bilhões de transistores. A Nvidia, por sua vez, lançou no mesmo ano um processador de 561 mm² com algo em torno de 7,1 bilhões de transistores (INTEL, 2013; NVIDIA, 2013). A evolução da tecnologia CMOS fica melhor explicita se compararmos este processador atual da Intel com o primeiro CI para processamento lançado em 1971, o Intel 4004, que possuía menos de 2.300 transistores para uma área de 12,0 mm².

Neste nicho tecnológico, o Brasil ainda está em processo de emergência, contando com algumas *Design Houses* (DH's) de CI's, como o Laboratório de Sistemas Integrados Tecnológicos (LSITEC, 2011). Além das DH's, o Brasil trabalha no desenvolvimento de uma *foundry* para a de fabricação de CI's (CEITEC, 2011), fechando assim o ciclo de produção de dispositivos microeletrônicos. Podemos citar também a Sixsemicondutores, que fabricará CI's para utilização em aplicações industriais e médicas (SIXSEMICONDUTORES, 2013). Com o ciclo de produção bem estabelecido, espera-se que o país possa ter diversos aparelhos e equipamentos mais acessíveis a população, devido à redução dos custos pela produção nacional.

Um exemplo importante destes é o Aparelho de Auxílio Auditivo (AAA). O IBGE de 2005 (2005) identificou no Brasil quase seis milhões de pessoas com alguma deficiência auditiva e o trabalho (PENTEADO, 2009) mencionou um número ainda maior, extrapolando para 10 milhões de deficientes em uma população de 196 milhões brasileiros. A deficiência auditiva impacta a qualidade de vida do paciente de diferentes formas para diferentes idades. Uma criança, por exemplo, terá seu aprendizado em uma sala de aula prejudicado. Um adulto, por outro lado, normalmente apresentará índices maiores de frustração, paranoia, insegurança, sentimento de inferioridade e depressão do que o resto da população. Um idoso com perda progressiva da audição será levado a um isolamento cada vez maior. Existem diversos fatores que causam a deficiência auditiva, como hereditariedade, acidentes ou traumas acústicos, excesso de ruído, doenças e drogas ototóxicas (PENTEADO, 2009). O fator mais comum é a presbiacusia que é a degeneração auditiva pela idade.

Com o auxílio da tecnologia, podem ser minimizados os efeitos desta deficiência nos portadores que poderão ter um melhor convívio social e desenvolvimento pessoal. Para o tratamento de uma anormalidade auditiva, o primeiro passo é o correto diagnóstico, obtido por meio de um exame de audiometria que avalia a audição. Quando detectada qualquer anormalidade realiza-se então a avaliação do problema para determinar o tipo de alteração e seu grau (ALISOM, 2011). Este exame também é utilizado na configuração do AAA para cada paciente, sendo por ele determinadas as faixas de frequência a serem tratadas pelo aparelho, o ganho mínimo que será necessário e os principais algoritmos que serão utilizados para trazer maior inteligibilidade e conforto com a utilização do aparelho.

A audição humana possui limites definidos quanto ao nível de intensidade sonora, pressão sonora e as frequências dos sinais:

- Intensidade: o limite mínimo é o valor mínimo que o nosso ouvido consegue detectar, sendo 10^{-12} W/m^2, equivalente a zero dB, o limiar da audibilidade. O limite máximo

surge quando a percepção do som começa a causar dor ao ouvinte, sendo 1,0 W/m^2, equivalente a 120 dBs, o limiar da dor. A deficiência auditiva devido à intensidade pode ser classificada de perda leve, quando se tem a incapacidade de ouvir sons abaixo de 30 dB, até de perda profunda, quando se tem a incapacidade de ouvir sons abaixo de 95 dB. Um AAA é indicado para este tipo de perda auditiva;

- Pressão: o ouvido é muito sensível a pequenas variações de pressão, sendo a mínima de 2,0*10^{-5} Pa e a máxima de 20 Pa. Uma deficiência auditiva ou trauma é causado pela exposição prolongada a ruídos com alta pressão sonora, que impacta na perda da capacidade de se ouvir sons de baixa pressão. Esta deficiência ocupa o segundo lugar entre as deficiências mais frequentes do aparelho auditivo humano, perdendo apenas para a presbiacusia. Normalmente, este tipo de deficiência é irreparável e é descartada a melhora da audição pelo uso de AAA;
- Frequência - A audição humana consegue captar sinais cujas frequências estão entre 20 Hz e 20 kHz. Pacientes com perda auditiva em altas frequências podem utilizar um AAA para fazer o deslocamento do sinal para baixas frequências e o mesmo vale para aqueles com perda em baixas frequências.

A área de audição mostrada na figura 1 define o plano dos sons audíveis, resumindo as especificações mencionadas acima.

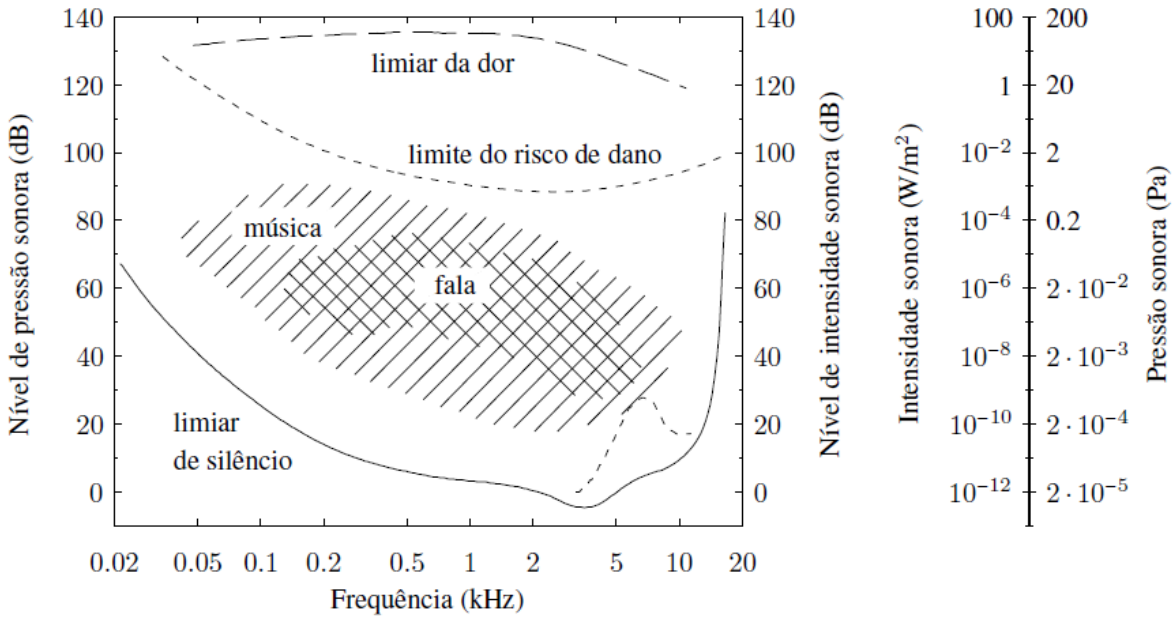

Figura 1: Área de Audição
Fonte: Zwicker (2007)

Um exemplo de redução da área de audição é a variação do limiar de silêncio, dada pela linha pontilhada no lado direito baixo, que pode ser causada pela exposição prolongada à música em volume elevado. Ocorrendo o aumento deste limiar, os sons devem ser amplificados para que adentrem na área de audição.

Mesmo com a capacidade de se ouvir sinais com frequências até 20 kHz, os sons da fala humana situam-se principalmente entre as frequências de 250 e 8,0 kHz, variando entre 20 e 60 dB de intensidade. Para comunicações telefônicas, usa-se ainda uma faixa menor de frequências, entre 300 a 3,4 kHz. Para se definir esta faixa de frequência foram levados em conta à inteligibilidade e a energia da voz. Verificou-se que na faixa de 100 a 1,5 kHz esta concentrada 90% da energia da voz humana, enquanto que acima de 1,5 kHz esta concentrada 70% da inteligibilidade das palavras. Baseado num compromisso entre estes dois valores, foi escolhida a faixa para comunicações telefônicas que garante 85% de inteligibilidade e 68% de energia da voz recebida pelo ouvinte (OLIVEIRA, 2007). Como será explicado mais a frente, estas informações tem grande relevância no projeto de aparelhos para auxílio auditivo. Como o objetivo de um AAA é tratar principalmente o som da fala ou os sinais com baixas frequências, pode-se trabalhar com banda de frequência reduzida de 20 kHz para 8,0 kHz.

1.1- Contexto

O AAA é um sistema de baixo consumo e reduzida área que realiza processamento em tempo real, recebendo um fluxo de dados, provenientes do processo de amostragem dos sinais sonoros, que serão manipulados através de sofisticados algoritmos de Processamento Digital de Sinal (PDS).

Qualquer sistema de processamento digital pode ser desenvolvido usando processadores programáveis ou *hardware* customizados, que serão fabricados usando uma tecnologia de circuitos VLSI (PARHI, 1999). Sistemas que usam processadores programáveis tem seu processamento realizado por *Software,* o que significa que há um algoritmo (conjunto de instruções) que será executado no processador. Sistemas com *hardware* customizados tem o processamento realizado por *hardware,* o que implica em que o algoritmo ou parte dele foi descrito em uma linguagem de descrição de *hardware* (Verilog ou VHDL[1]) e foi sintetizado

[1] A sigla VHDL significa Linguagem de Descrição de Hardware VHSIC (do Ingles VHSIC Hardware Description Language), e a sigla VHSIC significa Circuitos Integrados de Velocidade Muito Alta (do inglês *Very High Speed Integrated Circuits*)

para uma tecnologia de circuitos integrados, dando origem a um sistema com processamento *hardwired*. O termo *hardwired* é utilizado para se referir a blocos dedicados de processamento em *hardware* cuja funcionalidade é definida pelo mapeamento direto de um determinado algoritmo. Sabendo que, no geral, o objetivo de um sistema é maximizar o desempenho mantendo o custo baixo, fornecendo o correto processamento com o menor recurso possível, as melhores soluções de desenvolvimento muitas vezes implicam num sistema hibrido com ambos os processamentos.

Muitas vezes, para se atingir o processamento que uma aplicação exige sem ter elevado consumo, desenvolve-se uma arquitetura totalmente dedicada, chegando-se ao projeto de um ASIC (*Application Specific Integrated Circuit*), ou, a partir de um sistema programável, desenvolve-se alguns blocos adicionais para processamento *hardwired.* Isso ocorre porque o processamento por *hardware* é mais eficiente que o processamento por *software* em termos de velocidade e consumo, uma vez que para a execução dos algoritmos não há acessos à memória para buscas de instruções, acessos a unidades de operações aritméticas, entre outros procedimentos que seriam realizados por um processamento por *software*. Na figura 2 é mostrado o consumo de potência no processamento de sinais auditivos, quando realizado por um Processador Digital de Sinal, cuja sigla em inglês é DSP[2], genérico; por um DSP *hardwired*, que é um processador com arquitetura especializada para os algoritmos aplicados em aparelho auditivo; ou por um circuito analógico. Segundo Kates (2008) a implementação *hardwired* reduz por um fator de até três o consumo quando comparado a implementação em um DSP genérico.

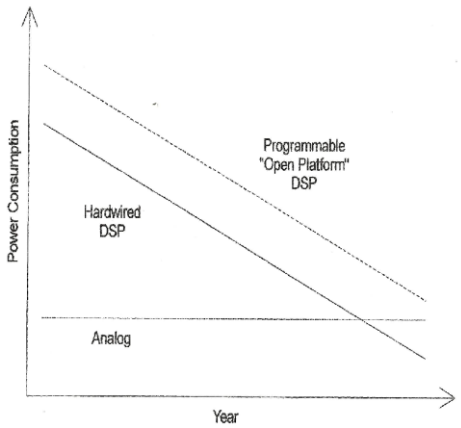

Figura 2: Consumo de potência ao longo do tempo
Fonte: Kates (2008)

[2] DSP é a sigla para *Digital Signal Processor* ou Processador Digital de Sinal. Neste trabalho tem-se

Como desvantagens do processamento *hardwired*, podemos mencionar a diminuição da flexibilidade do sistema, devido a impossibilidade de se alterar um algoritmo que está implementado em *hardware* (sintetizado), o aumento da área de um produto final pelo acréscimo de funcionalidades agora em *hardware* e também ao aumento do tempo de desenvolvimento, uma vez que o desenvolvimento de *hardware* é mais demorado que o desenvolvimento de *software*, considerando a mesma aplicação. Observamos que há um t*rade-off* entre implementações por *hardware* ou *software* e características do sistema. *Trade-off* é um termo muito utilizado em desenvolvimentos de CI's e significa o compromisso de projeto em situações em que há conflito de escolhas e tem que haver ponderações nas tomadas de decisão. No que concerne a projetos de CI's, os conflitos de escolhas ocorrem comumente devido a três parâmetros: desempenho, consumo de potência e área (custo). Pode-se adicionalmente considerar como parâmetro a flexibilidade do sistema e o tempo de desenvolvimento. Sempre que se favorece um dos parâmetros, os outros são excedidos. Por exemplo, caso se busque alto desempenho, pode-se utilizar a paralelização do processamento, o que aumentaria a área do circuito implementado, ou, alternativamente, pode-se aumentar a frequência de *clock*, o que aumentaria o consumo,

A discussão entre processamento por *software* ou *hardware* compõe o cenário deste trabalho, que buscará especificar um sistema de processamento para um aparelho de auxílio auditivo. A necessidade de processamento em tempo real, característico desta aplicação, em conjunto com a necessidade de ultra-baixo consumo de potência, devido ao uso de bateria, levam a uma solução de sistema com o processamento feito diretamente em vários blocos de *hardware.* Considerações como flexibilidade para permitir a execução de diferentes algoritmos e a restrição de área para o produto final, levam a um sistema com diversas funcionalidades implementadas em *software.* Propor um sistema que concilie as vantagens de cada maneira de processamento com o mínimo de desvantagem, mantendo os compromissos exigidos (processamento, área, consumo, flexibilidade) e o correto funcionamento do produto, é o foco deste trabalho.

1.2- Objetivo

O objetivo principal deste trabalho é a especificação de um sistema integrado (*System on Chip* - SoC), otimizado em termos de consumo de potência, para realizar o processamento de sinais digitais para AAA's e a implementação de blocos para processamento *hardwired* que auxiliarão um DSP na correção e melhoria do sinal auditivo. Tal sistema busca atingir alto poder de processamento dentro das especificações de consumo e área que a aplicação exige e manter a flexibilidade que é necessária aos aparelhos auditivos devido à contínua evolução de programas utilizados.

1.3- Justificativa

- Desenvolvimento de um sistema otimizado para uma aplicação que requeira alimentação por bateria e alto poder de processamento. Com isso o mestrando aprofundará seus conhecimentos de arquiteturas de DSP's assim como desenvolvimento de coprocessadores;
- A fabricação de produtos no âm*bit*o microeletrônico requer planejamento preciso quanto ao tempo de desenvolvimento. A abordagem de projeto proposto neste trabalho garante o *time-to-market* de um AAA, pois com ela se pode usar um DSP já existente e bem conhecido, o qual será otimizado por blocos dedicados de co-processamento;
- Sendo objetivo do projeto o baixo consumo, o processamento de certas funcionalidades através de blocos específicos possibilita que o DSP trabalhe em uma frequência de *clock* menor, uma vez que será menos exigido, e assim consuma menor potência (o consumo está diretamente relacionado à frequência de operação).

1.4- Metodologia

A abordagem seguida para o desenvolvimento do trabalho foi partir do estudo de AAA's, sua estrutura e algoritmos utilizados em processamento de sinais digitais, como exemplo algoritmos de cancelamento de realimentação. Com isso deseja-se estar apto para propor blocos dedicados para realizar o processamento de certos algoritmos e, assim, compor todo um sistema para aplicações em AAA's.

Em seguida o trabalho focará no estudo dos DSP's em busca de determinar quais são as principais características, os principais blocos funcionais e como são implementados. Com isso deseja-se poder determinar as características chaves na arquitetura de um DSP para ser utilizados em um AAA que contenha blocos de processamento específico.

A escolha de quais funções serão implementadas em *hardware* substituindo o processamento pelo DSP é o grande desafio neste trabalho.

É importante destacar que o sistema será especificado realizando a otimização de um DSP através do acoplamento de blocos de processamento *hardwired*, considerando os algoritmos por ele processados e tendo como meta o desenvolvimento de um AAA com menor consumo de potência.

1.5- Descrições dos Capítulos

No capítulo 2 serão apresentadas as informações pesquisadas e estudadas sobre AAA's e DSP's.

No capítulo 3 será apresentado um SoC para ser utilizado em AAA's, onde o processamento de sinais digitais é realizado por um DSP e blocos para processamento *hardwired.* Serão detalhadas a implementação e a funcionalidade dos blocos implementados.

No capítulo 4 serão apresentados e analisados os resultados obtidos por simulações, ao nível RTL (*Register Transfer Level*), dos blocos implementados e o circuito resultante da síntese.

Por fim, no capítulo 5, serão apresentadas as conclusões desta dissertação e trabalhos futuros.

Capítulo 2 - APARELHO DE AUXÍLIO AUDITIVO E DSP

Este capítulo condensa todas as informações pesquisadas e estudadas sobre AAA's e sobre DSP's. Iniciando pelos AAA's, apresentaram-se a evolução deles, desde os aparelhos com processamento analógico até os aparelhos com processamento digital, que tem hoje um número crescente de usuários, as variações dos aparelhos conforme as deficiências e os principais algoritmos para a correção do problema auditivo e melhora da inteligibilidade. Também se apresentaram as restrições dos AAA's como tamanho e consumo de potência. Toda a estrutura de um AAA, processamentos realizados, fluxo de dados será então detalhada.

Em seguida, no capítulo serão resumidas as pesquisas realizadas sobre DSP's, o grande responsável pela evolução dos AAA's e onde se concentram os maiores esforços para otimizações. Também serão apresentados três possíveis sistemas para executar algoritmos de PDS e será discutida a estrutura das arquiteturas de DSP's, enfatizando as diferenças entre as microarquiteturas Superscalar e VLIW. Por fim, serão apresentados alguns estudos de caso de DSP's com suas respectivas arquiteturas.

2.1-Aparelho de Auxílio Auditivo

AAA's são utilizados para resolver, ou ao menos minimizar, problemas de pessoas com a audição limitada ou com algum tipo de deficiência, que pode ter sido originado desde um problema congênito a algum trauma. O objetivo dos AAA's é aumentar a inteligibilidade e compreensão dos sons. A ideia básica é captar o som natural, processá-lo ("amplifica-lo") para aumentar a possibilidade de sua percepção, e aplica-lo diretamente no sistema auditivo do paciente. Para tanto, os componentes básicos de um aparelho auditivo são: o microfone, para captação do sinal sonoro e conversão em um sinal elétrico; a unidade de processamento, para a correção; o alto-falante, para conversão do sinal elétrico em sinal sonoro e sua aplicação; e a bateria, para alimentação dos componentes. Em se tratando da captação e aplicação do som não se tem grandes mudanças desde os primeiros aparelhos, isto é, os microfones e alto-falantes não sofreram grandes alterações. Porém no que concerne ao processamento dos sinais, ocorreram grandes transformações, partindo dos aparelhos com processamento analógico, com poucas opções de processamento, e terminando nos aparelhos

com processamento digital, com possibilidades de aplicação de complexos processamentos através de algoritmos sofisticados.

A primeira opção de AAA foi o aparelho analógico, sendo que os primeiros aparelhos utilizando transistores foram lançados na década de 50 e consistiam basicamente de microfone, amplificadores, alto-falante e bateria. A opção de processamento disponível nestes aparelhos era, principalmente, a especificação da resposta do ganho *versus* frequência. O grande problema deles era a amplificação tanto do som de interesse quanto dos ruídos ou sons do ambiente de não interesse, o que causava incômodos ao usuário.

Para o mesmo modelo de aparelho podia-se ter componentes internos com diferentes especificações para atingir a necessidade de amplificação requerida e assim atender a demanda do paciente. O ajuste para a resposta do ganho era tipicamente feita por uma chave e potenciômetros (*trim-pots*) eram fornecidos para dar o controle sobre o limiar de compressão ou limitar a potência de saída. Alguns aparelhos, um pouco mais complexos, já apresentavam geradores de tons e controle de volume mais acessível.

Os aparelhos analógicos, no geral, são eficientes em ambientes silenciosos, onde o paciente tem como fonte sonora apenas o que lhe interessa, uma pessoa falando ou um rádio, porém são inadequados para ambientes com diversas fontes sonoras, como qualquer ambiente público. Existem pessoas que ainda preferem o aparelho analógico, pois o som lhes soa mais natural, enquanto que os aparelhos digitais apresentam um som que lhes parece “robotizado”.

Frente à limitação de processamento dos aparelhos analógicos e com a evolução dos DSP’s e também dos algoritmos de PDS, possibilitando cada vez mais o tratamento específico para cada deficiência, o AAA digital é atualmente predominante. Segundo o trabalho Lamenza (2005), as principais vantagens do aparelho digital sobre o analógico são: capacidade de programação e precisão de ajustes, controle da realimentação acústica, redução do ruído, ajustes automáticos em função das mudanças ambientais, baixa distorção e excelente qualidade sonora. Além do processamento e melhoramento do som captado, os aparelhos digitais podem gerar sinais (tons) sonoros para auxiliar em exames de audiometria. A grande desvantagem é maior custo dos aparelhos digitais, o que torna difícil seu acesso em comunidades carentes.

O primeiro aparelho auditivo digital foi apresentado pela Nicolet Instrument Corp. em 1989, porém, somente em 1996 que a Oticon e Widex apresentaram os aparelhos BTE e ITE (tipos de aparelhos explicados a seguir) (AUDIBLE, 2012). Na figura 3 é mostrada a evolução na utilização de AAA’s digitais, que em 1996 correspondia a menos de 10% dos aparelhos vendidos e em 2005 já correspondia a mais de 90%.

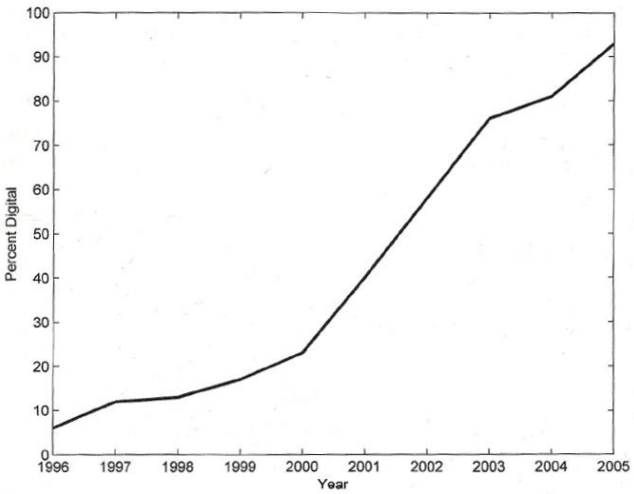

Figura 3: Porcentagem de aparelhos digitais versus o ano
Fonte: Kates (2008)

A escolha do estilo e também do tamanho do AAA depende de fatores como a quantidade de amplificação necessária, o custo, a vaidade e a destreza manual do usuário. Para cada estilo de AAA há imposições no projeto do aparelho, como por exemplo, no caso do modelo CIC, onde o aparelho é completamente introduzido no canal auditivo, a restrição de área de projeto é a mais impactante. Os quatro principais estilos de AAA disponíveis são (Figura 4):

- *Behind-the-ear* (BTE);
- *In-the ear* (ITE);
- *In-the-canal* (ITC);
- *Completely-in-the canal* (CIC).

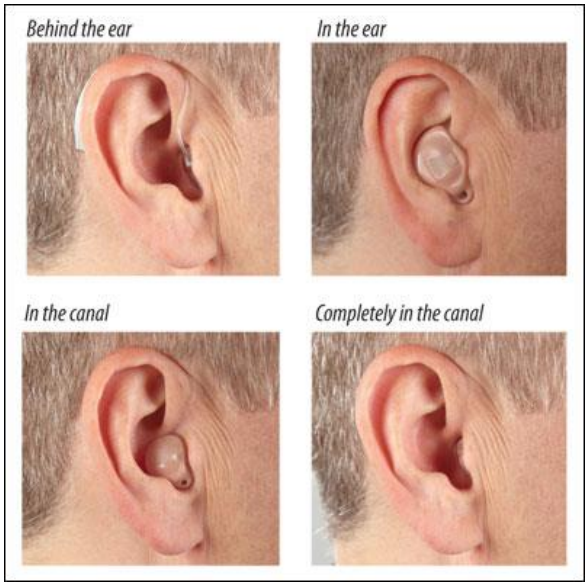

Figura 4: Estilos de AAA

Um AAA digital é, em sua forma mais simples, composto dos seguintes blocos:

- Microfone, entrada direta de áudio ou *telecoil* para a aquisição do som. Realiza a captação e conversão do som para sinal elétrico que será posteriormente convertido de analógico para digital. *Telecoil* refere-se a uma bobina para recepção de sinais de televisão, telefone e comunicação sem fio;
- Codec, acrônimo de Codificador/Decodificador. É um bloco de *hardware* ou *software* que faz a codificação e decodificação de sinais. No AAA o Codec compõe o caminho de entrada (codec de entrada) e de saída (codec de saída) do sinal. De forma geral o caminho de entrada é composto por amplificadores para os sinais recebidos e por conversor do sinal analógico para digital (ADC), o qual fornece um pacote de *bits* ao DSP. Este, por sua vez, retornará outro pacote de *bits* que foram processados pelos algoritmos. O caminho de saída é composto por filtros, por controlador de ganho, por interpolador, por conversor do sinal digital para analógico (DAC) e por um amplificador de saída;
- *Digital Signal Processing (DSP)*, onde se aplicam os algoritmos de tratamento do sinal. Os DSP's são processadores com uma arquitetura especificada, que possibilitaram grande evolução nos AAA's.
- Alto-falante (*speaker*), usado na saída do aparelho, convertendo o sinal elétrico em som audível;
- Memórias, para armazenamento de configurações e programas. São necessários dois tipos de memória: 1) memória não volátil, tipicamente EEPROM (*Electrically-Erasable Programmable Read-Only Memory*) ou mesmo FLASH, que irá armazenar dados que não podem ser perdidos como os programas e o *datalog*. *Datalog* consiste em dados armazenados de tempos em tempos com as configurações do aparelho naquele momento.

2) memórias voláteis, RAM (*Random Access Memory)*, utilizadas no DSP para realizar os processamentos;

- Bateria, fonte de energia. Em aparelhos auditivos, o tipo de bateria mais comum é o *zinc-air*.

2.1.1 O processamento digital – Algoritmos

Um algoritmo é uma sequência de instruções com o objetivo de solucionar um problema e produzir um resultado dentro de um intervalo de tempo finito com uma quantidade de esforço também finita. O tempo e o esforço na execução de um algoritmo

dependem basicamente de dois fatores: a maneira como o algoritmo foi escrito e o quanto de recursos há disponível para executá-lo. Os algoritmos podem ser escritos em diversas linguagens, desde as de alto nível, como por exemplo, C/C++ ou Java, até as de baixo nível, como Assembly. Linguagens de baixo nível podem gerar algoritmos mais eficientes na utilização dos recursos de *hardware*, porém são mais difíceis de escrever e corrigir, exigindo do programador um tempo maior para o desenvolvimento dos programas, que é o conjunto de algoritmos para uma aplicação específica. Arquiteturas de *hardware* muito customizadas tem um conjunto de instruções (*Instructions Set Architecture*, ISA) bem particular e por isso, muitas vezes, a melhor opção para se aproveitar ao máximo os recursos da arquitetura é se escrever programas diretamente em Assembly.

Os algoritmos de PDS são um tipo específico de algoritmo numérico. Sua principal característica é o processamento por *loops* repetitivos em tempo real, transformando um ou mais fluxos de dados numéricos (VANHOOK et al., 1993). Os programas típicos no processamento digital contêm rotinas para filtros FIR, filtros com resposta impulsiva finita; para filtros IIR, filtros com resposta impulsiva infinita; para convolução, muito importante na caracterização de sistemas LIT (Linear Invariante no Tempo); e para Transformadas de Fourier Discreta direta (Algoritmo *Fast Fourier Transform*-FFT) e inversa (IFFT), que passam o sinal do domínio do tempo para o domínio da frequência e vice-versa.

Novas técnicas de PDS como filtros adaptativos, algoritmos para redução de ruído e cancelamento de realimentação acústica e processamento de sinal ordenado (tecnologia *beam-forming*) tem sido amplamente utilizadas nos AAA's digitais. A detecção de ruído requer grande poder de processamento devido a seu caráter dinâmico: toda vez que uma pessoa que porta um AAA muda de ambiente, o ruído a ser eliminado precisa ser novamente detectado. A modulação em amplitude para distinção entre sinais desejados daqueles não desejados é uma das abordagens utilizadas para redução de ruído (KATES, 2008).

A primeira etapa de processamento dos aparelhos auditivos tem como objetivo a avaliação ou caracterização do sinal captado, com a finalidade de realizar melhores aquisições, detecção de padrões e eliminação de ruídos, aumentando a inteligibilidade. Posteriormente aplicam-se os algoritmos de correção apropriados para cada paciente, ou simplesmente aplica-se um ganho para que o sinal seja mais audível. Observe que o sinal amplificado foi processado para eliminar ruídos e a amplificação ocorrerá apenas ao som de interesse.

A seguir serão descritos alguns algoritmos de processamento utilizado nos diversos tipos de AAA. A principal função do processamento digital é melhorar a qualidade do sinal

captado eliminando o ruído para amplificação e correção do mesmo. Sabe-se que o sinal recebido pelo microfone é um sinal composto oriundo de várias fontes sonoras e que muitos sinais indesejados serão amplificados caso não sejam removidos. Adicionalmente, muitas vezes o sinal captado tem interferência do sinal gerado pelo próprio AAA, o que é chamado de realimentação acústica.

- **Microfone Direcional**

Quando uma pessoa está concentrada ouvindo o som de interesse, como no caso de uma conversa, ela geralmente olha para a fonte sonora. Mesmo assim outras fontes sonoras são percebidas e podem atrapalhar a compreensão. Um dos principais problemas para pessoas com deficiência auditiva é a redução da inteligibilidade da fala em ambientes ruidosos, uma vez que a deficiência causa perda da resolução temporal e espectral no sistema auditivo do paciente. A resolução temporal é a habilidade do sistema auditivo em detectar mudanças rápidas no estímulo sonoro e é caracterizada pelo menor intervalo entre dois estímulos sonoros perceptíveis, A resolução espectral é a habilidade de distinguir sinais com diferentes frequências (MATOS, 2010). Segundo Kates (2008), a perda de *Signal Noise Ratio*[3], uma importante métrica para percepção auditiva, é estimada como sendo cerca de 4-10 dB para pessoas com alguma deficiência audutiva.

A operação do microfone direcional tem sido utilizada há bastante tempo em AAA's a fim de preservar o sinal oriundo diretamente da fonte sonora de interesse e atenuar o que seria ruído ou interferência. O microfone direcional provê um processamento espacial, no qual a separação física das fontes sonoras é utilizada para distinguir os sinais e desta maneira aumentar o SNR. Enquanto uma pessoa com audição normal tem boa compreensão com SNR tão baixo quanto -5 dB, uma pessoa com deficiência precisa no mínimo de +5 dB de SNR para ter o mesmo entendimento (LUO; ARNDT, 2002).

Existem duas maneiras de se obter a operação de microfone direcional:

1) através de um microfone direcional, que é um microfone com duas entradas para captação de áudio e que realiza a operação de microfone direcional de forma fixa, através da própria estrutura do microfone;

2) através *array* de dois ou mais microfones omnidirecionais. Este tipo de microfone responde as flutuações de pressão acústica e é igualmente sensível aos sons vindos de

---

[3] *Signal Noise Ratio* ou Relação Sinal Ruído é a relação entre o nível do sinal de interesse a ser processado e o nível do ruído, é medido em decibél. O SNR é uma especificação muito importante no conversor ADC, pois indica o quanto que aquele sinal convertido corresponde ao sinal real.

todas as direções. O *array* de microfones omnidirecionais pode ter os seus microfones posicionados para se obter o mesmo resultado de um microfone direcional (1), com a possibilidade de aplicar filtros implementados em *hardware* aos sinais adquiridos, ou pode somente captar os sons e deixar que o DSP realize a operação de microfone direcional, através de algoritmos.

Tanto em 1) como em 2), o Microfone Direcional procura aumentar a inteligibilidade e o SNR através do uso da posição da fonte sonora e da diferença temporal de aquisição dos sinais de entrada.

Na figura 5 é apresentado o diagrama de funcionamento do microfone direcional, mostrando a aquisição dos sons por duas entradas diferentes, o mic frontal e o mic traseiro, e ainda os blocos de atraso (**τ**) e de ganho (**b**).

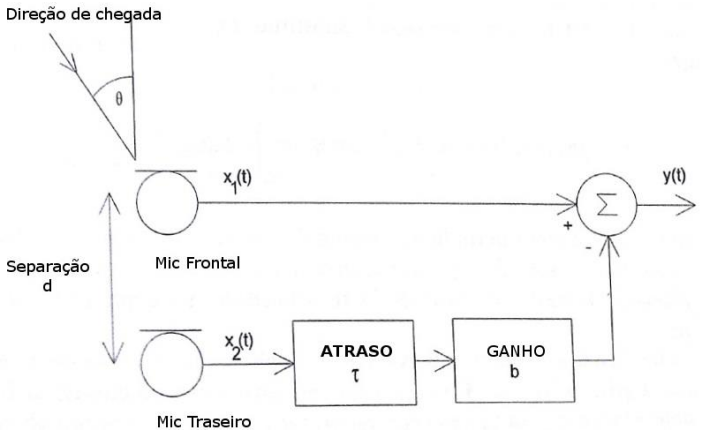

Figura 5: Microfone Direcional

A saída do microfone direcional (y(t)), um sinal com maior SNR, é uma função da distância entre os microfones (d), tipicamente entre 7 e 16 mm; do ângulo de chegada do sinal (θ) (para os sinais que a pessoa se direciona este ângulo será zero); do atraso interno **τ** inserido; e do ganho **b** do microfone traseiro. Uma importante especificação do microfone direcional é o padrão de diretividade (r), que é definido pela razão do atraso interno, o qual foi inserido por um bloco, e o atraso externo, o atraso de chegada do sinal entre os microfones devido a distancia (d) entre eles.

O tempo para que um sinal se propague do microfone frontal até o microfone traseiro é: $\delta(\theta) = \frac{d}{c}\cos(\theta)$ (1), onde c é a velocidade do som (aproximadamente 345 m/seg).

Considerando um sinal de audio externo denominado s(t), a saída do microfone frontal em resposta a este sinal é x1(t), sendo x1(t) = k*s(t) (2), onde k é uma constante que modela a

mudança de domínio do sinal, que entra como sinal sonoro e sai como um sinal elétrico. A relação entre a entrada e a saída no microfone frontal é direta. A saída do microfone traseiro é x2(t), sendo x2(t) = k*s(t - $\delta(\theta)$) (3). Agora a relação entre a entrada e a saída não é direta, pois se considera o atraso entre os dois microfones.

A equação 4 relaciona o atraso da chegada do sinal s(t) no microfone traseiro, devido a distância (d), com o atraso adicionado **τ** para produzir a saída y(t). Com isso tem-se

$y(t) =$ x1(t) – b*x2(t- τ) (4). onde *b* é ganho do microfone traseiro.

Considerando agora a frequência do sinal de chegada (ω), a partir da equação 4, (Kates, 2008) chega-se a seguinte equação : $|Y(\omega,\theta)| \approx \omega \frac{d}{c} \cos(\theta + \beta)$ (5)

O termo β é o ajuste do atraso do microfone traseiro, sendo $\tau = \beta(d/c)$.

Na figura 6 é mostrado o padrão de atenuação do som de entrada em função do ângulo de chegada para um microfone direcional (resposta da equação 5). Observe que a atenuação pode chegar em até 40 dB para os sons que não são de interesse, a 180° do observador, e que não há atenuação para os sons captados frontalmente.

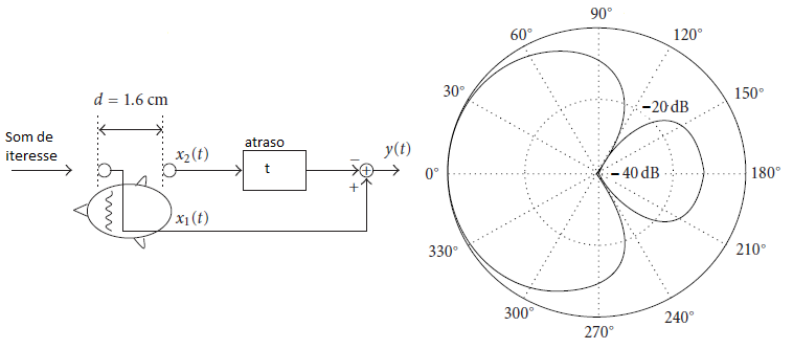

Figura 6: Resposta do microfone direcional
Fonte: Kates (2008)

Na figura 7 é mostrado um exemplo de um aparelho auditivo cujos sinais sonoros são captados por um *array* com três microfones omnidecionais que realizam a operação de microfone direcional. Na figura são descritos os diferentes fluxos dos sinais que passam por blocos para inserção dos atrasos T1 e T2, por subtratores e por filtros de compensação e de redução da banda de frequência do sinal.

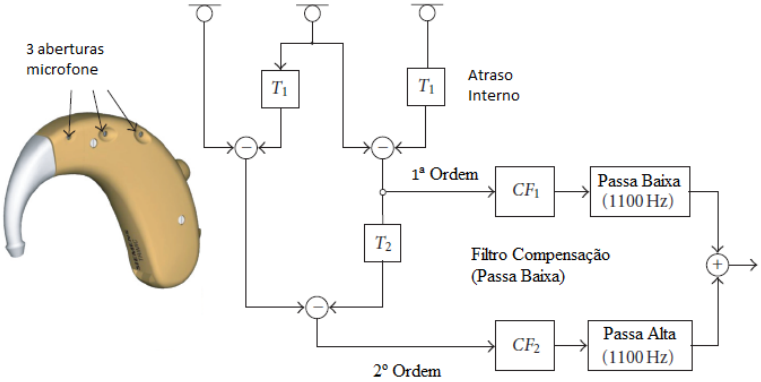

Figura 7: *Array* microfone omniderional

Existe a possibilidade de se ter *arrays* adaptativos, onde os filtros de cada microfone mudam seus coeficientes ao longo do tempo em respostas às mudanças das características espaciais das interferências.

Atualmente a maioria dos aparelhos auditivos usam apenas dois microfones omnidirecionais, devido principalmente às limitações de espaço, e deixam para realizar a operação de microfone direcional no DSP através de algoritmos. Os sinais são captados por estes dois microfones, são filtrados por um banco de filtros FIR programáveis e enviados ao DSP que, após a operação de microfone direcional, gera apenas um sinal com maior SNR. A melhoria deste sinal resultante é indiscutível quando se está em um ambiente com pouco ruído, entretanto, para ambientes muito ruidosos e que sejam fechados, causando assim grande reverberação no sinal, o ganho de SNR após a operação de microfone direcional pode ser pequeno ou mesmo imperceptível. Para aparelhos que são do estilo CIC, não há espaço para se ter dois microfones para recepção, o que impossibilita a operação de microfone direcional.

- Algoritmos de redução e supressão de ruído

A abordagem principal tanto nos algoritmos para redução quanto para supressão de ruído é primeiramente encontrar no sinal de entrada características que diferenciam o ruído do sinal de interesse e, a partir daí, aplicar filtragens baseados nestas características que reduzam ou suprimam o ruído com o mínimo efeito no sinal de interesse. Para redução, o sinal é apenas passado por filtros que atenuam o ruído, enquanto que para supressão, são aplicados algoritmos baseados na geração de sinais que serão adicionados ao sinal recebido para realizar a subtração do que é considerado ruído. São utilizados filtros adaptativos para a geração dos sinais que serão subtraídos do sinal captado.

Considerando-se principalmente o sinal da fala, os ruídos são tipicamente menos intensos e se diferenciam por características temporais e espectrais. Na figura 8 é mostrado um segmento de um sinal de fala que foi amostrada a 16 kHz. Percebe-se que o sinal tem picos chegando em 10 dB e vales de até -25 dB abaixo do valor de RMS (*Root Mean Square* ou conhecido em português como valor eficaz).

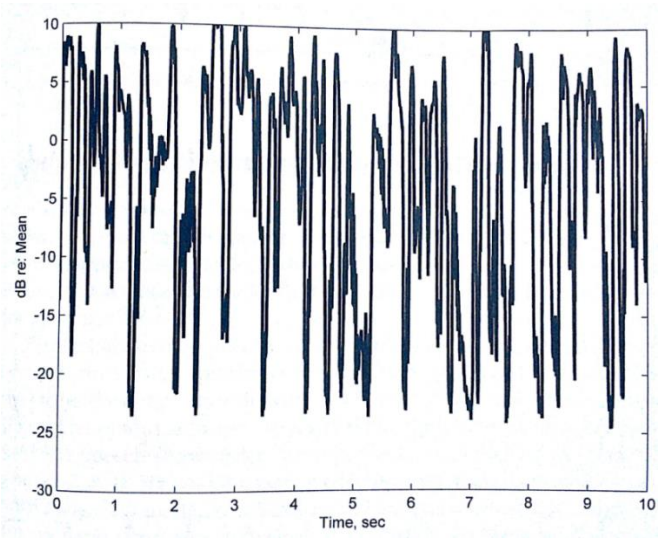

Figura 8: Sinal de um segmento de fala

Fonte: Kates (2008)

Na figura 9 é mostrado o envelope de um ruído gaussiano. Observa-se que são pequenas as flutuações do sinal comparado ao sinal da fala. O sinal gaussiano é estacionário, o que significa que não muda muito ao longo do tempo e medidas da potencia do sinal dão valores constantes.

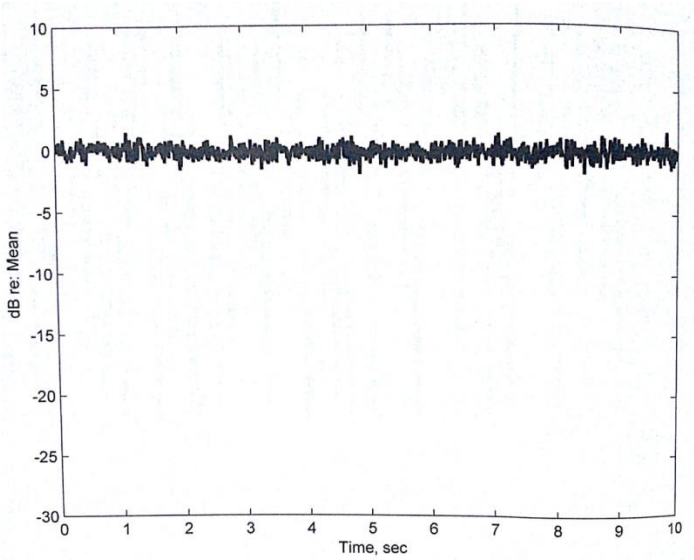

Figura 9: Envelope de um segmento de ruído gaussiano branco

Fonte: Kates (2008)

O comportamento dos principais ruídos auditivos está entre o sinal da fala e o ruído gaussiano, apresentando maiores flutuações que o sinal gaussiano, mas menores que o sinal da fala.

Além dos valores de amplitude, as frequências dos sinais recebidos também são utilizadas para diferenciar ruído do sinal de interesse. Segundo Kates (2008), as frequências dos sinais de ruído do trânsito e de balbucio estão entre as frequências do sinal da fala e gaussiano. Pode-se ainda dizer que o espectro dos sinais da fala são concentrados nas baixas frequências, enquanto que os de ruído tem normalmente seu espectro espalhado. Algumas exceções são o ruído de trânsito, também concentrado nas baixas frequências, e os ruídos estridentes, concentrados em altas frequências.

Agora que se têm uma ideia das diferenças entre os sinais da fala e de ruído, podem-se apresentar alguns algoritmos aplicados na redução e supressão de ruídos. São eles:

- ✓ Algoritmo de Rastreamento do Envelope de Vale: este algoritmo baseia-se no fato do sinal da fala ter grandes flutuações, enquanto o ruído é estacionário. Em sua implementação, tem-se detectores de pico e de vale para diferentes bandas do sinal de entrada. Os picos detectados são comparados entre si e uma razão alta entre eles indica um sinal de fala; uma razão baixa, em contrapartida, indica ruído. A redução do ruído é feita atenuando a banda onde o sinal apresenta baixa razão pico-vale. Segundo Kates (2008), resultados experimentais deste algoritmo em conjunto com o microfone direcional não mostraram melhoria na inteligibilidade do sinal em ambientes ruidosos, entretanto muitos pacientes manifestaram maior satisfação com o som processado, dizendo ouvirem um som mais compreensível.
- ✓ Filtros adaptativos passa-alta ou passa baixa: outro algoritmo muito utilizado nos aparelhos auditivos é aquele que reduz o tamanho da banda do sinal recebido, que é realizado por filtros adaptativos passa alta ou passa baixa. Caso o ruído esteja concentrado em uma banda de frequência, seja ela alta ou baixa, é aplicado um filtro que irá atenuar o sinal nesta banda. O filtro é responsável por detectar os limiares de frequência, ou seja, detectar a banda onde se encontra o ruído a ser atenuado.
- ✓ *Adaptive Line Enhancer* (ALE): é uma estrutura de filtro muito utilizada nos AAA's. Ele é um filtro adaptativo auto ajustável utilizado para redução de ruído. O ALE é capaz de separar as componentes periódicas de um sinal das componentes estocásticas, baseando-se no conceito de predição linear de um único sinal de entrada. Sua aplicação para redução de ruído é interessante devido sua simplicidade e facilidade de implementação (RAMLI, 2012).

- ✓ Algoritmos de supressão espectral: é uma família de algoritmos que realiza a subtração do que se estima ser ruído no sinal recebido. A abordagem clássica é dividir o sinal captado em bandas, realizar a FFT e identificar a magnitude do ruído em cada uma das bandas. Com a estimativa do ruído, é criado um sinal que é subtraído do sinal de entrada. A geração do sinal é feita por filtros adaptativos. O filtro mais clássico deste algoritmo é o Filtro Wiener, que assume que o sinal de entrada e o seu ruído sejam processos estocásticos lineares estacionários (ERKAN, 2009). É importante destacar que esta família de algoritmos causa distorções na amplitude do sinal captado, sendo que quanto mais supressão se aplica ao sinal maior será a distorção. Há, assim, um *trade-off* entre a quantidade de ruído a ser eliminado e a distorção introduzida no sinal.

- **Cancelamento de realimentação acústica**

Um sistema com realimentação é aquele onde parte ou mesmo todo o sinal de saída do sistema é retornado à entrada do mesmo. Tais sistemas são muito comuns em aplicações de controle, sendo às vezes mesmo desejada a realimentação. A realimentação acústica em aparelhos auditivos é um fenômeno físico que ocorre devido à captação pelo microfone do som gerado pelo próprio aparelho. Neste caso a realimentação não é desejada e tem um grande impacto no desempenho do AAA por limitar o ganho máximo que se pode aplicar ao sinal recebido além de produzir ruídos devido ao sinal de entrada indesejado. De fato, quando certo ganho é excedido, o sinal de saída do aparelho auditivo oscila, produzindo um som desagradável para o usuário, referido como assobio. A realimentação é um dos maiores problemas na implementação de um aparelho auditivo.

A maioria dos aparelhos auditivos é ventilada ou usa moldes abertos para que o usuário não tenha a sensação de estar com o ouvido tampado e sinta o efeito de oclusão. Esta sensação ocorre quando a própria voz é transmitida pelo crânio, que serve de meio para o som chegar ao canal acústico. Assim, a voz da própria pessoa soará bastante grave e terá um efeito desconfortável. O caminho entre o alto-falante e o microfone através dos dutos de ventilação é dominante para o efeito de realimentação. Ele é matematicamente modelado levando em conta o comprimento do duto e seu diâmetro.

Assim como na redução do ruído, uma das abordagens para diminuir o efeito da realimentação é reduzir o ganho do sistema nas regiões de frequência onde se espera ter a maior concentração de realimentação. O filtro *notch* é muito utilizado em aparelhos auditivos, pois é capaz de reduzir o ganho em qualquer faixa estreita de frequência selecionada. Ele

conseguirá reduzir o ‘assobio’ mas, assim como visto anteriormente, a redução do ganho pode também reduzir a inteligibilidade do som.

Outra abordagem é criar um modelo matemático do caminho de realimentação para o duto de ventilação. A partir deste modelo determinar o sinal de correção que minimiza o efeito da realimentação e subtrair do sinal captado pelo microfone esta correção. Esta abordagem utiliza filtros para a geração do sinal que será usado na subtração da entrada. Estes filtros são na maioria das vezes adaptativos.

A utilização de filtros adaptativos vem se tornando o estado da arte para eliminar o efeito da realimentação nos AAA’s (V, J *et al.*, 2006). Na figura 10 é mostrada a estrutura básica de cancelamento, sendo x(t) o sinal captado pelo aparelho auditivo, a função de transferência G(z) o modelo do caminho direto do sinal, o bloco ω^o correspondente ao caminho de realimentação e o filtro adaptativo ω(n) usado para redução da realimentação. A região pontilhada define o aparelho auditivo propriamente dito.

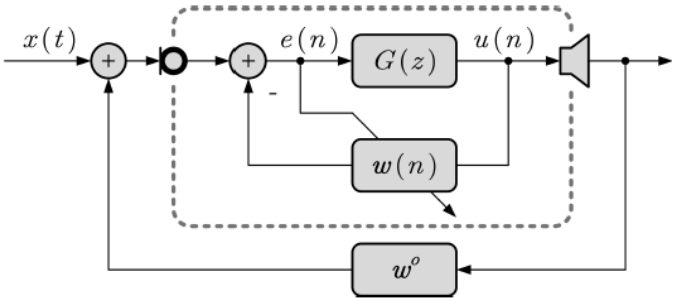

Figura 10: Aparelho AAA com realimentação acústica

A grande dificuldade na implementação do cancelamento neste caso é o levantamento dos coeficientes do filtro adaptativo, feito em tempo real e que requer grande poder de processamento. O principal algoritmo para atualização dos coeficientes é o algoritmo *LMS (Least-Mean-Square)*, sendo amplamente usado devido à sua simplicidade, eficiência computacional e bom desempenho em diversas condições de operação (Manolakis e Kogon, 2005). O cancelamento de realimentação com algoritmos baseados no método de predição de erro, conhecido como *PME (Prediction Method Error)*, tem sido fortemente estudado e é visto como solução (MOONEN; WATERSCHOOT, 2009).

O trabalho (MALUENDA, 2009) analisou o algoritmo LMS e também os algoritmos que usam a predição de erro e conclui que a escolha de um algoritmo é determinada por um ou mais dos seguintes fatores: taxa de convergência (número de iterações necessárias),

desajuste (diferença entre os parâmetros obtidos e os parâmetros ideais), rastreamento (capacidade de acompanhar as variações dos sinais), robustez (habilidade para operar frente a sinais mal condicionados), complexidade computacional, estrutura e propriedades numéricas (implementação do algoritmo num processador digital).

- Filtro Adaptativo

O filtro adaptativo é utilizado em diversas rotinas de tratamento em um AAA e está presente nos algoritmos de redução de ruído, no filtro ALE, no cancelador de realimentação etc. Segundo Diniz (2008), o desenvolvimento da tecnologia VLSI permitiu a difusão do uso de técnicas de processamento adaptativo de sinal em um grande número de aplicações. A característica principal deste filtro é que sua resposta em frequência é alterada de maneira automática, baseado em algoritmos recursivos. A alteração da resposta em frequência se deve a atualização dos valores dos coeficientes do filtro. A atualização é na verdade um ajuste dos coeficientes em busca de minimizar o erro entre o sinal de saída do filtro e o sinal desejado, que é um sinal sem ruído. Pelo fato do ruído ser desconhecido, existe a necessidade do filtro ser adaptativo, e assim se ajustar até a melhor resposta de filtragem.

Um filtro adaptativo consiste basicamente de duas partes: 1) um filtro com coeficientes ajustáveis, que pode ser do tipo FIR ou IIR; 2) um algoritmo adaptativo, que ajusta os coeficientes do filtro para melhorar seu desempenho. O algoritmo LMS é amplamente utilizado nos filtros adaptativos, como já mencionado para o caso da realimentação acústica, devido sua simplicidade (HAYKIN, 2002).

- Compressão da faixa dinâmica

Além da diminuição da inteligibilidade (diminuição do SNR), pessoas com deficiência auditiva apresentam uma redução da faixa dinâmica das frequências audíveis quando comparada a pessoas com audição normal. Segundo Kates (2008), a compressão deve ser utilizada para compensar essa redução da faixa dinâmica. Os algoritmos, basicamente, comprimem toda faixa dinâmica de frequências em uma faixa audível para o paciente. Pode-se realizar a compressão por limitação ou de forma dinâmica, em inglês *Wide Dinamic-Range Compression* (WDRC), que é a forma dominante em AAAs.

Os algoritmos de compressão são caracterizados pelo processo de análise da frequência. O sistema de compressão pode operar sobre o sinal de duas formas: sem dividi-lo em bandas de frequência, o que é o sistema mais simples de compressão; ou dividindo o sinal em bandas de frequência, canais, e aumentando a resolução no processamento com o custo de aumentar o atraso. Em Kates (2008) se afirma que a resolução de frequência e atraso de grupo

são propriedades fundamentais de um aparelho auditivo e que é necessário ter em mente o *trade-off* existente entre resolução e o atraso de processamento.

A maioria dos AAA's trabalha com compressão multicanal, sendo que os parâmetros do sistema são: o número de canais de frequência, as razões de compressão de cada banda e os tempos de ataque e liberação (*attack time* e *release time*) de cada banda de frequência.

2.1.2 Considerações do Processamento Digital

Está claro que o processamento por DSP's agrega grande valor a qualidade do sinal oriundo do meio, porém, para ser possível o processamento digital de um sinal de voz, é obrigatório à conversão deste sinal para o mundo digital, como mencionado anteriormente. É importante levantar algumas considerações sobre está passagem do sinal analógico para o mundo digital e sobre a representação digital.

A passagem do sinal analógico para o domínio digital é realizada através da amostragem que consiste pegar amostras do sinal analógico em intervalos de tempo e converte-los em sinais digitais (discretização no tempo e no valor). A taxa de amostragem deve obedecer a certas regras e a mais importante é imposta pelo teorema de Nyquist: a taxa de amostragem dever ser igual ou superior a duas vezes a frequência da componente mais rápida do sinal amostrado. Com isso se garante que a reconversão do sinal digitalizado para sinal analógico pode ser feita sem o efeito de *aliasing*. Quando se tem um sinal dito amostrado com a taxa de Nyquist isso significa que a taxa de amostragem é exatamente o dobro da frequência da componente mais rápida do sinal amostrado. O sinal discretizado é apresentado numa representação numérica. Esta representação possui regras para especificar os valores numéricos com precisão finita em uma faixa dinâmica. Estas regras são importantes para o projetista quando desenvolve o conjunto de instruções de um DSP ou as funções aritméticas executadas em *hardware*. Elas definem caraterísticas tais como se os dados estão em ponto fixo ou ponto flutuante e a resolução do dado (número de *bits* utilizado para representação, o que determina o erro de quantização). Um critério importante utilizado na escolha da resolução de dados em um DSP é o SNR desejado, uma vez que há perda de informação, ou introdução de ruído, quando se faz a representação numérica finita de valores de precisão infinita, como ocorre nos AAA's.

Os primeiros DSP's usavam representação em ponto fixo para as operações aritméticas (LAPSLEY; BIER; LEE, 1997) e, segundo Kates (2008), ainda hoje é assim na maioria dos

AAA’s. Dessa forma processadores DSP de ponto fixo são dominantes em AAA’s (LIU, 2008). A escolha por esta representação se deve principalmente a economia de consumo que ela proporciona quando comparada a representações em ponto flutuante. Para finalizar, em Liu (2008) foi constatado que não há necessidade de se ter processadores DSP trabalhando com precisão dos valores maior que a precisão do ponto fixo devido à sensibilidade do ouvido humano, proporcional ao logaritmo da intensidade sonora, e ao fato de que o som fraco, mascarado pelo som forte, não pode ser identificado pelo ser humano.

Considerando então a aritmética em ponto fixo, é necessário estabelecer a resolução das palavras. Tipicamente são utilizadas palavras com número de *bits* entre 16 e 24 para representar amostras nos diferentes AAA`s. Uma palavra de 16 *bits* permite uma representação inteira, em complemento de dois, de valores entre -32768 a 32767, que é a precisão aplicada em discos compactos de áudio. Porém para um AAA é importante considerar que as amostras serão processadas, o que significa que serão multiplicadas e somadas para produzir uma saída. Para que estas operações não causem distorções nos resultados, é necessário o aumento do número de *bits* da representação. As distorções são causadas basicamente por dois problemas: a saturação e o *underflow*. A fim de minimizar a distorção e diminuir ao máximo o erro devido a arredondamentos, os AAA’s mais modernos trabalham com 24 *bits* para o processamento, o que significa que o DSP realiza as operações com palavras de 24 *bits*.

2.1.3 Fluxo do Sinal

O fluxo de sinais, ou a sequência de operações que serão realizadas, de um aparelho auditivo digital pode, no geral, ser dividido em três etapas principais:

1. *Front end* ou codec de entrada

É a etapa responsável pela aquisição do sinal analógico e a passagem deste para o mundo digital, preparando o dado para ser enviado à etapa 2. Na figura 11 é apresentado um codec de entrada. Nela há três meios principais de aquisição: microfones, para som ambiente, o *telecoil*, para telefones, e a Entrada Direta de Áudio (DAI), necessária para ligar o aparelho auditivo a um sistema FM. Os três podem estar presentes no mesmo aparelho e o meio ativo a cada momento depende do perfil de aplicação selecionado no aparelho. Por exemplo, se o

usuário for atender um telefone ele selecionará o perfil que ativa a entrada do *telecoil*. Os sinais recebidos pelos microfones passam por Amplificadores de Ganho Programável (PGA) para aumentar o nível dos sinais com baixa intensidade e assim melhorar sua conversão para digital. Um multiplexador é utilizado para selecionar quais sinais irão paras os conversores ADC de acordo com o perfil de aplicação selecionado. O número de conversores está relacionado ao número de canais de entrada, microfones, do aparelho auditivo. A saída do ADC é uma palavra (*stream* de *bits*) modulada em PCM (*Pulse-code modulation*).

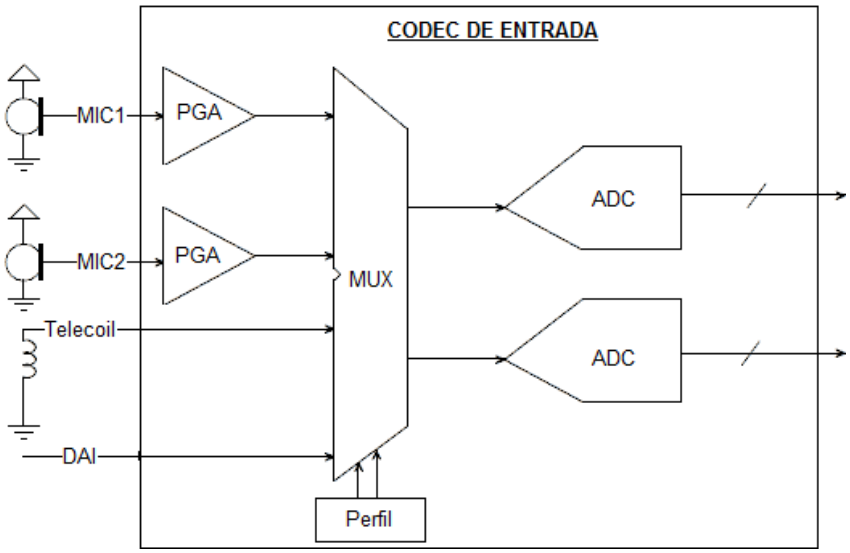

Figura 11: Codec de Entrada

Configurações Sigma-Delta (ΣΔ) são frequentemente utilizadas para implementar ADC de baixo consumo para processamento de sinal de áudio (SCHREIER; TEMES, 2004). ADC's ΣΔ com arquitetura de sobreamostragem[4] garantem alta resolução e são tolerantes as imperfeições da parte analógica do projeto do ADC (CUBAS; NAVARRO, 2012), sendo uma boa opção para estas aplicações. Um ADC ΣΔ com sobreamostragem é composto por uma parte analógica (modulador ΣΔ, com um integrador/amostrador, um quantizador e um conversor D/A) e uma parte digital (filtro decimador passa-baixa) que diminuirá a taxa de dados, muitas vezes retornando a taxa de Nyquist.

Uma importante consideração no projeto do conversor, impactando o projeto de um AAA, se refere à largura de banda do sinal considerado de interesse. Considere um modulador ΣΔ de segunda ordem com sobreamostragem igual a 128 vezes a frequência de Nyquist, para

[4] Sobreamostragem é termo usado para dizer que o processo de amostragem ocorre com uma frequência muito mais elevada que a frequência de Nyquist.

uma banda de interesse de 8,0 kHz. Seguindo o teorema de Nyquist, o modulador trabalharia com uma frequência de amostragem de 2,048 MHz (8,0*128*2), produzindo *bits* de informação nesta mesma taxa. Estes *bits* passam pelo estágio de decimação, composto por filtros decimadores digitais, para compor uma palavra a ser transmitida para unidade de processamento a uma taxa mínima de 16 kHz. Caso a banda de interesse fosse de 20,0 kHz, o modulador trabalharia com uma frequência de 5,120 MHz e a unidade de processamento receberia amostras a uma taxa mínima de 40 kHz, devendo executar todos os algoritmos de processamento nessa taxa.

Considerando o desenvolvimento de aparelhos auditivos de menor consumo e a maior simplicidade do *hardware*, faz-se a largura de banda do sinal de interesse captado restrito a 8,0 kHz, que atende aos sinais de fala e fonia (para comunicações telefônicas). A restrição de banda diminuirá assim a frequência de operação no ADC, principalmente nos filtros do seu decimador, e a taxa de transferência de dados para a etapa 2, a unidade de processamento digital.

No trabalho (Carvalho e Navarro, 2012) é analisado o projeto de um decimador em múltiplos estágios de baixo consumo, taxa de saída igual à taxa de Nyquist, para um modulador com fator de sobreamostragem igual a 128 e foi constatado que a decimação em múltiplos estágios tem o consumo menor comparado a decimação em um único estágio.

2. Unidade de Processamento Digital

É a etapa responsável por aplicar algoritmos para resolver ou minimizar o problema da deficiência auditiva do paciente, proporcionando maior conforto acústico. Todo o processamento pode ser realizado por apenas um DSP ou por um conjunto DSP-Coprocessadores. Na figura 12 é apresentado o fluxo nessa etapa.

Considerando o sinal captado por dois microfones omnidirecionais, a etapa 1 entrega duas palavras (pacote de *bits*) a uma taxa determinada pela taxa de conversão dos ADC's. Tais pacotes passarão por filtros configuráveis, tipicamente filtros passa baixas. As saídas dos filtros serão processadas pelo algoritmo de microfone direcional, discutido anteriormente, gerando então uma única saída. Esta saída será processada pelo algoritmo de cancelamento de realimentação acústica (FBC). O sinal resultante é um sinal com o mínimo de ruído, pronto para receber o processamento para correção para o tipo de deficiência. O primeiro passo na correção é o reconhecimento do tipo de fala, realizado pelo processo de análise do sinal. Após a análise e com as informações da deficiência do paciente, obtêm-se os parâmetros para cada

algoritmo que será aplicado ao sinal: um ganho diferente para uma determinada faixa de frequência, parâmetros para o algoritmo compressão WDRC, parâmetros para o algoritmo que desloca as frequências altas do sinal para baixas, caso o paciente tenha problema para ouvir os sinais de alta frequência, entre outros. O segundo passo da correção é a síntese onde os algoritmos de correção propriamente ditos são aplicados.

O sinal de saída do processamento para correção é passado por um circuito controle de ganho automático (CGA), para aumentar o ganho dos sinais com baixo nível. O sinal é então processado por um filtro ALE para redução de ruído, e por um filtro configurável, tipicamente um passa altas (PA). A saída será desta etapa será enviada para a etapa 3.

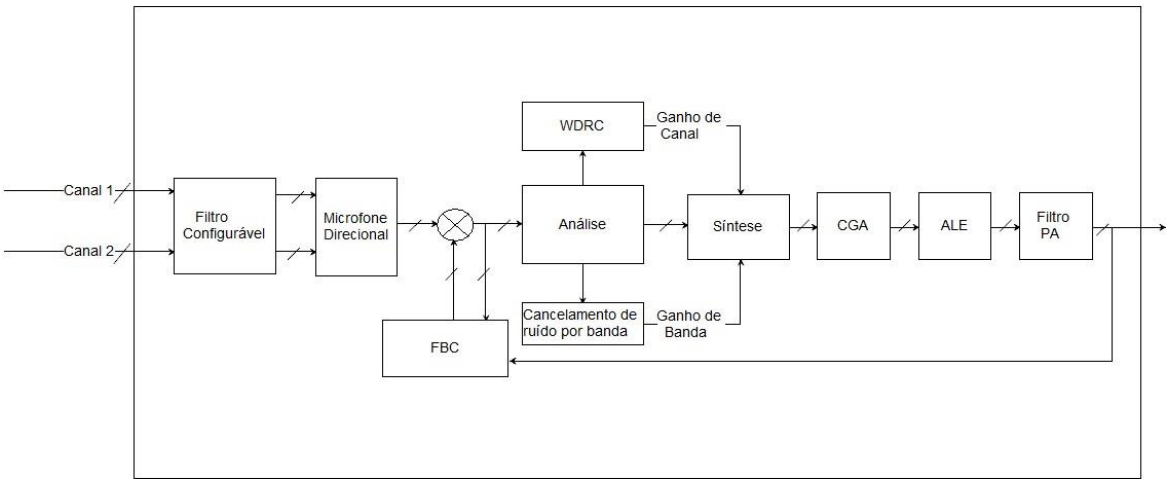

Figura 12: Fluxo de sinal para processamento digital

3. *Back end* ou Codec de saída:

É a etapa responsável por receber o sinal processado no domínio digital, passa-lo ao domínio analógico e aplica-lo ao alto-falante, para que assim se produza o sinal compreensível ao paciente. É também onde se aplica o controle de volume e onde se tem o gerador de tom. Na figura 13 é apresentado o fluxo nessa etapa.

O DSP entrega palavras ao codec de saída com a mesma taxa que recebe do ADC. A primeira operação no codec é a ação do controle de volume (CV), que irá multiplicar as palavras por um valor escolhido pelo usuário e que lhe garanta o maior conforto. Após esta operação, tem-se o bloco de compressão de saída (AGCO) o qual modula o sinal para que não haja saturação, bloqueando a saída quando a sua entrada estiver acima de um determinado limiar. Para garantir maior linearidade da conversão digital para analógico, o sinal digital

passa por um interpolador, que irá aumentar a taxa de dados, e após vai para DAC. A saída do DAC, sinal analógico, é amplificada e, por fim, aplicada ao alto-falante.

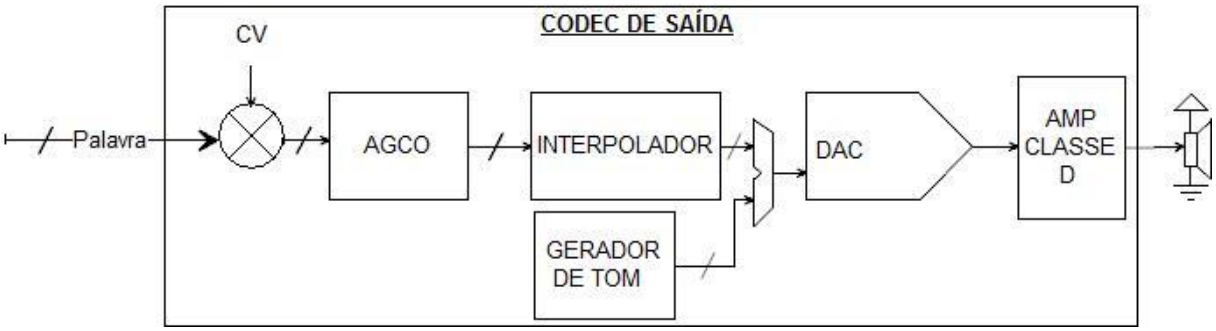

Figura 13: Codec de Saída

Para aparelhos auditivos digitais, comumente se usa um DAC ΣΔ de terceira ordem e um amplificador classe D. Amplificadores classe D tem menores consumo (alta eficiência), distorção e tamanho quando comparados com amplificadores de classes A e B (JACOBS, 2006; INTEGRATED, 2011).

2.1.4 Plataformas para desenvolvimento de AAA's Digitais

Esta seção irá apresentar três plataformas propostas para o desenvolvimento de AAA's, mostrando como elas evoluíram ao longo dos anos e passando assim uma ideia das restrições atuais em área e consumo para AAA's. Por plataforma entende-se o conjunto de blocos necessários, como DSP's, conversores ADC e DAC, gerenciador de alimentação, interfaces, etc., para que uma empresa de aparelhos auditivos possa implementar seus algoritmos e desenvolver seus produtos.

- Plataforma da Texas Instruments –TI (INSTRUMENTS, 2013)

A TI é uma empresa estado-unidense, fundada em 1930, atuante na área de semicondutores, especialmente no mercado de DSP's, microcontroladores e conversores (ADC's e DAC's). Ela lançou em 2000 uma plataforma de desenvolvimento para aparelhos auditivos digitais baseados em seu DSP programável de ponto fixo, o TMS320C5000E. Podem ser observados na figura 14 os blocos que compõem toda a plataforma, que além do DSP possui conversores e reguladores de tensão para fornecer o correto valor de tensão para os diferentes blocos.

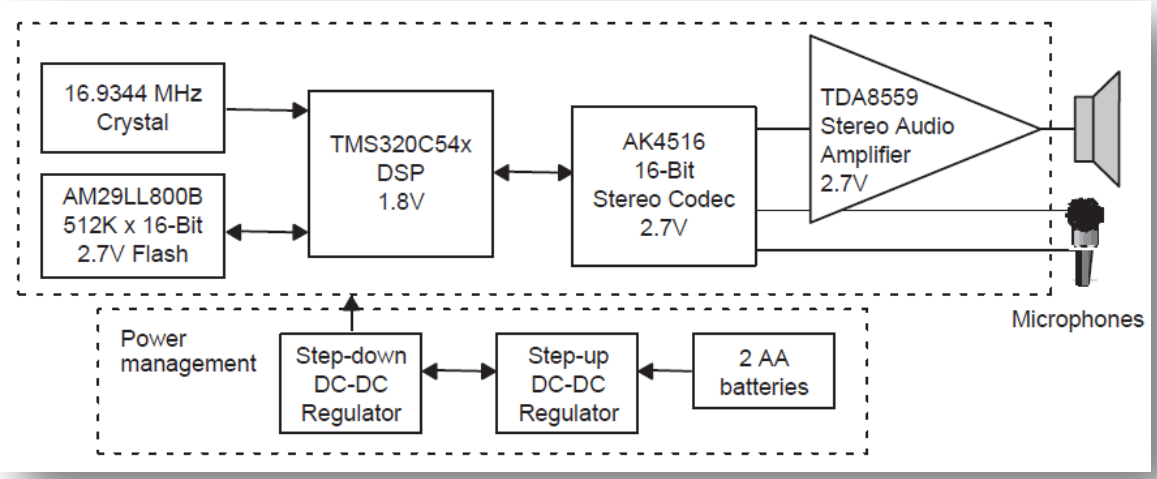

Figura 14: Plataforma TI

Esta plataforma é um sistema de tempo real capaz de processar duas entradas com banda de frequência chegando até 24 kHz (0-24 kHz) para cada canal. Seu processamento está todo concentrado em seu DSP, desta forma, limitando-o para aplicação de apenas algumas rotinas de PDS desenvolvidas na época. Os algoritmos implementados foram basicamente três:

- *Shaping* de frequência: este algoritmo separa em bandas de frequência o sinal de entrada e desta forma permite compensar a perda do paciente de forma bem precisa;
- Redução de ruído adaptativo: este algoritmo tem como objetivo aumentar a inteligibilidade do sinal. O núcleo deste algoritmo é o *Real-Time Adaptive Correlation Enhancer* (RACE) , o qual provê a redução de ruído. O RACE é essencialmente um filtro FIR adaptativo de *loop*-aberto.
- Compressão de amplitude: este algoritmo é aplicado ao sinal de saída com o objetivo de manter todo o sinal dentro de um nível sonoro que não cause incomodo ao usuário do aparelho.

O bloco AK4516 realiza a amplificação do sinal de entrada através de um PGA, a conversão análogo-digital do sinal captada através de um ADC ΣΔ e também a conversão digital-análogo do sinal processado através de um DAC ΣΔ. O ADC possui resolução de 16 *bits*, frequência de amostragem de até 48 kHz, Distorção Harmônica Total (TDH) de -85 dB e Faixa Dinâmica (DR) de 90 db. O DAC possui resolução de entrada de 16 *bits*, saída *single-*

end (não diferencial), TDH de -86 dB e DR de 90 dB. O bloco AK4516 possui ainda uma saída sinalizando *overflow* e um circuito que limita a saída automaticamente.

O bloco TDA8559 é um amplificador *stereo* de baixa tensão que pode ser conectado diretamente a um alto-falante. A plataforma também possui um bloco de memória com 512 k palavras de 16 *bits* (AM29LL800B).

A especificação da plataforma quanto ao consumo é de 232 mW para tensões de alimentação de 1,8 V para o DSP e 2,7 V para os conversores e amplificadores, o sistema operando com 100 MIPS e 44,1 kHz de frequência de amostragem. Tal consumo permite uma autonomia de 23 horas com duas baterias AA, porém é um consumo muito alto se comparado ao desejado de 1mW para os AAA's, como será visto na seção 2.3. A especificação de área, considerando o encapsulamento e todos os periféricos como micro-fone e alto-falante, é de 14,6 cm x 9,37 cm, aproximadamente 136 cm², para uma tecnologia CMOS de 0,25 µm. Claramente esta plataforma não satisfaz os requisitos de área dos AAA's, pois, tipicamente, a dimensão desejada para um CI que implementa um AAA é de 20 mm^2.

Observa-se que esta plataforma não apresentou nenhum bloco de processamento específico para auxiliar o DSP, sendo que todo o processamento deve ser realizado por *software*. Como será apontado nas plataformas seguintes, a adição de blocos que realizam processamento por *hardware*, auxiliando o DSP, foi o caminho para se alcançar maiores poder de processamento e eficiência.

- Plataforma da Gennum – GA3280 (GENNUM, 2013)

A Gennum Corporation é uma empresa de semicondutores Canadense, fundada em 1987, e que foi adquirida pela Semtech em 2012. Desenvolveu em 2006 a plataforma hibrida Voyager (GA3280) para aplicações de instrumentação auditiva, sendo esta um sistema de ultra-baixo consumo para o processamento de áudio. O diagrama detalhado desta plataforma é mostrado na figura 15. Nela é possível observar dois blocos, CI's, independentes: o GC5055 (bloco superior) que é um codec de áudio de alta fidelidade e o GC5003 (bloco inferior) que é um DSP.

Toda a plataforma GA3280 é programável, o que permite o desenvolvimento de novos algoritmos de PDS, que podem ser executados eficientemente no GC5003, e a utilização de diferentes configurações no codec GC5055.

O GC5003 é um DSP desenvolvido para processamento de áudio de forma conjunta em *software* e *hardware*, uma vez que sua arquitetura é composta por *clusters* com

processamentos específicos. Todo o GC5003 será melhor detalhado na seção 2.2.4. Adianta-se que as operações de FFT/IFFT, o algoritmo para o cancelamento de realimentação acústica e o limitador de compressão de saída são realizados diretamente por *hardware*.

O GC5055 é um codec de entrada e saída projetado especificadamente para aparelhos de auxílio auditivo. Possui parâmetros de configurações, como por exemplo, a taxa de decimação do sinal proveniente do ADC e a seleção da entrada, que permitem que o codec seja usado para diferentes aparelhos. Possui também regulador de tensão e um monitor de bateria.

O codec de entrada é composto por dois ADC's, decimador, dois blocos para remoção de sinais contínuos, dois filtros *anti-aliasing* e um bloco de *cross-fader* de entrada. O ADC possui arquitetura ΣΔ de segunda ordem, com frequência de amostragem nominal de 2,048 MHz, resolução de 20 *bits* e uma faixa dinâmica (DR) de até 95 dB. A largura da banda de frequência do sinal de entrada pode ser selecionada de 8,0 até 48 kHz, dependendo da configuração do decimador. O codec possue quatro entradas de áudio, sendo dois microfones, um *telecoil* e uma entrada DAI. Devido aos dois ADC's pode-se utilizar os dois microfones como um microfone direcional.

O bloco de *cross-fader* permite a suavização do sinal quando ocorre a mudança da fonte de entrada do aparelho, pois ele gradativamente atenua o sinal de uma entrada, enquanto, ao mesmo tempo, amplifica o sinal da outra entrada. O *cross-fader* também pode ser programado para atenuar gradualmente apenas uma entrada de áudio do seu máximo até o zero (*mute*), ou amplificar do zero até o seu máximo.

O codec de saída é composto por um DAC, um gerador de tons e pelos blocos de controle de volume, de *cross-fader* de saída e de compressão de saída (AGCO). O DAC possui arquitetura ΣΔ de terceira ordem com 88 dB de DR. O modulador do DAC recebe as amostras de áudio no formato PCM e as converte em um sinal de 1 *bit* no formato PDM (*Pulse-density modulation*) que é passado a um circuito de ponte H. A ponte H é um circuito CMOS que funciona como um *driver* de corrente, convertendo o sinal PDM para um sinal diferencial de tensão de baixa impedância, adequado para aplicações em aparelhos auditivos.

A especificação de consumo máximo para a plataforma GA3280 é de 25 mW para tensão de alimentação de 1,1 V, com o sistema operando em 42 MIPS e 2,0 MHz de frequência do *clock*. A especificação de área, considerando o encapsulamento, é de 5,46mm x 3,15mm, aproximadamente 17,2 mm², com uma tecnologia CMOS de 180 nm. A plataforma tem uma espessura de 1,65 mm e suas especificações de dimensões permitem sua aplicação em todos os estilos de AAA.

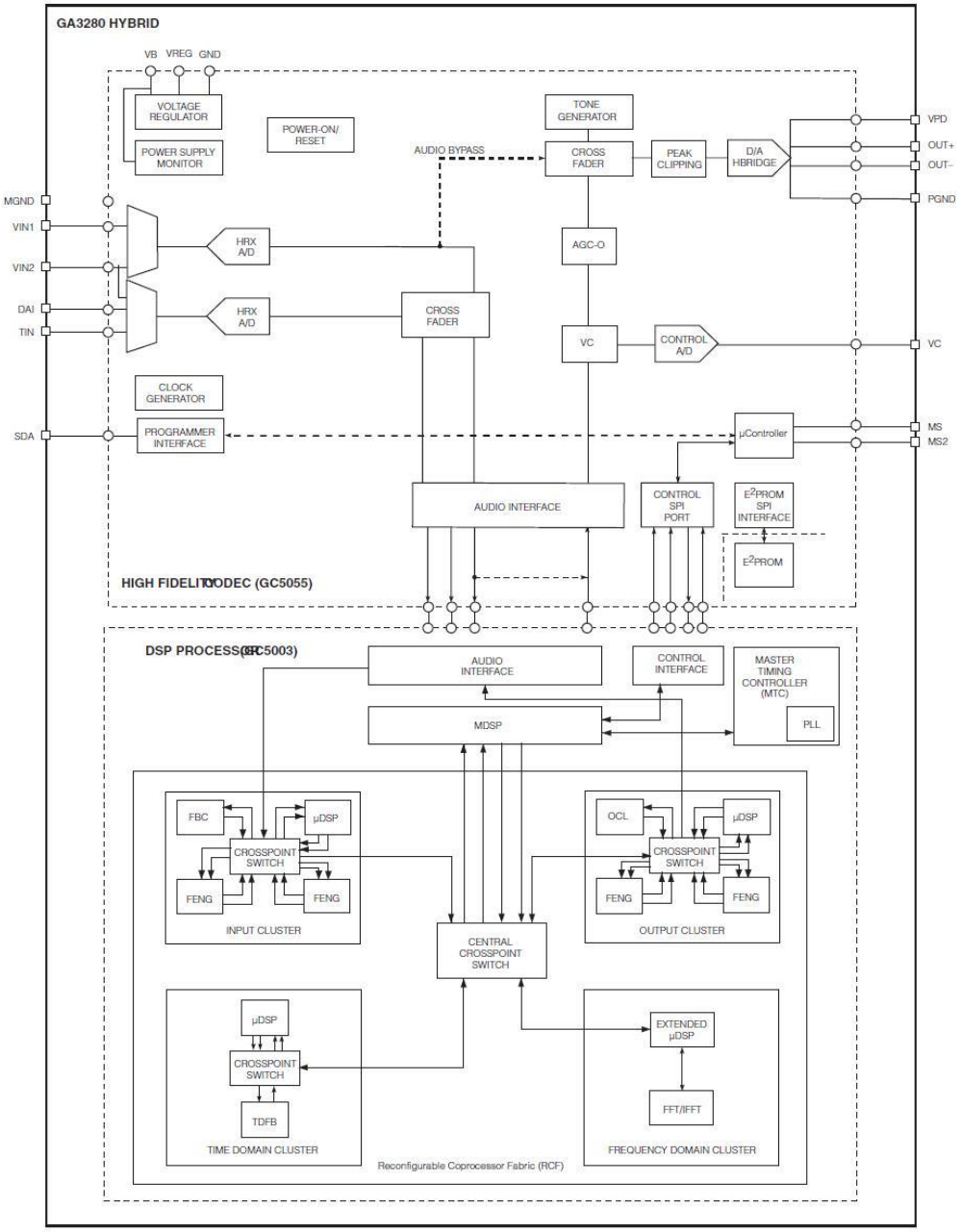

Figura 15: GE3280- Diagrama de Blocos –Plataforma da Gennum

- Plataforma da ON Semiconductor - EZAIRO 5920 (SEMICONDUCTOR, 2013)

A ON Semiconductor é uma empresa estado-unidense, fundada em 1999, atuante na área de semicondutores. Desenvolveu a plataforma EZAIRO 5920, a mais moderna plataforma para a implementação de aparelhos de auxílio auditivo, que foi lançada em 2012.

Ela é uma plataforma programável, com alta fidelidade sonora, além de ultra-baixo consumo. O diagrama detalhado desta plataforma é mostrado na figura 16.

O EZAIRO tem o processamento sendo realizado pelo DSP CFX e por um coprocessador, o HEAR, que é um Acelerador de Processamento de Sinal Configurável. O DSP utilizado é totalmente programável, DSP de propósito geral, com dados de 24 *bits*, podendo operar com frequências de *clock* entre 1,28 MHz até 5,12 MHz. Desta maneira pode-se equilibrar poder de processamento e consumo de potência. Ele será melhor detalhado na seção 2.2.4. O HEAR é um coprocessador configurável pelo DSP otimizado para processamento de áudio, principalmente para os algoritmos de microfone direcional, de cancelamento de realimentação, de redução de ruído e de compressão de faixa dinâmica.

O *codec* de entrada do EZAIRO possui quatro canais de recepção, cada um composto por um pré-amplificador com ganho de 12 até 30 dB, um ADC com arquitetura ΣΔ e resolução de 16 *bits*, e um decimador. Um canal é destinado ao *Telecoil*, outro ao DAI. Os dois canais restantes são para os microfones, os quais processam sinais com banda de frequência de 8,0 kHz até 16 kHz, mantendo a DR de 85 dB, ou podendo chegar a 110 dB quando se combinam dois ADC's para um microfone.

O *codec* de saída do EZAIRO possui um interpolador, que realiza a interpolação do sinal de saída do DSP para aumentar a taxa de sobre-amostragem, um DAC ΣΔ, cujo sinal de saída é então passado para um *driver* de saída que pode ser conectado diretamente ao alto-falante. A DR do codec de saída é de 94 dB e a THD é de −76 dB.

O EZAIRO possui ainda um controlador de interrupção, que pode ser gerada pelo DSP ou pelo HEAR, um regulador de tensão usado para os diferentes dispositivos que operam com tensões diferentes, um monitor de bateria para sinalização, uma interface GPIO (*General Purpose Input/Output)* para periféricos, e uma UART (*Universal Asynchronous Receiver/Transmitter*) para comunicação RS−232.

Sua especificação de consumo é de 2,0 mW com uma tensão de alimentação de 1,25 V, para o cenário de maior processamento, sendo este quando a frequência de *clock* do DSP está em 5,12 MHz. A especificação de área, considerando o encapsulamento, é de 5,97 mm x 3,43 mm, aproximadamente 20,5 mm², com uma tecnologia CMOS de 130 nm. Esta área permite o desenvolvimento de quaisquer estilos de AAA's.

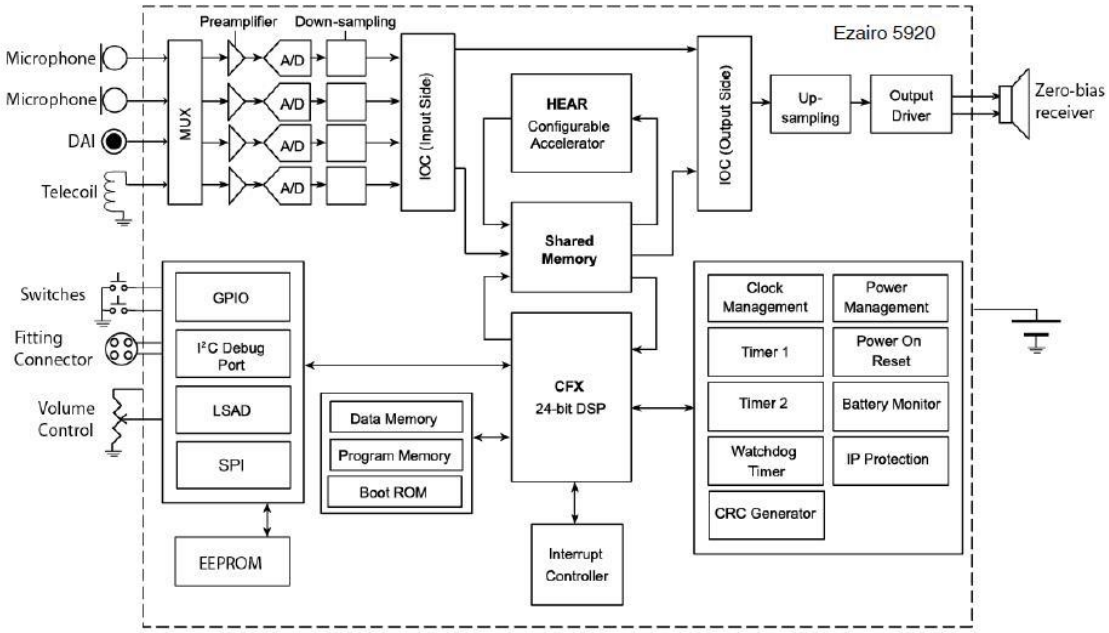

Figura 16: EZAIRO- Diagrama de Blocos –Plataforma

Na tabela 1 resumem-se as principais características de fabricação e operação das três plataformas mencionadas. Como a plataforma da TI não apresenta nenhum *hardware* mais específico de processamento, como os *clusters* do GA3280 ou o HEAR da ON Semiconductor, observa-se que ele necessita mais que o dobro do número de MIPS, acarretando um consumo maior. As três plataformas foram fabricadas com tecnologia diferentes e a da TI tem maior tensão de alimentação, o que justifica em parte seu elevado consumo.

Tabela 1: Resumo das características das plataformas descritas

Desenvolvedor	Desempenho (MIPS)	Tensão Alimentação (V)	Consumo (mW)	Tecnologia (nm)
TI	100	1,8	232	250
Gennum (GA3280)	42	1,1	25	180
ONSemiconductor (EZAIRO)	45	1,25	2	130

2.2- Digital Signal Processor – DSP

Um Processador de Sinal Digital ou *Digital Signal Processor* (DSP) é um tipo específico de microprocessador programável para aplicações de processamento de sinal, o qual possui características próprias que o torna poderoso para tais aplicações. Ele é indicado para aplicações de tempo real pela superioridade na velocidade de processamento comparada à dos microprocessadores para aplicações genéricas. Essa superioridade se dá pelo fato de que os DSP's são baseados na ideia de determinismo, que consiste no conhecimento prévio do tempo exato de execução de determinadas tarefas, evitando assim o tratamento de interrupções, comum em microprocessadores. A sua especificação é frequentemente ditada pela tarefa a ser realizada, ou seja, as arquiteturas dos processadores DSP são moldadas pelos algoritmos de PDS, e com isso são pouco flexíveis quanto à programação (LIU, 2008; INSTRUMENTS, 2011).

Segundo Frantz (2000), os DSP's são diferenciados e projetados para tratar tipos específicos de aplicação, principalmente as que requerem processando de grande quantidade de dados. O DSP nasceu com a compreensão que muitas funções matemáticas podem ser solucionadas através de operações de multiplicação e soma.

De acordo com Parhi (1999), há duas importantes características que distingue o processamento dos algoritmos de PDS dos algoritmos de propósito geral. A primeira característica é a necessidade de *throughput*[5] de tempo real, onde os dados precisam ser processados assim que são recebidos, em oposição ao caso onde primeiro eles são armazenados e depois processados em modo *batch* (segundo plano). A segunda característica é a orientação a dados (*data driven*) em oposição à orientação a controle (*control driven*), o que implica em que qualquer tarefa ou computação no DSP ocorre depois que o dado de entrada, no caso do AAA a amostra atual, estiver disponível, fazendo com que o sistema seja sincronizado pelo fluxo de dados. Uma maneira de ilustrar essa característica é observar o fluxo de dados (amostras) na etapa 2, mencionado anteriormente. O dado entrege pelo decimador vai passando de bloco em bloco de processamento, controlando assim o fluxo das operações.

---

[5] *Throughput* ou taxa de transferência é a quantidade de dados transferidos de um lugar a outro, ou a quantidade de dados processados em um determinado intervalo de tempo.

Os DSP's apresentam algumas características comuns na arquitetura e na microarquitetura que lhes garante desempenho superior. Assim os DSP's apresentam:

- Unidades dedicadas de multiplicação e soma que realizam as operações em um único ciclo (MAC). Estas unidades são de grande impacto em termos de melhora de desempenho, uma vez que os algoritmos de DSP são principalmente do tipo multiplicação e soma. Os DSP's atuais contam normalmente com dois MAC's e são conhecidos como Dual MAC. Além dos MAC's, DSP's de alto desempenho possuem mais que um multiplicador em seu *datapath*, permitindo assim várias multiplicações por ciclo de instrução;
- Microarquitetura de múltiplo acesso à memória, que possibilita o acesso completo a memória *on-chip* em um único ciclo, além do uso de bancos de memórias separadas e independentes em termos de barramento. As memórias em processadores DSP são normalmente do tipo *zero-wait state*, significando que o processador não precisa esperar para realizar o acesso à memória;
- Modos especializados de endereçamento, tais como: endereçamento circular e endereçamento do tipo *bit-reversed*, usado na transformada rápida de Fourier (FFT);
- Instruções especiais de controle de execução, como por exemplo, o *"zero-overhead loop*". Esta instrução possibilita a execução de *loops* ou instruções de repetição sem demandar ciclos de *clock* para atualizar e testar os contadores de *loop*;
- Conjunto de instruções especializado (ISA), fazendo máximo uso do DSP e explorando o paralelismo. Duas classes de microarquitetura que executam múltiplas instruções em paralelo são a VLIW *(Very Long Instruction Word)* e Superscalar e serão detalhadas mais a seguir.

A TI foi uma das pioneiras no desenvolvimento de processadores DSP. O TMS32010, apresentado em 1983, foi o seu primeiro DSP, que já incorporou um *hardware* especializado para prover multiplicação em um único ciclo de *clock*. Atualmente os DSP's são unidades chaves que possibilitam o processamento e desenvolvimento tecnológico de muitos produtos eletrônicos.

São diversas as áreas, onde se realiza processamento de dados de forma intensiva (ANGOLETTA, 2008), que tem o DSP como unidade de processamento: sistema de comunicação, multimídia, automotivo, aparelhos médicos e até mesmo aplicações militares.

Um sistema que requeira processamento digital de sinais pode ser implementado de três formas diferentes, dependendo dos objetivos, das restrições, dos algoritmos, e da relação entre consumo, flexibilidade do sistema e poder de processamento. Fatores como a necessidade de rápido desenvolvimento do sistema para rápido lançamento do produto no mercado podem também colaborar nas decisões de sua implementação. As formas de implementação de tal sistema são através de:

- processadores de Propósito-Geral ou G*eneral-Purpose Processor* (GPP);
- processadores que possuem um conjunto de Instruções Específico para Aplicação ou *Application-Specific Instruction Set Processor* (ASIP);
- um Circuito Integrado de Aplicação Específica ou *Application-Specific Integrated Circuit* (ASIC).

Na figura 17 nos é mostrado o mercado global de aplicações de PDS e a participação nele de cada uma das formas de implementação. Pela figura observa-se que a aplicação de DSP's implementados como ASIP vem crescendo continuamente. As características destas três formas (GPP, ASIP e ASIC) serão detalhadas na próxima seção.

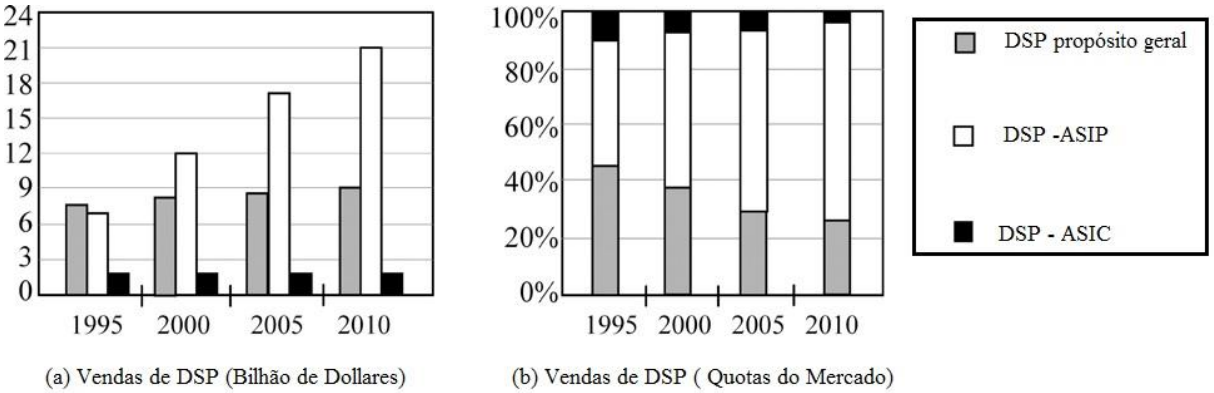

Figura 17: Números de DSP's vendidos versus no ano

Fonte: Liu (2008)

2.2.1 GPP, ASIP e ASIC

Um Processador de Propósito-Geral, GPP, pode ser utilizado na implementação de um sistema que requeira processamento digital de sinais. O termo "Propósito-Geral" implica que o processador não é designado para uma atividade ou classe de atividades específicas. Pode-se citar como exemplos do uso de GPP o processamento utilizado em mercados de ações e nas previsões de tempo ou de terremotos. Estes tipos de análises são frequentemente implementadas em *software,* as quais são executados em supercomputadores que não

possuem restrição de consumo, podendo assim operar com frequência de *clock* na ordem de Giga-Hertz. Um GPP visa um elevado desempenho para quaisquer aplicações em vez de maximizar o desempenho de uma única aplicação específica.

Um Processador que possui um conjunto de Instruções Específico para Aplicação, ASIP, é um processador designado para uma classe de aplicações restrita, cuja especialização do ISA trará maior poder de processamento com menor consumo de potência. O ASIP pode ser visto como um processador programável que apresenta certo nível de flexibilidade, o que lhe permite executar certos algoritmos dentro de uma classe para a qual ele foi designado. Além do conjunto de instruções dedicado, um ASIP também possui tipos de dados dedicados. A principal diferença entre um GPP e um ASIP é o domínio de aplicação: GPP's não são para uma classe de aplicações específica, podendo ser usado para qualquer aplicação, enquanto o ASIP é bem específico.

Um Circuito Integrado de Aplicação Específica, ASIC, é um CI totalmente customizado para uma determinada aplicação. Existem dois casos que ASIC's são necessários para o processamento digital de sinais. O primeiro é quando se tem um requerimento de processamento extremo, isto é, altíssimo desempenho, sendo que neste caso processadores programáveis GPP ou ASIP não atenderiam a carga de processamento. O segundo é quando se tem requerimentos de ultra baixo consumo ou ultra baixa área de ocupação. Aparelhos de auxílio auditivo são exemplos típicos para aplicação de ASIC's devido ao alto fluxo de dados a ser processado em tempo real e as restrições de consumo e área (LIU, 2008). Na implementação de um ASIC é feito o mapeamento direto de algoritmos para um determinado CI através de uma linguagem de descrição de *hardware*. Devido a este mapeamento direto, a implementação tem pouca flexibilidade, sendo possíveis apenas alterações de parâmetros e acomodando de algoritmos de similares de processamento.

Na figura 18 é mostrado o consumo (considerando uma tecnologia de fabricação de 90 nm) em função do MOPS (Milhão de Operações Por Segundo) para as diferentes abordagens de implementação de sistemas que requeiram processamento digital de sinais. A utilização de GPP é típica em aplicações com não mais de mil MOPS. ASIC's para PDS pode suportar aplicações com consumo muito baixo (menos que 0,01 mW/MHz) e alto desempenho.

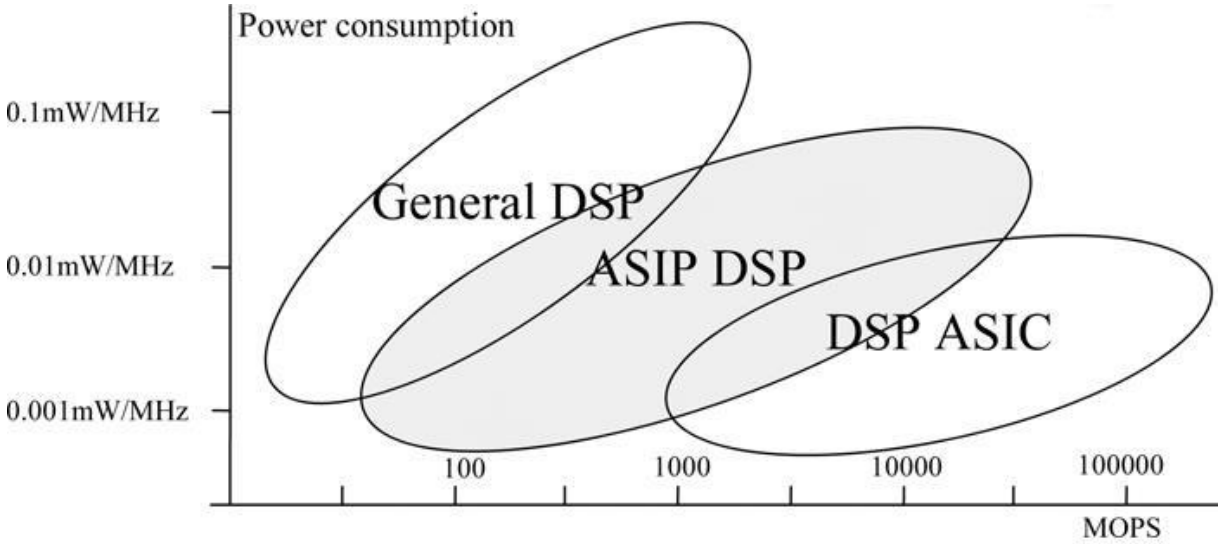

Figura 18: Potência consumida versus capacidade de processamento para as três diferentes formas de implementação de sistemas para processamento de sinais
Fonte: Liu (2008)

2.2.2 DSP – Arquiteturas

Um DSP consiste basicamente, como qualquer processador, de um núcleo de processamento onde são realizadas as operações, mencionado como *core* em inglês; de memória volátil para a execução das operações; e de periféricos para realizar comunicações entre o DSP e o mundo exterior. O núcleo de processamento pode ser dividido de forma simples em *datapath* e *controlpath.* O *datapath* consiste de circuitos de transformação de dados, que são as unidades de processamento (Unidade Lógica Aritmética – ULA, MAC, Deslocadores, Multiplicadores) e de armazenamento temporário destes dados, registradores (Vahid e Givargis, 1999). O *controlpath* consiste de uma máquina de estado finita programável ou controlador de fluxo de programa, de memória de programa e de decodificador de instrução. De forma simples, a execução de um algoritmo de PDS é controlada pelo *controlpath* que decodifica as instruções, realiza os acessos à memória e passa ao *datapath* os operandos e as operações a serem realizadas em unidades específicas de processamento.

As arquiteturas de DSP's foram desenvolvidas baseando-se principalmente em arquiteturas RISC (*Reduced Instruction Set Computing)*, porém com algumas características de arquiteturas CISC (*Complex Instruction Set Computing*), como diferentes modos de endereçamento e operações de memória, normalmente para melhora do desempenho nas

computações interativas, por exemplo, a convolução que chega a ser 10% das operações em um DSP (LIU, 2008).

RISC é uma classe de arquiteturas de processadores que possui um conjunto de instruções simples e de tamanho único e que normalmente executam as instruções com a mesma quantidade de ciclos de *clock*. CISC é outra classe outra de arquiteturas que possui um conjunto de instruções complexas e de tamanhos variados, tornando o processador mais versátil.

Segundo Liu (2008), a arquitetura de um processador é a organização de seu núcleo e seus periféricos, incluindo memória. Tal organização pode ser dividida em duas partes: a definição do seu ISA e a especificação de sua Microarquitetura, que é a organização do *hardware*. Para um determinado ISA, pode haver diferentes microarquiteturas que o implementa.

O desempenho, o custo e o consumo de potência são parâmetros chaves no projeto da microarquitetura. Do ponto de vista do desempenho, é possível um grande incremento pela paralelização em nível de instrução, ou em inglês *Instruction Level Parallelization* (ILP), na execução. Sistemas computacionais são considerados com ILP quando a execução de seus programas é realizada com multiplicas operações (como soma, multiplicação e *loads*) executando simultaneamente, mesmo se apenas um único dado de execução estiver presente no sistema (FISHER; FARABOSCHI; YOUNG, 2005). Segundo Patterson e Hennessy (2007), o primeiro processador com *pipeline* que explorava o ILP apareceu em 1960 e por volta de 1985, a maioria dos processadores de alto desempenho já utilizava ILP para sobrepor a execução de instruções e melhorar o desempenho.

Uma importante característica das aplicações de DSP é que seus algoritmos tem o fluxo de programa bem conhecido antes de serem executados, uma vez os processamentos são realizados principalmente por *loops* repetitivos. Esta característica possibilita a paralelização na busca e execução de múltiplas instruções por ciclo de *clock*. Uma preocupação quando se tem execução em paralelo é sobre as dependências, sendo que existem dois tipos de dependência que precisam ser tratadas: a dependência de dados, onde uma instrução precisa de dados gerados por outra instrução, e a dependência de recursos, quando se tem mais de uma operação necessitando utilizar a mesma unidade de processamento, i.e. duas instruções de multiplicação e apenas um multiplicador. Duas microarquiteturas utilizadas em DSP’s que permitem explorar o paralelismo na execução dos algoritmos (ILP) são a Superscalar e o VLIW.

- Superscalar, é uma microarquitetura que suporta processamento paralelo e pode operar programas com instruções irregulares. Executa múltiplas instruções em paralelo, realizando o reconhecimento e tratamento das dependências de dados através de unidades exclusivas para tais tarefas, em diversas unidades de execução para evitar a dependência de recursos (LIU, 2008). Por tratar os problemas de dependência através do *hardware*, tem grande complexidade ao nível de circuitos, o que impacta nas área e consumo dos DSP's. Por outro lado, com esta microarquitetura pode-se reduzir a complexidade ao nível do *software*, o que reflete na simplificação de compiladores e na possibilidade da reutilização de programas em DSP's que tem diferentes microarquiteturas mas mesmo ISA.

- VLIW, é uma microarquitetura que, juntamente com um compilador avançado, é uma alternativa para a microarquitetura Superscalar. O compilador irá procurar e tratar as dependências antes da execução dos algoritmos. Quando o compilador detecta dependências ele troca a ordem das instruções ou insere instruções *NOP (No Operation),* o que acarreta a diminuição do *throughput* do DSP. Uma instrução VLIW contém múltiplas instruções menores e é executada em ciclo de *clock.* Assim que a instrução VLIW é buscada da memória, todas as instruções contidas nela são decodificadas e executadas em múltiplas unidades sem qualquer verificação de dependência.

Na tabela 2 são mostradas resumidamente as principais características das duas microarquiteturas.

Tabela 2: Comparação entre arquiteturas Superscalar e VLIW

Microarquitetura	**Superscalar**	**VLIW**
Custo (Silício)	Alto	Baixo
Consumo potência	Médio	Baixo
Eficiência de Código	Alta (toda instrução é útil)	Baixa (devido aos NOP's)
Execução Paralela	Implícita Não perceptível	Explícita Notada pelo compilador
Dependência de Dados	Gerenciada por *Hardware*	Gerenciada pelo Compilador
Dependência de Recursos	Gerenciada por *Hardware*	Gerenciada pelo Compilador
Execução Fora de Ordem	Gerenciada por *Hardware*	*Schedule* pelo Compilador

Programa baixo nível	Fácil de Programar	Difícil de Programar
Compatibilidade (Código Binário)	Compatível para diferentes microarquiteturas	Não Compatível

Em geral, quando se tem um DSP que terá alto volume de produção, baixos custo financeiro e consumo de potencia, microarquiteturas VLIW são mais recomendadas. Quando há a exigência de compatibilidade entre microarquiteturas para reuso de programas ou a necessidade de programação em baixo nível, microarquiteturas Superscalar são recomendadas (LIU, 2008).

2.2.3 Memória

Para a correta manipulação dos sinais, um sistema com DSP possui memórias voláteis (RAM - *Random Access Memory)* e não volátil (EEPROM - *Electrically-Erasable Programmable Read-Only Memory*). A memória RAM é para o armazenamento dos dados processados e também para o rápido acesso das instruções, que são carregadas da memória EEPROM, onde ficam armazenados os programas. A memória RAM é interna ao DSP, sendo que o núcleo de processamento tem acesso direto a ela, enquanto que a memória EEPROM é externa ao DSP. Normalmente existe um circuito responsável por carregar as instruções e parâmetros da memória EEPROM para a memória RAM, conhecido como *bootloader*.

Um dos primeiros computadores programáveis apresentava uma organização proposta por Jonh Von Neumann, onde havia uma única memória, para armazenamento tanto de dados como de programas, e um único barramento de acesso a essa memória. Microarquiteturas com essa característica recebem o nome de arquiteturas Von Neumann. Tal microarquitetura não é indicada para sistemas que requeiram alta velocidade de processamento e grande fluxo de dados, pois se gasta muito tempo em acessos a memória por falta de paralelismo.

A solução para a exigência de maior processamento e grande fluxo de dados foi a arquitetura Harvard, que possue memórias e barramentos de acesso separados para dados e programas, possibilitando a leitura/escrita de instruções e dados concorrentemente. Recebeu este nome, pois foi desenvolvida na Universidade de Harvard.

Os modernos DSP’s apresentam microarquiteturas Harvard modificada, que consiste em duas memórias para armazenamento de dados, comumente chamadas de XMEN e YMEN, e uma memória para armazenamento temporário de programa, PMEM. Com esta

microarquitetura explora-se ainda mais o paralelismo, sendo possível carregar dois dados e uma instrução por ciclo de leitura/escrita.

Em sistemas embarcados modernos que utilizam DSP, a área e consumo de potência do subsistema de memória é mais de 10 vezes maior que do *datapath*, fazendo da memória um componente crucial nos projetos. A constatação da crescente demanda por acesso à memória nos sistemas de processamento digital é importante por ser esta demanda mais um parâmetro que impacta na escolha do sistema de processamento (*software vs hardware)*. A decisão por um processamento por *hardware* pode ser feita para reduzir o acesso e tamanhos das memórias (volátil e não volátil), porque algoritmos colocados em *hardware* não precisam acessar ou ocupar memórias.

2.2.4 DSP – Estudos de Caso

Foram pesquisados alguns DSP's comercias e esta seção descreverá três deles. Esses DSP's são utilizados em diversos produtos com severas restrições de projeto, incluindo aparelhos auditivos, razão pela qual foram escolhidos para análise.

Os três desenvolvedores dos DSP's são:

- ✓ CSEM, Centro Suíço de Eletrônica e Microtecnologia, fundada em 1984;
- ✓ NXP, empresa de semicondutores da Holanda, fundada em 1975 pela Philips com o nome de Philips Semiconductors. Foi vendida em 2006 e passou a se chamar NXP Semiconductor;
- ✓ Gennum Corporation, já mencionada na seção 2.1.4.

Características comuns aos três DSP's são a opção de aritmética com ponto fixo, memória Dual Harvard e o núcleo de processamento com arquitetura carregar-salvar (em inglês *load-store*). Neste tipo de arquitetura, as instruções trabalham apenas com dados armazenados em registradores internos e os resultados são armazenados nestes mesmos registradores. Através de uma instrução *load* dados podem ser transferidos das memórias para os registradores; através de uma instrução *store* dados podem ser transferidos dos registradores para as memórias.

Com a arquitetura carregar-salvar, a maior parte dos DSP's possuem três estágios básicos de *pipeline* para execução de qualquer instrução:

Os estágios consistem em:

- estágio de leitura (*read ou fetch*) que realiza a leitura do *opcode* da memória de programa;
- estágio de decodificação (*decode*) que realiza a decodificação do *opcode*. Também realiza a leitura dos dados da memória com informações de endereço presentes na instrução;.
- estágio de execução (*execute*) que realiza a operação definida pela instrução. É feito também a escrita de volta na memória ou em registrador.

Estes estágios são comuns aos DSP's, porém a quantidade de ciclos de *clock* necessária para a completa execução de uma instrução é diferente para cada caso.

1. CSEM – Icyflex4(CSEM, 2008)

O Icyflex4 é um núcleo de processamento RISC que realiza operações com dados de 16 ou 32 *bits*, escalável, que pode ser utilizado tanto como um microprocessador programável de propósito geral quanto como um DSP embarcado em um SoC. Seu núcleo foi projetado com otimizações para aplicações orientadas a controle ou aplicações orientadas a dados, como são as aplicações típicas de DSP's. Sendo disponibilizado como VHDL (*soft block),* pode ser sintetizado com algumas customizações definidas para a aplicação. Apresenta um ISA adequado para compilação de programas em linguagem de alto nível, como C, oferendo a possibilidade de desenvolvimento de programas com esta linguagem.

As características chaves deste DSP são:

- Memória Harvard com dois barramentos de dados (XMEM, YMEM) um barramento para programas (PMEM);
- *Pipeline* para transferências à memória, com dois ciclos para acessar tanto a memória de programa (buscar instrução) quanto a memória de dados (carregar ou salvar dados);
- Dual MAC de 32 *bits*;
- *Datapath* com dezesseis multiplicadores e dezesseis ALU's de 16-*bits* que suportam operações com operandos de várias dimensões. Os multiplicadores suportam dezesseis multiplicações simultâneas para operandos com 16 *bits*, oito multiplicações para um operando com 16 *bits* e outro com 32 *bits* e quatro

multiplicações para operandos com 32 *bits*. As ALU's suportam dezesseis operações simultâneas para operandos com 16 *bits*, oito para operandos com 32 *bits* e quatro para operandos com 64 *bits*;

- Pacote de instruções com 64-*bits* (VLIW) contendo uma, duas ou três operações que podem ser executadas em paralelo ou em sequência;

As principais características deste núcleo de DSP que o tornam versátil e com bom rendimento são:

- Largura do barramento de dados da memória configurável, podendo ser de 64-*bits* até 512-*bits* (definido na pré-síntese do VHDL);
- Compressão da instrução NOP (*NO Operation*). Até duas instruções de NOP podem ser codificadas para serem inseridas entre operações em um pacote de instrução;
- Quatro modos de endereçamento por índice reconfiguráveis em tempo de execução para algoritmos de PDS;
- Quatro instruções de processamento de dados (especifica para aplicação) reconfiguráveis em tempo de execução;
- Uma unidade de processamento de dados ou *Data Processing Unit* (DPU), implementada como coprocessador para operações muito especializadas em aplicações especificas;
- Número escalável (de 1 a 8) de Unidade de Processamento Vetorial ou *Vector Processing Unit* (VPU), para tratar o processamento de vetor de dados (definido na pré-síntese do VHDL);
- Uma unidade de movimentação de dados ou *Data Move Unit* (DMU), a qual transmite dados para a memória ou para unidades de processamento, como VPU ou DPU.

Como apresentado, este núcleo de DSP é escalável e configurável antes de ser sintetizado para o *layout* físico. Devido às duas unidades VPU e DMU, o núcleo pode prover endereçamento e instruções especializadas que são executas por unidades de *hardware* dedicadas, o que garante menor consumo total do DSP.

Na tabela 3 são apresentados os valores de área e consumo, para duas diferentes frequências de operação e tensão de alimentação, deste núcleo com duas unidades de VPU. Os circuitos foram sintetizados a partir da descrição em VHDL na tecnologia de 65nm da *foundry* TSMC (Taiwan Semiconductor Manufacturing Company).

Tabela 3: Valores de desempenho do núcleo Icyflex4 sintetizado com duas unidades de VPU

Tecnologia	**TSMC 65nm**	**TSMC 65nm**
Tensão de alimentação (V)	0,9	1,0
Frequência de referência (MHz)	10	100
Área (mm^2)	0,54	0,59
Consumo de potência (µW/MHz)	36-125	44-163

Fonte: Nagel et al. (2009)

2. NXP – Coolflux (NXP, 2009)

O Coolflux é um núcleo de processamento RISC que realiza operações com dados de 24 *bits*, de ultra baixo consumo. É capaz de operar dois MAC's, realizar duas operações de memória e atualizar dois ponteiros por ciclo de *clock.* Dessa forma ele tem alta eficiência por ciclo, sendo indicado para aplicações de computação intensiva. Devido ao seu ISA e o desenvolvimento de um compilador C próprio (Chess), o Coolflux é eficiente na compilação de códigos escritos em C. Seu núcleo é disponibilizado como *soft block* (Verilog e VHDL), sendo que na composição do DSP (núcleo + memória + periféricos), pode haver até 15 núcleos de processamento.

As características chaves do Coolflux são:

- Memória Harvard com dois barramentos de dados (XMEM, YMEM) de 24 *bits* e um barramento para programas (PMEM);
- Uma unidade de acesso direto a memória ou *Direct Memory Access* (DMA);
- Dual MAC de 24 *bits*;
- Instruções e memória de programas (PMEM) de 32 *bits*
- *Datapath* com dois multiplicadores de 24x24 *bits*, três ALU's e quatro acumuladores de 56-*bits*;
- Quatro registradores, sendo dois de 24 *bits* e dois de 56 *bits*;
- Duas unidades de saturação e arredondamento;
- Suporte para gerenciamento de consumo de energia através de instruções *stop* e *restart*.

Diferente do Icyflex4, o Coolflux é um núcleo pouco escalável, sendo possível apenas definir o número de núcleos no sistema. Ele não possui reconfigurações em tempo de execução e não oferece as opções de novas instruções ou novos modos de endereçamento.

A plataforma EZAIRO, mostrada anteriormente, utilizou o DSP Coolflux como opção para o processamento dos algoritmos de PDS. Para atingir as exigências de processamento de um aparelho auditivo, o DSP foi usado em conjunto com um coprocessador que executa as rotinas de PDS típicas nos aparelhos auditivos, e assim garantir o desempenho com um consumo razoável. Além da ON Semiconductor, a Pronak, uma empresa suíça que desenvolve aparelhos auditivos desde 1999, utiliza o DSP Coolflux em seus diversos modelos (PRONAK, 2013).

Na tabela 4 são apresentados o número de portas lógicas e valores de consumo para três diferentes frequências de operação deste núcleo sintetizado em três tecnologias da própria NXP. A tensão de alimentação foi nos três casos de 1,2 V.

Tabela 4: Valores típicos de consumo do *layout* do núcleo Coolflux

Tecnologia (nm)	**65**	**90**	**130**
Frequência de referência (MHz)	300	245	200
Número de Gates	48k	48k	48k
Consumo de potência (µW/MHz)	45	60	80

Fonte – Nxp (2013)

3.Gennum- GC5003 (GENNUM, 2007)

O GC5003 é um ASIC reconfigurável desenvolvido para aplicações de áudio, sendo composto por um DSP principal, chamado de µDSP, e quatro diferentes *clusters.* Para o GC5003, *cluster* é um aglomerado de blocos de processamento em *hardware*, os coprocessadores (Copro), como mostrado na figura 19, onde cada coprocessador tem um uma funcionalidade específica atribuindo, assim, uma funcionalidade ao *cluster*.

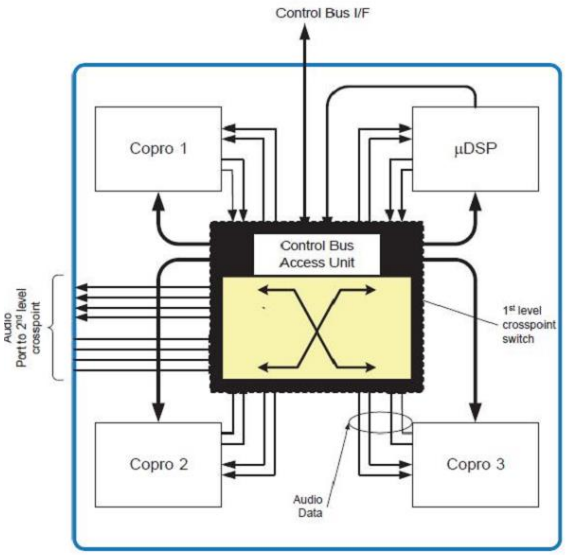

Figura 19: *Cluster* genérico do GC5003

Os quatro *clusters* formam um arranjo definido como *Reconfigurable Coprocessor Fabric (RCF)*, permitindo flexibilidade de processamento mesmo com o predomínio do processamento por *hardware*. O µDSP é um DSP programável de propósito geral e é responsável pelo controle geral, determinando os códigos que serão executados em cada *cluster* bem como o controle do fluxo de dados. Ele tem acesso às memórias e às interfaces com o mundo externo. Possui uma arquitetura RISC de 20 *bits* com apenas uma unidade de MAC e não possui um compilador para linguagem em alto nível, sendo que a opção de programação é o Assembler. O sistema não é disponível como *soft block* e também não é escalável.

As características chaves do µDSP são:

- Memória Harvard com dois barramentos de dados (XMEM, YMEM) e um barramento para programas (PMEM);
- Uma unidade de acesso direto a memória (DMA);
- *Datapath* de 20-*bits* com a possibilidade de concatenação para formar palavras de 40-*bits*;
- um Multiplicador 20x20 *bits* (único ciclo), uma ULA e um acumulador de 48 *bits*;
- Um circuito de deslocamento (*barrel shifter*) de 48 *bits*;
- Instruções de 20-*bits*;
- Instrução para realizar Log2(x) e 2^x em um único ciclo.

O RCF, como já mencionado, é composto por quatro *clusters*, definidos como *cluster* de entrada, *cluster* de saída, *cluster* domínio-tempo e *cluster* domínio-frequência. Os *clusters* são conectados uns aos outros por seletores de conexões (*crossover switches*), que controlam a transferência de dados entre os *clusters*. Como mostrado na Figura 19, existe também dentro do *cluster* o seletor de conexão para a transferência de dados entre as unidades de processamento internas.

Cada *cluster* possue um µDSP (núcleo), com uma arquitetura dual MAC que realiza operações com dados de 20 *bits*, e unidades especificas de processamento, típicas para AAA's, descritas a seguir:

- Cluster de Entrada: possui dois filtros (*Filter Engine*), que podem ser programados para filtragem IIR de biquadradas de até oitava ordem ou filtragem FIR de até 40 coeficientes, e um coprocessador que realiza o cancelamento de realimentação acústica (*Feedback Canceller* - FBC). Existe um caminho dedicado para dados entre os *clusters* de entrada e saída para facilitar a implementação do FBC;
- *Cluster* de Saída: possui dois filtros (*Filter Engin*e), que podem ser programados para filtragem IIR de biquadradas de até oitava ordem ou filtragem FIR de até 40 coeficientes, e um coprocessador que realiza a compressão do sinal de saída;
- *Cluster* domínio-tempo: possui um coprocessador cuja finalidade é decompor ou recompor o sinal de entrada em bandas de frequência, podendo separar o sinal de áudio de entrada entre duas até 10 bandas.
- *Cluster* domínio-frequência: possui um coprocessador configurável para realizar as operações de FFT e IFFT em palavras de 16, 32, 64, 128 ou 256 *bits*.

O GC5003 é um ASIC, diferente dos outros dois DSP's mencionados que são ASIP's. O GC5003 compõe a plataforma Voyager, mostrada anteriormente, juntamente com o *codec* GC5055.

2.3- AAA's e DSP's

A tendência geral nos projetos de aparelhos portáteis alimentados por bateria é que o consumo seja a principal preocupação, enquanto tamanho e velocidade ficam em segundo plano. Porém, para AAA's, tamanho, que afeta a quantidade de recursos disponíveis, e velocidade, que afeta a quantidade de operações realizadas pelo DSP, não podem ficar em segundo plano. Devido às restrições de consumo, tamanho e velocidade, e dado que os

algoritmos de PDS, desenvolvidos inicialmente em laboratórios, pouco levam em consideração o número de operações possíveis por ciclo ou a quantidade de memória disponível para dados e programas, surgem limites quanto à quantidade de algoritmos a serem implementados e processados em um AAA.

Após pesquisas, foi constatado que a dimensão desejada para um CI que implementa um AAA é de 20 mm^2 e o consumo não deve ultrapassar 1 mW para tecnologias com tensão de alimentação de 1,2 V. Dentro destas condições é possível o desenvolvimento de qualquer dos tipos de AAA. O trabalho de Mosch et al. (2000) apresenta, por exemplo, um *chipset* para aparelhos auditivos utilizando um DSP com desempenho de 50 MOPS e consumo menor que 1mW para tensão de alimentação de 1,2 V.

Frente a essas restrições, os principais AAA's são implementados como ASIC's ou possuem um DSP de ultra-baixo consumo com coprocessadores que trabalham em conjunto, aumentando assim o número de algoritmos possíveis de serem utilizados e mantendo as especificações de projeto. Como mencionado, a utilização de um DSP está relacionada com a flexibilidade do sistema, enquanto a utilização de processamento direto por *hardware* esta relacionada com a melhora da eficiência energética. Infelizmente flexibilidade e eficiência energética são requisitos conflitantes.

Capítulo 3 - PROPOSTA DE SISTEMA – COPROCESSADORES

Neste capítulo será apresentado um SoC para ser utilizado em Aparelhos de Auxílio Auditivo (AAA's) digitais, onde o processamento de sinais digitais é realizado por um DSP e blocos para processamento *hardwired.* Será feita a especificação do DSP, com justificativas para a escolha, e da sua comunicação com os blocos. Também serão detalhadas as implementações dos blocos que foram desenvolvidos para fundamentar a proposta de processamento por *hardware* no lugar de processamento por *software*. A especificação do sistema tem como base os estudos dos aparelhos auditivos, dos DSP's e dos algoritmos mais comuns utilizados em AAA's.

Para uma breve contextualização, SoC's passaram a ser utilizados depois de ser dominante o conceito de integrar todos componentes de um sistema em uma única placa ou *board.* Com o SoC¸ o sistema compõe um único *chip* com um único empacotamento, o que evita problemas de conectividade e torna o sistema mais robusto. Atualmente, os SoC's que aparecem no mercado são guiados pela demanda de aumento de funcionalidades e integram tanto blocos digitais como analógicos, o que possibilita sistemas mais completos e complexos. A dissipação de potência é um dos limitantes fundamentais no aumento da complexidade dos SoC's, pois a alta integração resulta no aumento do consumo.

Foi visto em Smith (1998) que um dos grandes "gargalos" na execução dos algoritmos de PDS é a transferência de informação, i.e. instruções, dados, parâmetros e endereços, entre as unidades de processamento e as memórias RAM, impactando no desempenho e no consumo de potência. Será visto que com a utilização de coprocessadores, é possível a redução no acesso à memória por parte do DSP na execução de rotinas de tratamento do sinal auditivo em um AAA.

3.1 Sistema – Coprocessadores

Com o objetivo de definir um sistema para o processamento digital de um AAA, propomos o SoC da figura 20 como resultado dos estudos realizados. Este SoC é composto pelo codec de entrada, pelo sistema para processamento digital e pelo codec de saída.

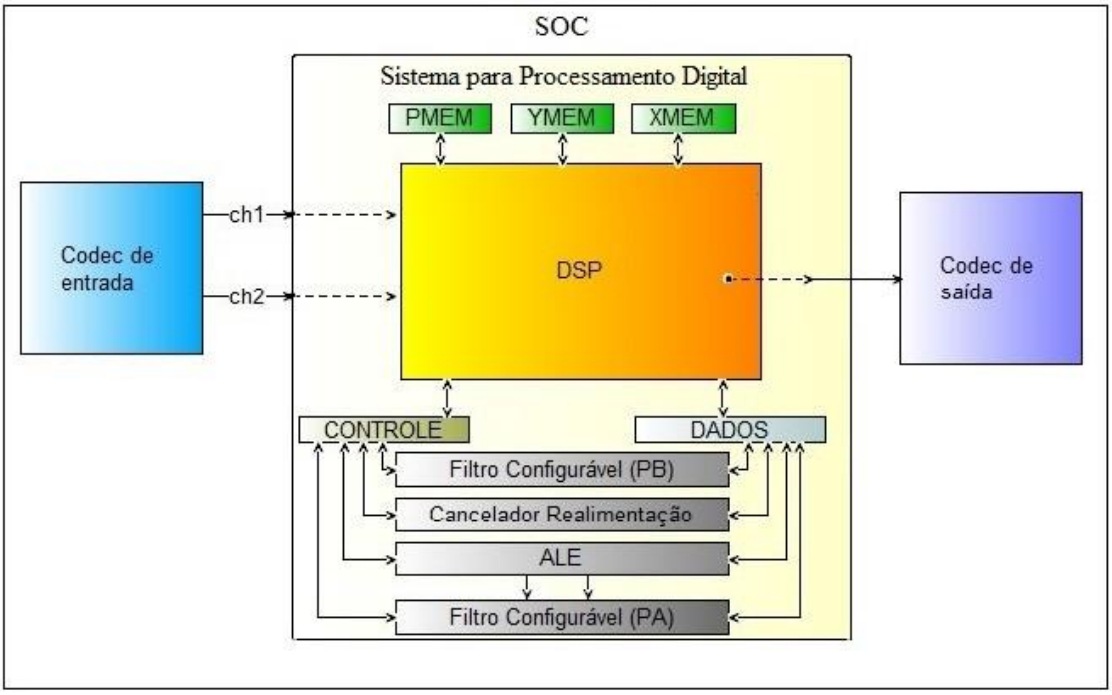

Figura 20: Sistema proposto para processamento digital de um AAA

O codec de entrada, responsável pela passagem do sinal analógico para sinal digital, é formado por dois amplificadores de ganho programável, dois ADC's ΣΔ de segunda ordem, além de seletores e registros de configuração. Os dois ADC's possibilitam o uso de dois microfones omnidirecionais para assim compor dois canais para aquisição sonora. O codec de entrada proposto é como aquele descrito na seção 2.1.3 e figura11.

O sistema para processamento digital, responsável por aplicar os algoritmos de correção e melhoria do som, é composto por:

- um DSP com memórias RAM's XMEM, YMEM e PMEM, para implementar e executar variados algoritmos de PDS;

- três blocos que realizam o processamento *hardwired* (coprocessadores), sendo eles: o Filtro Configurável; o bloco para cancelamento de realimentação acústica; e o bloco para redução de ruídos periódicos, conhecido como Filtro ALE;

- blocos Controle e Dados para a comunicação entre DSP e os coprocessadores. Esta comunicação é sucintamente descrita na próxima seção.

O codec de saída, responsável pela passagem do sinal digital para sinal analógico, é formado por um controle de volume, um interpolador, um bloco de compressão de saída, um DAC ΣΔ de terceira ordem e por um amplificador classe D. O codec de saída proposto é como aquele descrito na seção 2.1.3 e figura13.

Dos blocos que realizam o processamento *hardwired*, de agora em diante chamados apenas de coprocessadores, o Filtro Configurável e o Filtro ALE foram descritos em Verilog, sintetizados e implementados (*layout* completo). O Filtro Configurável pode ser utilizado como filtro passa baixas ou filtro passa altas frequências, dependendo exclusivamente de seus coeficientes, o que será explicado neste capítulo. No sistema proposto na figura 20, têm-se dois filtros configuráveis, um para ser usado como passa baixas (PB) e outro, como passa altas (PA) frequências. Todos os coprocessadores possuem parâmetros para configuração do seu funcionamento.

Os coprocessadores são utilizados para otimizar o sistema, uma vez que a função por eles executada poderia ser feita no DSP através de algoritmos. Existem critérios para escolha de quais processamentos, rotinas de tratamento do sinal sonoro, devam ser realizados por coprocessadores em vez de ser executados no DSP. Neste trabalho, os critérios aplicados foram a funcionalidade das rotinas e a complexidade dos algoritmos executados no DSP. Nos três coprocessadores a operação principal é filtragem, que é realizada através de uma sequência de multiplicações e somas com as amostras do sinal e coeficientes do filtro. Essas sequências exigem alta demanda de processamento do tipo MAC e também alta taxa de transferência de dados entre DSP e Memória RAM, uma vez que para a execução das filtragens é necessário carregar tanto os coeficientes dos filtros como as amostras do sinal.

3.2 DSP e Comunicação

Para melhor compreensão do sistema proposto, nesta seção discutimos a escolha de um DSP e como deve ser a comunicação entre este DSP e os coprocessadores, que recebem os parâmetros de configuração e as amostras a serem processadas.

Visando um baixo consumo e a menor área de CI, dentre os DSP's estudados e mencionados na seção 2.2.4, o Coolflux é a melhor opção para ser utilizada em um aparelho auditivo. É também a melhor opção para se acoplar blocos para processamento *hardwired*, a intenção neste trabalho, pois é disponibilizado como *soft block*, o que permite a síntese do núcleo e dos coprocessadores no mesmo *layout*. Além disso, possui um compilador, que permite o rápido desenvolvimento de rotinas de PDS, e instruções de gerenciamento de consumo de energia. Fazendo o correto uso destas instruções é possível parar o funcionamento do DSP enquanto algum dos coprocessadores estiver realizando um processamento e, desta maneira, economizar potência. Por fim, o Coolflux possui uma

arquitetura simples e pouco escalável, o que colabora para a adição de blocos. De forma diferente, o Icyflex é altamente escalável, o que não o torna uma boa opção para ser utilizado em conjunto com blocos de processamento *hardwired*, mesmo sendo seu núcleo disponibilizado como *soft block* e possuindo um compilador.

O DSP da Gennum (GC5003) já é um ASIC que apresenta uma arquitetura totalmente voltada para aplicações de AAA's, com blocos de processamento dedicado, como por exemplo, o bloco para cancelamento de realimentação. Por ser um ASIC, a adição de novos blocos com processamento *hardwired* não é necessária.

Para a comunicação entre o DSP e os coprocessadores, utilizam-se duas interfaces diferentes, uma para dados e outra para o controle. Cada coprocessador é visto pelo DSP como um periférico com um endereço específico. Assim, para a utilização dos coprocessadores é gerada pelo DSP uma solicitação de interrupção que é enviada através do bloco Controle ao coprocessador endereçado. O bloco Controle também recebe o sinal do coprocessador que informa o término das operações, e envia este sinal ao DSP, que pode estar em um modo de espera. Através do bloco Dados, é enviado ao coprocessador os parâmetros de sua configuração e os dados que serão utilizados em suas operações. O bloco Dados também recebe a resposta do coprocessador e a envia ao DSP.

Existe um caso específico de comunicação que é realizada pelo Filtro ALE a fim de tornar o sistema mais direto e com isso mais eficiente. Dependendo da configuração inicial do Filtro ALE, a sua resposta pode ser direcionada diretamente a um Filtro Configurável, como mostrado no sistema da figura 20. O sinal de controle informando o término de sua operação também é direcionado diretamente ao Filtro Configurável.

3.3 Filtro Configurável – FIR

Foi observado que a operação de filtragem é a mais frequente para o processamento digital em AAA's. Considerando o domínio da frequência, podemos dividir os filtros mais simples em quatro tipos, são eles:

- Filtro passa-baixas: deixa passar os sinais com frequências abaixo de uma frequência específica, definida como frequência de corte, e atenua os sinais com frequências acima desta frequência especifica;

- Filtro passa-altas: deixa passar os sinais com frequências acima da frequência de corte, e atenua os sinais com frequências abaixo;

- Filtro passa-faixa: deixa passar os sinais com frequências que estejam entre duas frequências de corte, e atenua os sinais com frequências acima ou abaixo destas duas frequências de corte;

- Filtro rejeita-faixa: deixa passar os sinais com frequências acima ou abaixo de duas frequências de corte, e atenua os sinais com frequências que estejam entre as duas frequências de corte. O filtro *notch*, muito utilizado para eliminar ruídos, é um tipo de filtro rejeita-faixa que possui as frequências de corte muito próximas, rejeitando assim apenas uma frequência especifica.

O filtro de resposta finita ao impulso, conhecido como filtro FIR, é amplamente utilizado para implementar operações de filtragem em AAA's. Isso se deve principalmente ao fato do filtro FIR ter resposta com fase linear na sua banda passante, o que significa que o atraso no sinal, devido às operações, será igual para todas as frequências. Esta é uma característica fundamental para aparelhos auditivos, uma vez que a fase do sinal sonoro é uma informação importante.

Para o projeto de um filtro FIR, consideram-se as características desejadas no domínio da frequência, especificando as oscilações (*ripple*) das bandas de passagem (A_{pass}) e de rejeição (A_{stop}), a frequência de passagem (F_{pass}) e a frequência final de rejeição (F_{stop}), como mostrado na figura 21 para um filtro passa baixas.

É importante mencionar que a diferença entre a F_{pass} e F_{stop} é conhecida como faixa de transição, sendo que quanto menor esta faixa, maior será o número necessário de coeficientes para o filtro, caso sejam mantidas as outras especificações.

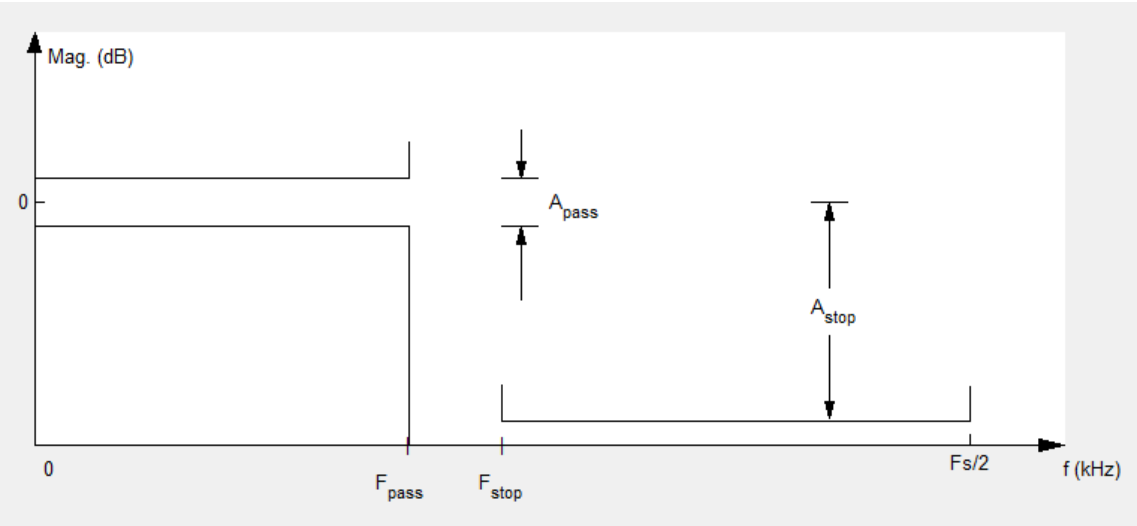

Figura 21: Especificação das variáveis para se projetar filtro FIR

Uma vez especificadas as características desejadas do filtro, existem diversos métodos (algoritmos) para obtenção da resposta impulsiva do filtro. Os valores da resposta impulsiva do filtro são os coeficientes que o caracterizam e usando esta resposta, ou esses coeficientes, é possível se obter a equação de operação de filtragem de um sinal, como mostrado:

$$y(n) = b_0 * x(n) + b_1 * x(n-1) + ... + b_{M-1} * x(n-M+1) = \sum_{k=0}^{M-1} bk * x(n-k) \text{ , (1)}$$

onde y(n) é o sinal de saída, x(i) é o sinal de entrada, b_k é o conjunto de coeficientes e M é o número de coeficientes do filtro.

A operação de filtragem, como pode ser visto, requer M multiplicações e M-1 somas. Quando na implementação do filtro FIR se aplica uma estrutura onde se multiplicam as amostras do sinal digital atrasado pelos coeficientes, sabendo que o sinal é atrasado até o número de coeficientes do filtro, se diz que é uma implementação na forma direta 1, também conhecida como *tapped delay line* ou filtro transversal (MITRA, 2001).

O coprocessador desenvolvido em Verilog para realizar o Filtro Configurável, realiza o processamento *hardwired* para diferentes tipos de filtros FIR, uma vez que sua configurabilidade se refere ao número e aos valores dos coeficientes.

O coprocessador tem um limite de até 128 coeficientes. Este valor foi escolhido após a elaboração de diversos filtros FIR, com diferentes bandas de transição e diferentes atenuações, implementados em Matlab com o *Filter Design and Analysis Tool* (FDATool), utilizando o algoritmo *equiripple* para obtenção dos coeficientes (resultados apresentados na seção 4.1.1, tabela 6).

3.3.1 Matlab – FDATool

O FDATool é uma interface gráfica do *toolbox* de processamento de sinais do Matlab, usado para projetar e analisar filtros (MATLAB, 2013). Através do FDATool é possível projetar filtros FIR com as especificações desejadas, analisar a resposta em frequência e também a resposta impulsiva. Através dos valores da resposta impulsiva, determinam-se os coeficientes do filtro para sua implementação.

Como exemplo de utilização, na figura 22 é mostrado um filtro passa baixas projetado pelo FDATool com as seguintes especificações: F_{pass} = 4,0 kHz e F_{stop} = 6,0 kHz, A_{pass} = 0,01 dB e A_{stop} = 80 dB.

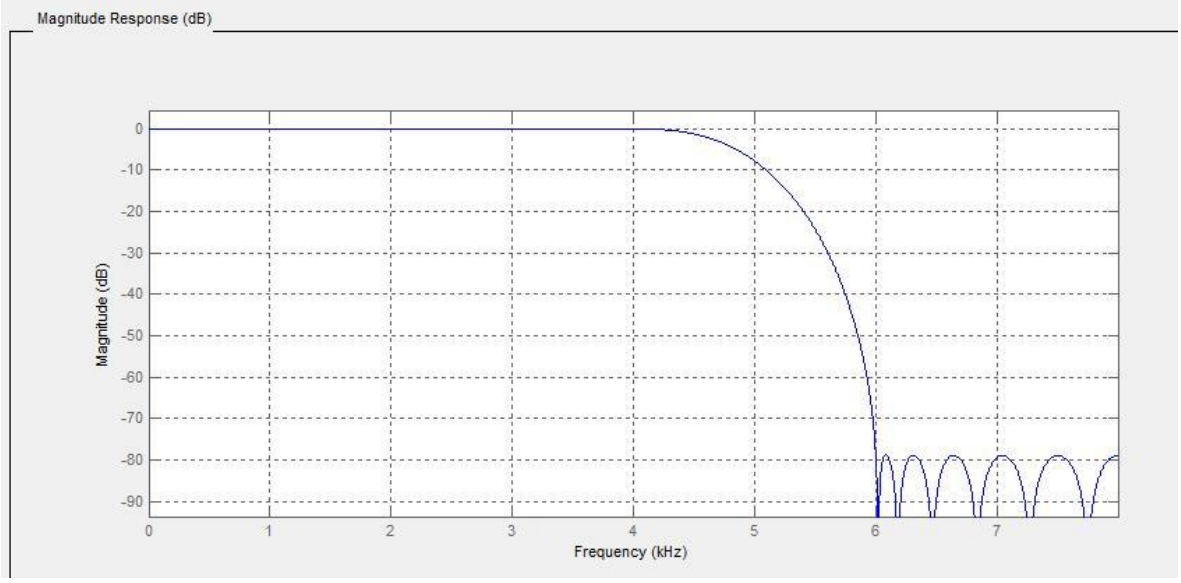

Figura 22: Resposta de um filtro passa baixas no domínio da frequência. Especificações: F_{pass} = 4,0 kHz e F_{stop} = 6,0 kHz, A_{pass} = 0,01 dB e A_{stop} = 80 dB

Pode-se observar que a banda de transição é de 2,0 kHz, sendo que o filtro começa a atenuar o sinal depois dos 4,0 kHz e só se tem a atenuação de 80 dB com 6,0 kHz.

Observando a resposta impulsiva deste filtro (Figura 23) gerada pelo FDATool, é possível constatar que ela contêm 33 saídas (coeficientes). Uma característica importante para implementação de filtros FIR com fase linear é que sua resposta impulsiva é simétrica, como pode ser observado na figura 23. Isto significa que os valores dos coeficientes são simetricamente idênticos, ou seja, o primeiro valor, b_0, e o último, b_{M-1}, são idênticos.

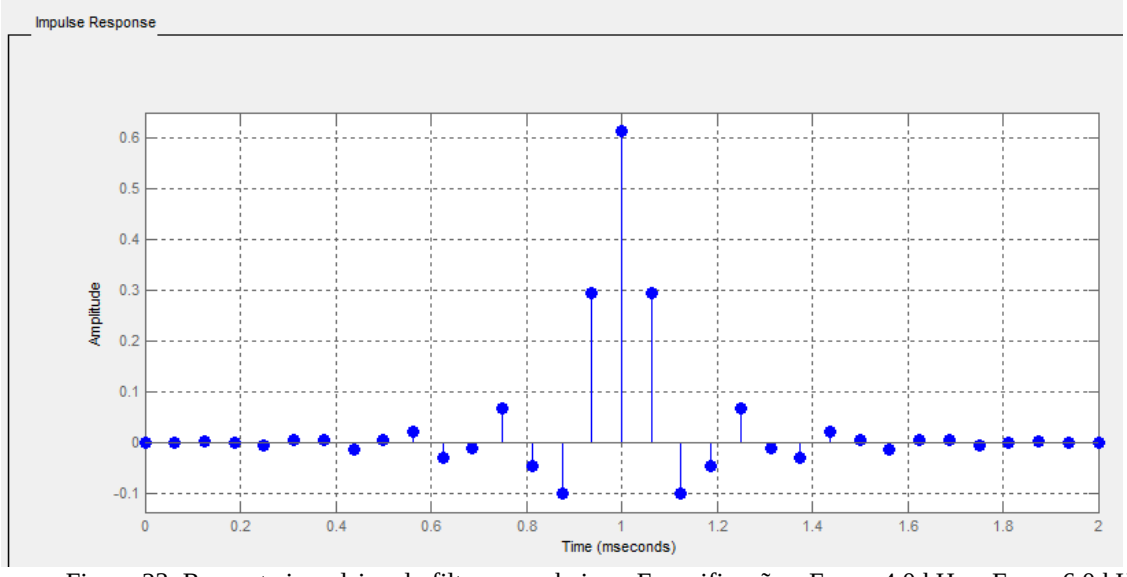

Figura 23: Resposta impulsiva do filtro passa baixas. Especificações: F_{pass} = 4,0 kHz e F_{stop} = 6,0 kHz, A_{pass} = 0,01 dB e A_{stop} = 80 dB

A simetria permite a redução na complexidade do filtro através da redução do número de multiplicações, uma vez que é possível realizar a soma de duas amostras 'simétricas' e posteriormente realizar a multiplicação pelo coeficiente. Para exemplificar, considere a amostra atual, que seria multiplica pelo coeficiente b_0, e a amostra mais antiga, que seria multiplicada pelo coeficiente b_{M-1}, que é igual a b_0. Devido à simetria é possível somar essas duas amostras e realizar apenas uma multiplicação.

Além de diminuir a complexidade do filtro, uma implementação utilizando a simetria permite que se armazene apenas a metade dos coeficientes.

3.3.2 Arquitetura do filtro configurável implementado

Sabe-se que a operação de filtragem é computada por meio dos coeficientes do filtro, que definem a resposta em frequência, e das amostras de dados que são armazenadas e lidas ao longo do processo de filtragem. Cada nova amostra que chega ao filtro é armazenada, junto com as M-1 amostras que a precederam. O processo de armazenamento pode ser realizado por um banco de registradores de deslocamento, conhecido como *shift register*, ou por uma estrutura de memória conhecido como memória circular ou *circular buffer* em inglês.

Com o *shift register*, para cada nova amostra, as M amostras armazenadas se deslocam de um registrador para o outro, sendo a mais antiga descartada. Diferentemente, na memória circular não se realiza o deslocamento da amostra, pois esta estrutura utiliza um ponteiro de escrita que irá apontar para a posição da memória onde a nova amostra deve ser salva. O ponteiro de escrita é incrementado a cada nova escrita e quando ele atinge para a última posição, o incremento faz com que ele aponte novamente para a primeira posição da memória, fechando assim um círculo. Além do ponteiro de escrita, a memória circular utiliza dois ponteiros de leitura que apontam para a posição de duas amostras simétricas, ou seja para b_0 e b_{M-1} ou b_1 e b_{M-2}, etc.. Estas amostras simétricas são aquelas que devem ser somadas na operação de filtragem.

O filtro configurável foi implementado com a arquitetura apresentada na figura 24 e possui:

- uma memória circular para o armazenamento das amostras;
- uma memória para o armazenamento dos coeficientes;
- um somador (somador 1) para realizar a soma de duas amostras devido a simetria dos coeficientes;

- um multiplicador para a multiplicação dos coeficientes;

- um somador/registrador que irá somar os resultados das multiplicações e registrar o valor da reposta final do filtro.

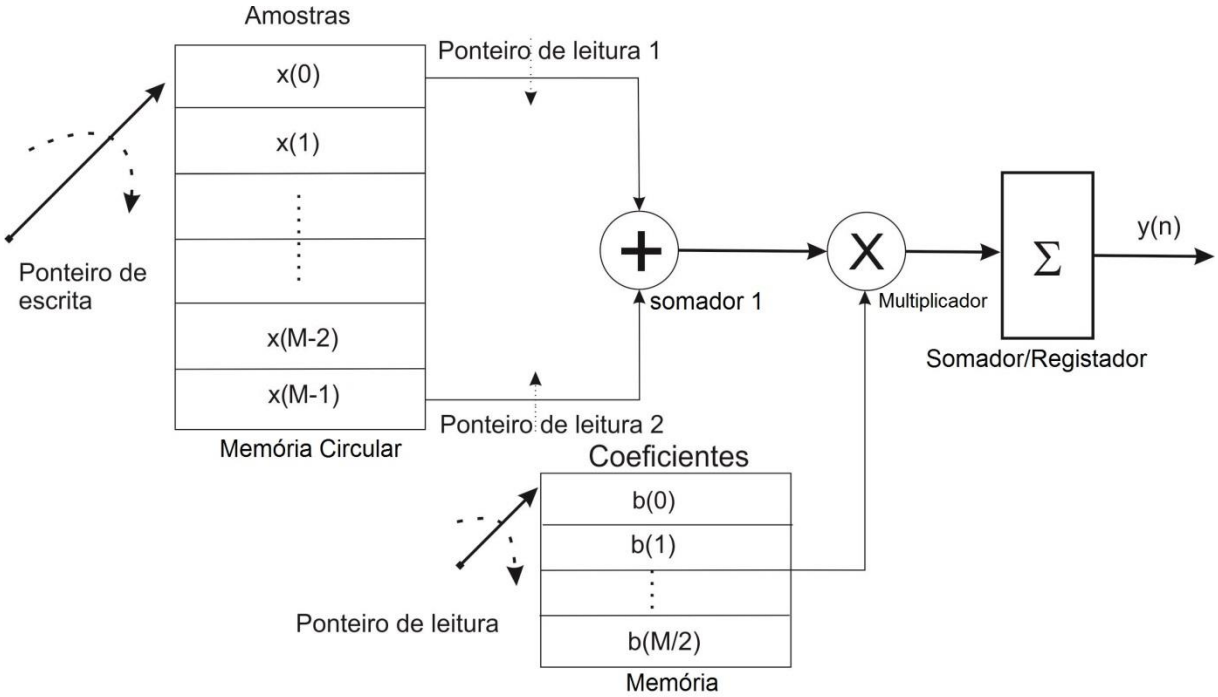

Figura 24: Arquitetura do filtro configurável

A utilização da memória circular é justificada pela comparação feita em Carvalho e Navarro (2012), onde se mostrou que, em questão de consumo de potencia, filtros com memória circular são mais de quatro vezes mais eficiente, comparados à implementação com registradores de deslocamento. Implementações com memória circular tem como desvantagem o aumento da área devido à lógica de controle dos ponteiros de escrita e leitura.

A utilização de apenas um multiplicador na arquitetura do filtro configurável define sua arquitetura como serial. Nesta arquitetura há o compartilhamento de *hardware* à custa de frequência do *clock* muitas vezes maior que à taxa de entrada de dados. No caso do Filtro Configurável, tem-se o compartilhamento do *hardware* do multiplicador entre todos os coeficientes. Por outro lado, a utilização de um multiplicador para cada coeficiente define uma arquitetura paralela. A arquitetura paralela é mais eficiente em questão de consumo de potência, pois se pode realizar as multiplicações com uma frequência de *clock* poucas vezes maior que a taxa de entrada das amostras do filtro, porém não permite a reconfiguração do filtro para a utilização de um número de coeficientes variados.

O filtro configurável terá 24 *bits* de resolução para as amostras de entrada e 16 *bits* para os coeficientes. A resolução da amostra de saída também será de 24 *bits*, sendo a saída obtida através de arredondamento e truncamento do valor no registrador que contem o valor final das operações. Este registrador tem 46 *bits* para garantir que não haja erros de *overflow* devido ao crescimento do número de *bits* após cada operação. Os arredondamentos e truncamentos podem ser observados no código Verilog da implementação do Filtro Configurável, que está como apêndice A.

3.3.3 Interface e Operação

O filtro configurável têm a interface mostrada na figura 25 e sua operação está descrita no código Verilog apresentado no apêndice A. A descrição dos sinais de interface é dada abaixo:

- ***clk*** e ***rst_n*** (1 *bit*): sinas de relógio (*clock*) e inicialização (*reset*) para os filtros;
- ***input_ch1*** e ***input_ch2*** (24 *bits*): as duas entradas para as amostras dos filtros;
- ***coeff0_127, coeff63_64*** (16 *bits*): entradas para os coeficientes dos filtros. São no total 64 coeficientes para cada filtro;
- ***filter_lenght*** (8 *bits*): sinal de entrada que informa o número de coeficientes do filtro;
- ***sample_in*** (1 *bit*): sinal de entrada para informar a disponibilidade de nova amostra;
- ***num_channel*** (1 *bit*): sinal de entrada usado para informar se deve processar duas amostras ou apenas uma, possibilitando a redução do consumo de potência;
- ***data_to_dsp_ch1*** e ***data_to_dsp_ch2*** (24 *bits*): as duas saídas com as respostas do filtro;
- ***sample_out*** (1 *bit*): sinal de saída que avisa o término da filtragem.

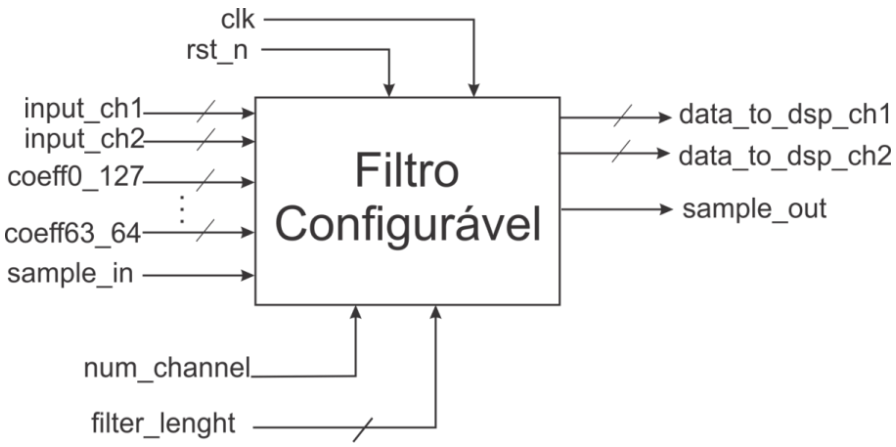

Figura 25: Interface do filtro configurável

A expressão para as respostas dos canais 1 e 2 é:

data_to_dsp_ch = $\sum_{k=0}^{filter-lenght-1} coeff * input_ch(n-k)$

A operação do filtro configurável é da seguinte forma: inicialmente o filtro recebe os parâmetros de configuração, que são: o número de coeficientes *(filter_length)*, os coeficientes e número de canais (*num_channel*). Como mostrado na figura 25, o filtro desenvolvido contêm duas entradas, possibilitando a recepção e filtragem de dois canais, i.e. dois microfones omnidirecionais, porém pode ser configurado para receber apenas um canal, como para o caso do *telecoil* ou DAI. Após a etapa de configuração, e já em operação, ele irá receber cada nova amostra, gerada no codec de entrada, através do DSP, que envia as amostras dos canais através do bloco Dados e o sinal de controle *sample_in* através do bloco Controle. O filtro irá retornar as respostas da filtragem e o sinal de controle *sample_out*.

Cada uma das amostras ficará armazenada em uma memória local do coprocessador, a memória circular. O número de amostras armazenadas depende do número de coeficientes do filtro. Após a escrita na ultima posição da memória, começa a sobrescrita das amostras já armazenadas.

Devido à memória local do coprocessador, o DSP acessa a memória RAM (XMEM ou YMEM) apenas na configuração e para salvar o resultado da operação do coprocessador, reduzindo o total de acessos, uma solução para o "gargalo" visto em (SMITH, 1998).

3.3.4 Considerações Finais

Para a implementação dos filtros digitais em *hardware*, há o problema do erro de quantificação, devido à redução do número de *bits* na representação dos valores dos coeficientes. Os coeficientes gerados pelo Matlab através da resposta impulsiva tem uma precisão "infinita", porém ao utilizar aos valores na implementação em *hardware*, a precisão foi reduzida para 16 *bits* em complemento de dois (1 *bit* para sinal e 15 para a magnitude do valor). Isto tem impacto no SNR do filtro e será melhor explicado no capítulo de simulações.

Uma importante consideração a ser feita diz respeito ao armazenamento das amostras. Para minimizar o impacto em termo de área desse armazenamento, foram utilizados registrados sem *reset*, o que pode impactar na redução de até 40 % da área do *flip-flop*. Em questão de funcionalidade não há perda pela falta do *reset*, uma vez que os primeiros valores gerados pelo filtro, as amostras inicias, são descartadas. Isso ocorre até que o filtro entre em regime de operação, como será mostrado no capítulo de simulações.

3.4 Filtro ALE

O filtro ALE é um filtro para eliminação de ruído muito utilizado em processamento de fala e por isso aplicado em aparelhos auditivos. Sua utilização se deve aos bons resultados obtidos na eliminação do ruído e também por ser um coprocessador de baixo consumo de potência. Possui apenas uma entrada, diferente de outros coprocessadores de eliminação de ruído, e emprega um filtro adaptativo composto por um filtro IIR ou um FIR. O filtro FIR tem a vantagem da simplicidade e da estabilidade durante o processo de adaptação (YAN, 2006).

O filtro ALE é baseado nos conceitos de predição linear e autocorrelação. Um sinal periódico pode ser perfeitamente previsto utilizando-se combinações lineares de suas últimas amostras, enquanto que para um sinal não periódico é impossível tal previsão. A predição linear estima os valores futuros de um sinal periódico, permitindo sua eliminação. A autocorrelação se refere à correlação de um sinal com seus próprios valores passados e futuros. Ela permite a criação de um sinal desejado (futuro) baseado nas amostras recebidas (passado) (AVALOS; SANCHEZ; VELAZQUEZ, 2011).

Aplicando os conceitos mencionados, o filtro ALE utiliza o seu sinal de entrada, d[n], para criar um sinal atrasado, x[n], que será utilizado como referência para o filtro adaptativo eliminar sinais periódicos, como mostrado na figura 26. O atraso é realizado através de cascateamento de *flip-flops*, representado por D também na Figura 26. O erro e[n], a diferença entre a saída do filtro, y[n], e o valor atual do sinal de entrada, é utilizado pelo algoritmo adaptativo para ajustar os coeficientes de um filtro FIR, o que será melhor explicado na próxima seção.

O filtro ALE desenvolvido neste trabalho para ser implementado em *hardware* utiliza o algoritmo *Least Mean Square* (LMS) para realizar as atualizações (ajustes) dos coeficientes do filtro FIR.

Existem dois importantes parâmetros que devem ser escolhidos para o correto funcionamento do filtro ALE: o valor do atraso D, que depende da autocorrelação do sinal de interesse, e o número de coeficientes do filtro FIR utilizado. Ambos têm impacto direto na área e consumo de potência do coprocessador.

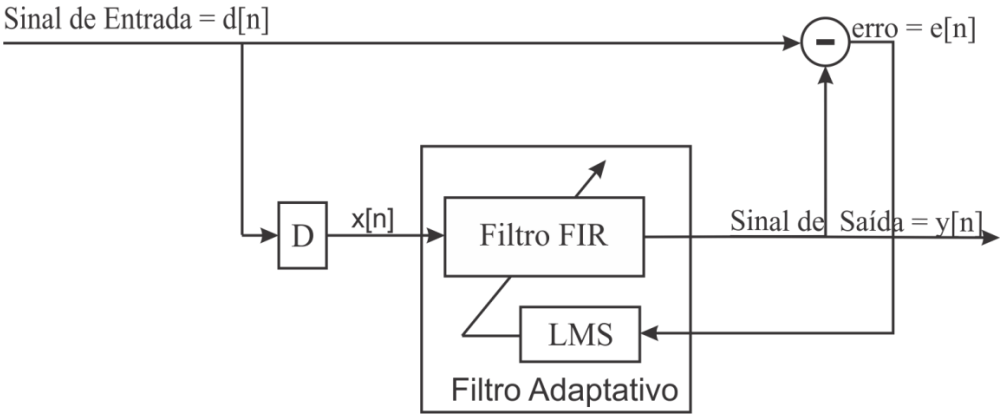

Figura 26: Estrutura do Filtro ALE

3.4.1 *Least Mean Square* – LMS

O algoritmo LMS é um algoritmo de filtragem adaptativa linear de grande simplicidade computacional. A versão amplamente utilizada é uma derivação da versão inicialmente proposta por Bernard Widrow e Marcian E. Hoff em 1960 e é baseada no método *steepest descent*, um método de otimização para encontrar o mínimo erro médio quadrado. O LMS é usado para determinar os coeficientes do filtro FIR (HAYKIN, 2005). A partir dos sinais mostrados na figura 26, as equações que definem o algoritmo LMS podem ser sumarizadas como:

$$y[n] = x[n]^T * w[n], \quad (1)$$

$$e[n] = d[n] - y[n], \quad (2)$$

onde y[n] é a saída do filtro FIR adaptativo, e[n] é o sinal de erro dado por (2), x[n] são as amostras recebidas com atraso e armazenadas no filtro FIR, e w[n] são os coeficientes do filtro FIR, os quais são ajustados.

O ajuste dos coeficientes é feito recursivamente pela equação 3:

$$w[n-1] = w[n] + \mu * e[n] * x[n], \quad (3)$$

onde μ é um paramento conhecido como passo de adaptação, em inglês *step size*, o qual controla a taxa de convergência do algoritmo LMS.

A escolha do adequado valor de μ é extremamente importante para o correto funcionamento do filtro adaptativo. Um valor pequeno de μ pode levar a um longo tempo de convergência, ao passo que um valor grande pode levar a divergência do algoritmo, assim degradando a estimativa de erro do filtro adaptativo. Em certos casos, como quando se sabe o tipo de ruído do sinal de entrada, o melhor valor de μ para a convergência do algoritmo LMS está dentro de um faixa que depende das características do ruído. Por exemplo, para um ruído branco de entrada com média zero e variância σ^2_u, a condição de convergência é:

$$0 < \mu < 2/\sigma^2_u$$

3.4.2 Matlab – ALEDEMO

O Matlab possui um demonstrativo (demo) chamado **Aledemo** que ilustra a utilização de filtros ALE. Neste demo é feita a amostragem, com uma frequência de 8192 Hz, de um pequeno trecho da música "Aleluia" de Handel, criando assim o sinal Xi, cujos valores máximos e mínimos são 0,4 e -0,4. É também criado um sinal V, o qual é uma senóide de 1,0 kHz, cujos valores máximos e mínimos são 0,5 e -0,5, amostrada na mesma frequência de 8192 Hz. Com a soma do sinal Xi e V é composto o sinal Xin. Este sinal é aplicado ao filtro ALE para testes, o qual gera o sinal Y como saída.

Na figura 27 são mostrados os sinais Xi, Xin e Y, criados a partir do *script* do apêndice C, desenvolvido para validar a implementação do filtro ALE descrito em Verilog. Percebe-se que o sinal de saída do filtro ALE está muito próximo do sinal originalmente amostrado, ou seja, sinal sem ruído.

A partir do demo foi possível determinar os valores do passo de adaptação μ e do tamanho do atraso do sinal na implementação feita neste trabalho.

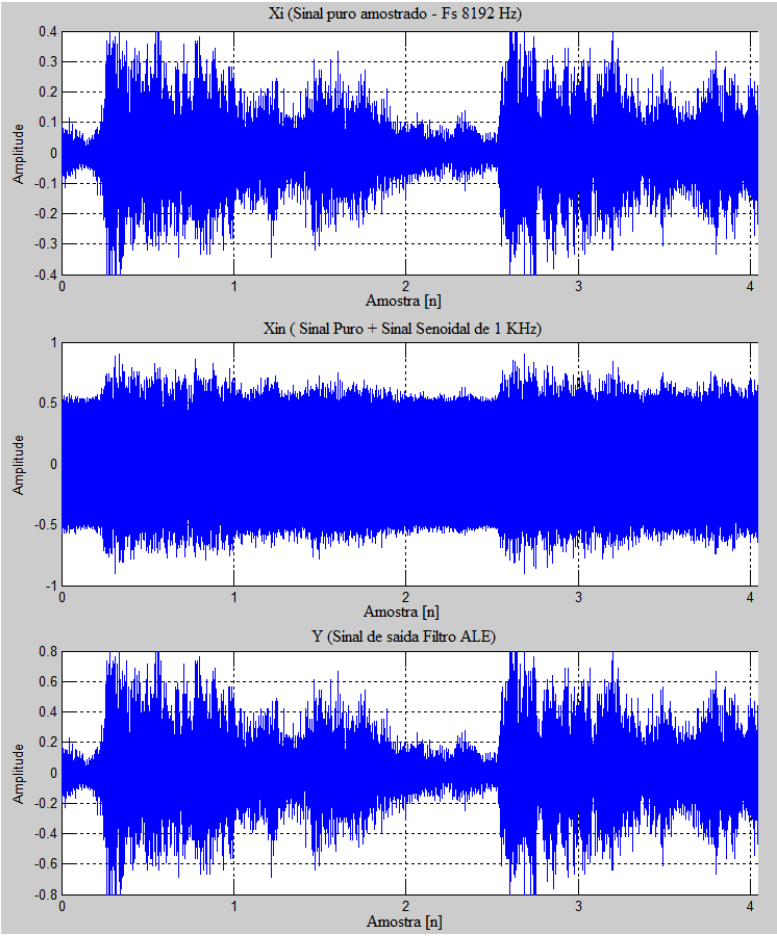

Figura 27: Sinal de entrada, sinal de entrada mais ruído em 1,0 kHz e saída do filtro ALE implementado em Matlab

O sinal de erro mostrado na figura 28 é o sinal resultante da diferença entre os sinais de entrada Xin e de saída do filtro ALE. Como mencionado anteriormente, este sinal é utilizado para ajustar os coeficientes do filtro FIR. Percebe-se que o sinal de erro é senoidal, assim como o ruído que foi somado ao sinal Xi.

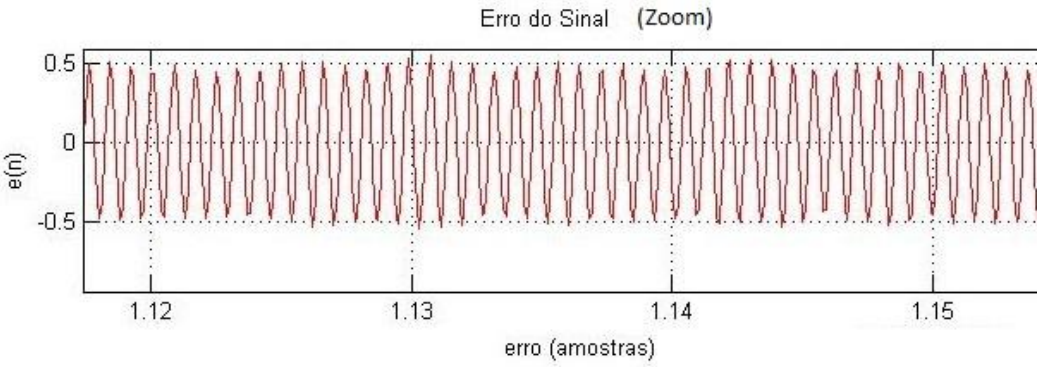

Figura 28: Sinal de erro do filtro ALE implementado em Matlab

3.4.3 Arquitetura do filtro ALE implementado

O filtro ALE tem 24 *bits* de resolução para as amostras de entrada e 16 *bits* para os coeficientes. A resolução da amostra de saída também será de 24 *bits*, sendo a saída obtida através de arredondamentos e truncamentos do valor no registrador que contem o valor final das operações. Este registrador tem 47 *bits*. O truncamento pode ser observado no código Verilog da implementação do filtro ALE, que está como apêndice B. Também para o filtro ALE, foi feito o truncamento dos valores dos coeficientes.

A arquitetura do filtro ALE é apresentada na figura 29, que mostra o sinal de entrada d[n], o sinal x[n] obtido através do atraso do sinal d[n], o sinal e[n] que é obtido pela subtração do sinal d[n] pela saída do somador, o sinal y[n], que é também a saída do filtro ALE. Observe que o sinal e[n] é multiplicado por μ e o resultado da multiplicação é utilizado para ajustar os coeficientes do filtro: w0 até w23.

Na figura os operandos Z^{-1} e Z^{-D} representam operações de atraso, sendo Z^{-1} o atraso de um (1) ciclo de *clock* e Z^{-D} o atraso de D ciclos de *clocks*.

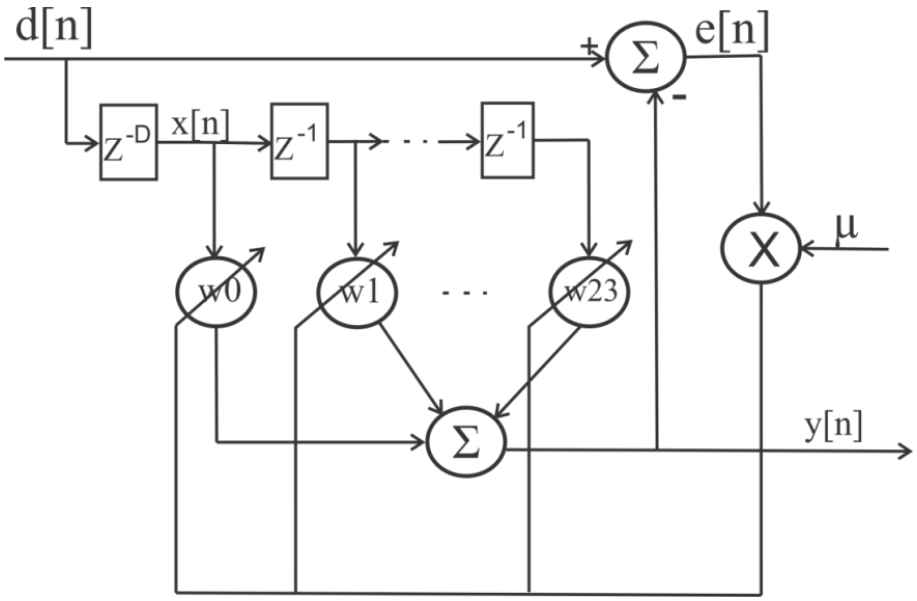

Figura 29: Arquitetura do filtro ALE implementado em Verilog

Para o filtro ALE, mesmo utilizando um FIR, não há simetria entre os coeficientes, e com isso não se pode realizar a soma de duas amostras antes da multiplicação pelo coeficiente. Foram utilizados registradores de deslocamento para armazenar e deslocar as amostras x[n], como pode ser observado na figura 29. A justificativa para se usar

registradores de deslocamento ao invés de memória circular é o número de coeficientes: com um número reduzido de coeficientes, no caso 24 coeficientes, o aumento da área de circuito devido à lógica de controle dos ponteiras da memória circular não compensa a redução no consumo de potência que é obtido.

Para esta implementação, o atraso D foi de apenas um (1) ciclo de *clock*, o que pode não ter sido o bastante, como será mostrado nas simulações. A justificativa para este valor é a restrição de área do coprocessador, pois para cada atraso de (1) ciclo de *clock* no sinal de entrada d[n], são necessários 24 *flip-flops*.

O filtro ALE foi implementado com 24 coeficientes, sendo que tal valor foi obtido através de simulações no Matlab. As simulações mostraram que com 16 coeficientes não se tem adequada redução do ruído, restando um ruído audível. Com 32 coeficientes, por outro lado, o filtro elimina completamente o ruído, porém não apresenta grande melhoria comparado a implementação com 24 coeficientes. Assim, visando menores área e consumo, foram utilizados apenas 24 coeficientes.

Para finalizar, a arquitetura de multiplicadores do filtro ALE é paralela, ou seja, utiliza um multiplicador dedicado para cada coeficiente. A escolha da arquitetura paralela também se deve ao número de coeficientes. Com os 24 multiplicadores, é possível realizar todas a multiplicações em um único ciclo de *clock*, o que possibilita que a frequência de operação deste coprocessador seja baixa podendo chegar à ordem de centenas de kHz. Dessa forma se diminui o consumo de potência. A resposta de todas as multiplicações é somada por um único somador, mostrado na figura 29.

3.4.4 Interface e Operação

O filtro ALE têm a interface mostrada na figura 30 e sua operação está descrita no código Verilog apresentado no apêndice B. A descrição dos sinais de interface é dada abaixo:

- ***clk*** e ***rst_n*** (1 *bit*): sinas de relógio (*clock*) e inicialização (*reset*) para os filtro;
- ***data_from_dsp*** (24 *bits*): entrada para as amostras de 24 *bits* do filtro;
- ***sample_in*** (1 *bit*): sinal de entrada para informar a disponibilidade de nova amostra;
- ***out_filter*** (1 *bit*): sinal de entrada que seleciona qual é a saída do filtro ALE;
- ***mu*** (4 *bits*): valor do coeficiente μ utilizado pelo filtro;

- ***data_to_dsp e data_to_filter*** (24 *bits*): as duas possíveis saídas com a resposta do filtro. O filtro ALE pode ter sua resposta direcionada diretamente a um filtro configurável ou ser enviada de volta ao DSP;
- ***sample_out*** (1 *bit*): sinal de saída que avisa o término da filtragem.

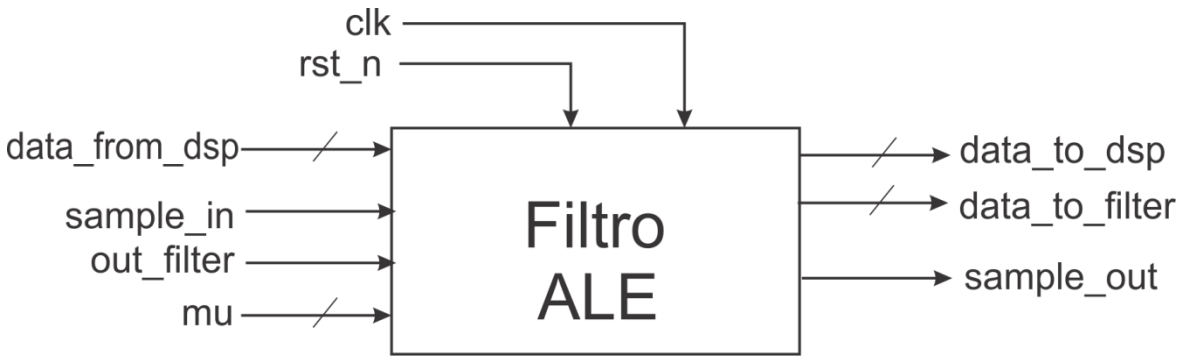

Figura 30: Interface do filtro ALE

A operação do filtro ALE é da seguinte forma: inicialmente o filtro recebe os parâmetros de configuração, que são: o valor de μ e a seleção da saída (out_filter). O valor de out_filter determina se a resposta do filtro ALE será enviada de volta ao DSP ou para o filtro configurável de saída, que é um passa altas frequências. Essa segunda opção segue o fluxo descrito em 2.1.3.

O filtro ALE recebe as amostras do DSP através do bloco Dados, após ele ter realizado o processamento para o controle de ganho automático, e o sinal de controle *sample_in* através do bloco Controle. Após o término das operações do filtro ALE, é gerado o sinal *sample_out*, que é enviado ao bloco Controle ou ao filtro configurável de saída. A seleção é feita também pelo valor de out_filter.

O total de 24 amostras ficarão armazenadas em um banco de registradores local, também para reduzir o acesso à memória RAM pelo DSP.

Para cada nova amostra que o DSP envia ao filtro ALE, primeiro é enviado o sinal de controle *sample_in.* Este sinal é utilizado por uma máquina de estado, também descrita em Verilog, que controla a sequência de operações do filtro ALE. A sequência dos 10 estados dessa máquina está descrita na figura 31 e para maiores detalhes da implementação e funcionamento ver apêndice B.

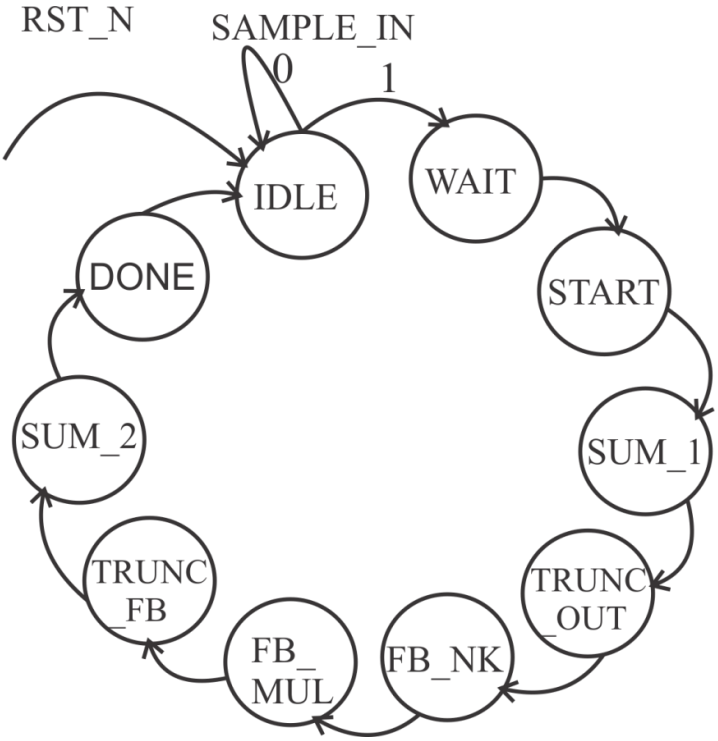

Figura 31: Máquina de estado do filtro ALE que controla os processos do filtro

Existem 8 registradores de 1 *bit* controlados pela máquina de estado, e cada um desses registradores ativa um processo do filtro. Para cada estado, exceção dos estados IDLE e WAIT quando os registradores são zerados, um dos registradores tem sua saída ativada em 1. Assim, por exemplo, quando a máquina estiver no estado SUM_1 é realizado a soma dos resultados das multiplicações das amostras pelos coeficiente. No estado TRUNC_OUT, um registrador de 24 *bits* recebe o valor truncado do resultado de SUM_1 e assim armazenando a resposta final das operações para filtragem.

Capítulo 4 - SIMULAÇÕES E RESULTADOS

Neste capítulo serão apresentados e analisados os resultados obtidos pelas simulações, ao nível RTL (*Register Transfer Level*), dos coprocessadores implementados em Verilog, começando pelo Filtro Configurável e finalizando com o Filtro ALE. Através das simulações foi possível comprovar a funcionalidade dos coprocessadores. Também serão mostrados e discutidos os resultados de consumo de potência e área obtidas das sínteses dos filtros.

As simulações foram realizadas no **Modelsim**, uma ferramenta da Mentor Graphics utilizada para compilar e realizar simulações de códigos Verilog. A partir do Modelsim foi possível gerar figuras que mostram os sinais de entrada e saída de cada coprocessador e também analisar os valores dos mesmos, para comprovar a atenuação e eliminação de ruídos.

Os sinais de estímulo aplicados aos coprocessadores tem precisão de 24 *bits* representados em complemento de dois. Tal representação é a mais utilizada nos DSP's pois permite que sejam utilizadas as mesmas regras para as operações de soma e subtração, diminuindo a complexidade do *hardware*. Por esta razão foi considerada a melhor opção de representação de sinais.

Como resultado físico, foi gerado o *layout* dos coprocessadores implementados em Verilog, feito na tecnologia CMOS 0,35µm da AMS (*Austria Micro Systems*), tensão de alimentação 1,8 V e três níveis de metal. A síntese foi realizada com as células digitais do *design kit* e com o modelo típico de caracterização. O fato de se usar células com o modelo típico para síntese tem grande influência nas características temporais e de consumo de cada célula.

A biblioteca de células digitais fornece informações físicas, como tamanho e funcionalidade, e temporais, como tempo de transição da saída, de cada célula, i.e. *flip-flop*, portas lógicas, somadores, etc.. Estas informações foram utilizadas pelos programas **RTL Compiler** e **SoC Encounter** durante a geração do *layout*, ambas as ferramentas da Cadence, empresa que fornece ferramentas EDA (*Electronic Design Automation*) para o completo desenvolvimento de ASIC's.

O RTL Compiler realiza a síntese de um código RTL (Verilog comportamental), gerando um *netlist*[6], que é um Verilog estrutural. Além do código RTL e da biblioteca de células digitais, o RTL Compiler precisa de um arquivo de *constraints* (restrições), que é usado para a criação do arquivo de definição da árvore de *clock (Clock.ctstch)*, que será utilizado posteriormente pelo SoC Encounter. O SoC Encounter realiza o *place*, posicionamento das células dentro de uma área específica, e o *route*, conexões físicas entre os sinais das células, do circuito descrito no Verilog estrutural. Ele também cria os anéis de alimentação (vdd e gnd) e a árvore de *clock* baseada no arquivo *Clock.ctstch*. Além do *netlist* e da biblioteca de células digitais, o SoC Encounter precisa da descrição de *layout* de cada célula, fornecida por um arquivo LEF (*Layout Exchange Format)* para o *place&route*. Foi a partir do SoC Encounter que se obtiveram as estimativas de consumo de potência e área dos coprocessadores sintetizados.

4.1 Filtro Configurável

As simulações do Filtro Configurável tiveram como estímulos os sinais criados a partir do *script* do apêndice C, desenvolvido no Matlab.

A amplitude dos sinais foi mantida dentro de 66% da faixa de quantização da entrada para fazer os sinais mais realísticos, considerando um codec de entrada com um conversor ADC ΣΔ de segunda ordem. A consideração de amplitude se deve ao fato que conversores ΣΔ de segunda ordem, geralmente, são projetados para trabalharem com sinais de entrada com excursão máxima de 75% da faixa de quantização, para assim garantir a estabilidade do modulador (CUBAS, 2013).

Inicialmente, analisamos o número de ciclos de *clock* necessários para que o filtro realize todas as operações. No caso do Filtro Configurável, o número de ciclos de *clock* necessários está relacionado com o número de coeficientes do filtro. Na tabela 5 é mostrada a relação entre o número de coeficientes e o número de ciclos de *clock*:

[6] Neste trabalho o termo *netlist* será utilizado para se referir a um Verilog estrutural, como aquele gerado pelo RTL Compiler.

Tabela 5: Relação entre número de coeficientes e número de ciclos de *clock* para o Filtro Configurável

Número de Coeficientes	Número de ciclos de *Clock*
32	24
64	40
127	72

Observa-se que a relação entre o número de coeficientes e o número de ciclos de *clock* é dada por:

(número de ciclos de *clock*) = (número de coeficientes/2) + 8.

Caso o número de coeficientes seja ímpar, faz-se a divisão considerando um coeficiente a mais.

4.1.1 Resposta Impulsiva – Filtros Passa Baixa

Como mencionado acima, os valores da resposta impulsiva de um filtro, que tem suas características no domínio da frequência especificadas no FDATool, são usados como os valores dos coeficientes aplicados ao Filtro Configurável. Este filtro recebe os coeficientes como parâmetros de entrada e realiza a operação de filtragem. O número máximo de coeficientes que o filtro projetado pode receber é 64, o que, pela simetria, significa que pode-se especificar filtros com respostas impulsivas de até 128 valores.

O Filtro Configurável pode ser utilizado como um filtro passa baixas frequências (PB) ou como um filtro passa altas frequências (PA), como apresentado na seção 3.1. Para o projeto de um filtro passa altas, deve ser especificado, da mesma maneira que o filtro passa baixas, as frequências de passagem e rejeição, e também os *ripples* de passagem e rejeição.

A especificação que mais influência o número de coeficientes no projeto de um filtro é o tamanho da faixa de transição, como já mencionado. Além disso, quanto maior a atenuação na faixa de rejeição e quanto menor a atenuação na banda passante, maior será o número de coeficientes.

A fim de determinar o tamanho máximo do Filtro Configurável aqui implementado, foram projetados diversos filtros no FDATool, e na tabela 6 são resumidos os resultados para o número de coeficientes obtidos. Nota-se que o valor de 64 coeficientes (equivalente a 128 para filtros simétricos) é suficiente para implementar todos os filtros que foram descritos na tabela, que, para aplicações em AAA’s, cobrem perfeitamente as principais situações de uso.

A numeração de cada filtro (primeira coluna da tabela 6) foi utilizada para se fazer referência ao filtro e suas especificações ao se falar das simulações. A partir do filtro 11 têm-se as especificações de filtros passa altas frequências, onde a frequência de rejeição foi definida como zero, indicando que o filtro deve filtrar desde sinais com frequências na ordem de Hz.

Tabela 6: Especificação de filtros FIR passa baixas e passa altas frequências, projetados no FDATool considerando frequência de amostragem de 16 kHz

Filtro FIR	**Frequência (kHz)**		***Ripple* (dB)**		**Número de Coeficientes**
	Passagem-F_{pass}	Rejeição-F_{stop}	Passagem-A_{pass}	Rejeição-A_{stop}	
1	2,0	3,0	0,01	80	
2	3,5	4,0	0,1	80	108
3	3,5	4,0	0,01	75	126
4	4,0	5,0	0,01	80	66
5	4,0	6,0	0,01	80	33
6	4,0	8,0	0,01	60	19
7	4,5	5,0	0,01	70	126
8	6,0	7,0	0,01	80	66
9	6,0	8,0	0,01	70	33
10	7,0	8,0	0,01	50	76
11	0,5	0,0	0,1	80	110
12	0,6	0,0	0,01	80	110

Pode ser observado que os filtros com frequência de corte próxima ao limite da frequência de Nyquist (8,0 kHz), não conseguem ter uma atenuação igual a 80 dB.

4.1.2 Sinais Aplicados e Respostas

Com o intuito de comprovar a funcionalidade do código implementado, foi escrito em Verilog um *testbench* que simula o comportamento do filtro operando com um DSP. O filtro recebe como parâmetros os coeficientes, o número de canais e o seu tamanho. Para cada nova

amostra que o DSP recebe, ele envia o sinal de *sample_in* para que o filtro inicie sua operação. Desta forma, utilizando-se o *testbench* desenvolvido, foram aplicados sinais senoidais criados. Para todas as simulações do Filtro Configurável, a taxa de amostragem dos sinais de entrada foi de 16,0 kHz e as operações do coprocessador foram realizadas a frequência de *clock* de 1,0 MHz.

Foram aplicados no mesmo filtro dois sinais com frequências diferentes, um sinal em cada uma das suas entradas: no canal 1 um sinal dentro da banda passante e no canal 2 um sinal na banda de rejeição. A primeira simulação realizada foi em um filtro com as especificações do filtro 3 da tabela 6. Foi aplicado um sinal de 3,5 kHz no canal 1 e no canal 2 um sinal de 4,0 kHz. Na figura 32 são mostrados os sinais de entrada e de saída de cada canal.

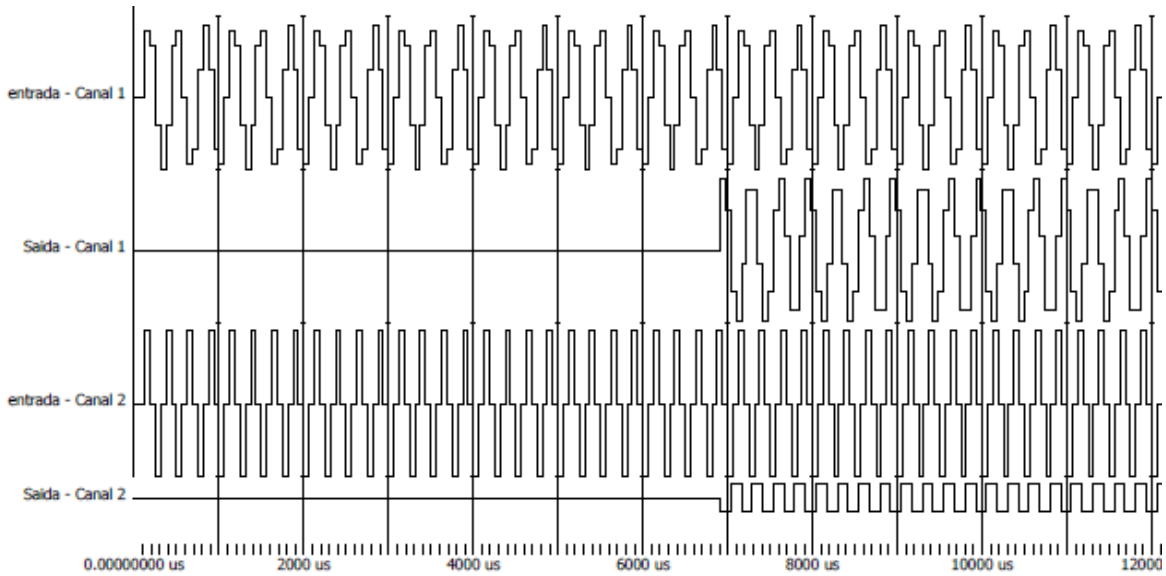

Figura 32: Simulação de dois sinais entrando no filtro (Modelsim). Sinal do canal 1 de 3,5 kHz e do canal 2 de 4,0 kHz

Na tabela 7 são mostrados os valores máximos e mínimos de cada sinal. O cálculo da atenuação é realizado na próxima seção.

Tabela 7: Valores máximos e mínimos dos sinais do canal 1 e canal 2 aplicados ao filtro 3 da tabela 6

Sinal	**Valor de Entrada**		**Valor de Saída**	
	Máximo	**Mínimo**	**Máximo**	**Mínimo**
Input_ch1- Canal 1	$5{,}5364 \times 10^6$	$-5{,}5364 \times 10^6$	$5{,}5289 \times 10^6$	$-5{,}5289 \times 10^6$
Input_ch2- Canal 2	$5{,}5364 \times 10^6$	$-5{,}5364 \times 10^6$	908	- 782

A segunda simulação realizada foi em um filtro com as especificações do filtro 8 da tabela 6. Foi aplicado no canal 1 um sinal de apenas 200 Hz e no canal 2 um sinal de 7,0 kHz. Na figura 33 são mostrados os sinais de entrada e de saída de cada canal.

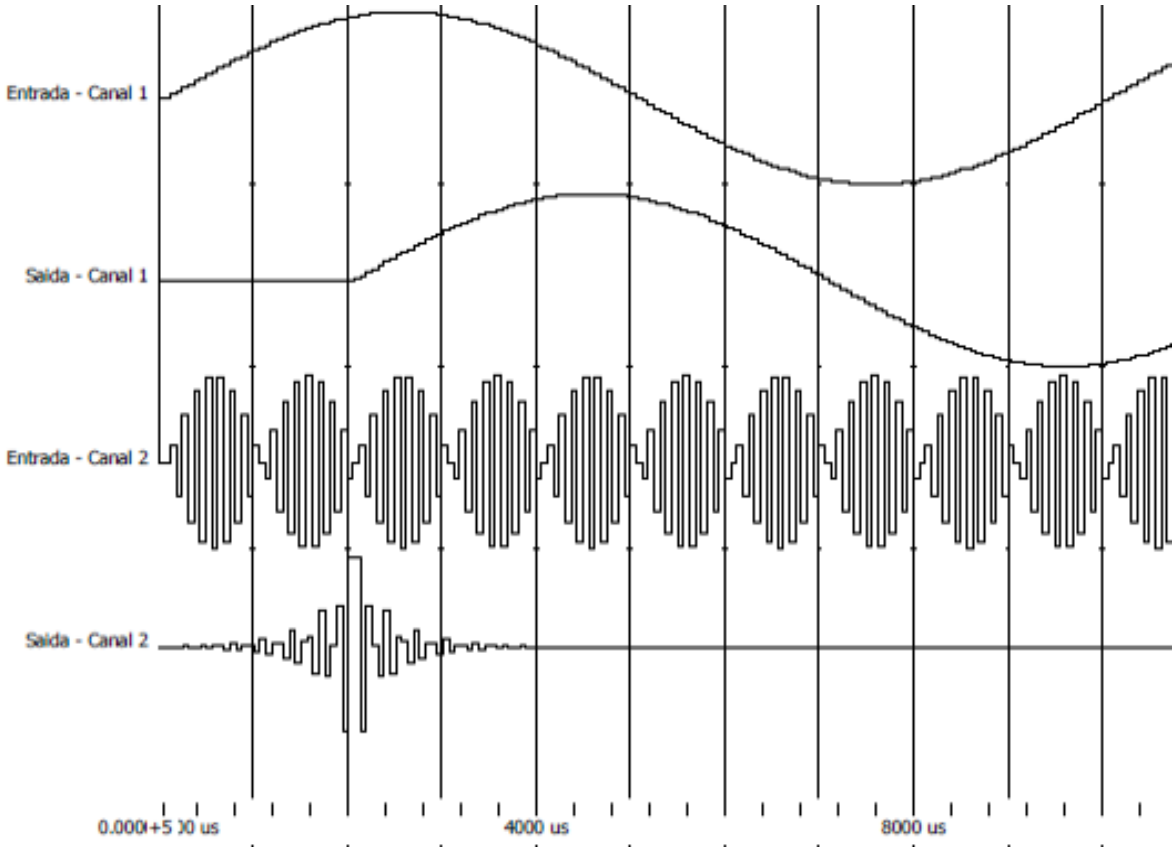

Figura 33: Simulação de sinais entrando no filtro: Sinal do canal 1 de 200 Hz e do canal 2 de 7,0 KHz (Modelsim)

Na tabela 8 são mostrados os valores máximos e mínimos do canal 2. O cálculo da atenuação é realizado na próxima seção.

Tabela 8: Valores máximos e mínimos dos sinais do canal 2 do filtro 8

Sinal	**Valor de**	**Entrada**	**Valor de**	**Saída**
	Máximo	**Mínimo**	**Máximo**	**Mínimo**
Input_ch2- Canal 2	$5{,}5364 \times 10^6$	$-5{,}5364 \times 10^6$	874	- 748

4.1.3 Análise dos Resultados

É possível perceber nas simulações um intervalo de transição no início das respostas dos filtros, sendo de aproximadamente 700 µs para a implementação do filtro 3 e de aproximadamente 400 us para a implementação do filtro 8. Este intervalo de transição se deve ao tempo em que o filtro preenche todas as posições da memória circular com valores de amostra válidos. Como o filtro 3 tem um número maior de coeficientes (126 contra 66), ele demora mais para iniciar o regime correto de operação, pois sua memória circular precisa de mais amostras para ser completamente preenchida.

A partir dos valores da tabela 7 é possível calcular os valores de atenuação da banda passante e da banda de rejeição. Os valores calculados em dB são:

Atenuação na Banda Passante: 0,01 dB

Atenuação na Banda de Rejeição: 75 dB

A partir dos valores da tabela 8 é possível calcular o valor de atenuação da banda de rejeição do filtro 8. Observe que neste filtro, o valor da atenuação do sinal na banda de rejeição não atende a especificação de 80 dB. O valor calculado em dB é:

Atenuação Banda de Rejeição: 76 dB

A impossibilidade de se atingir a especificação é causada principalmente por fontes de erro na implementação do filtro: o número de *bits* da representação dos coeficientes e o truncamento do valor final do filtro. Os coeficientes do filtro são obtidos pelo Matlab com uma precisão de 32 *bits*, porém, para não exceder consumo e área, a representação dos coeficientes foi reduzida para 16 *bits*. Esta redução impossibilita que o coprocessador tenha a resposta ideal, como pode ser obervado na figura 34, que demonstra que a atenuação do filtro 8, com os coeficientes representados com 16 *bits*, está em torno de 77 dB na frequência de 7 kHz.

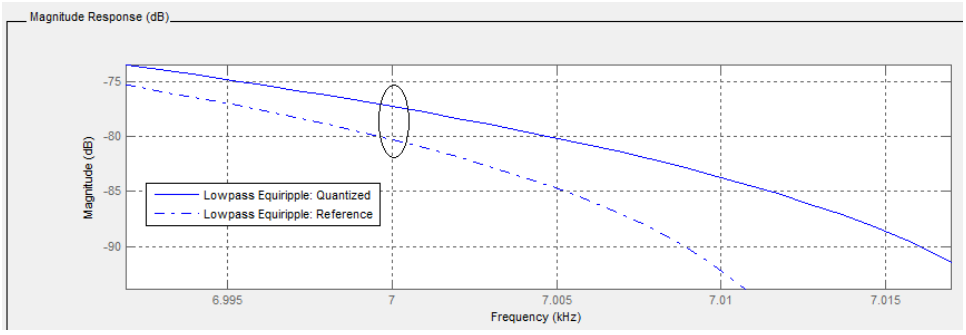

Figura 34: Diferença da resposta em frequência do filtro 8 com a representação dos coeficientes com 32 *bits* (linha pontilhada) e 16 *bits* (linha contínua)

Além da redução do tamanho da palavra para representação dos coeficientes, o crescimento de *bits*, devido às operações de soma e multiplicação, faz com que no interior do filtro o sinal tenha mais de 24 *bits* de informação. Dessa forma, quando é realizado o truncamento, informação valida será eliminada.

O erro na implementação do filtro se mostrou maior nos filtros com as especificações próximas ao limite da frequência de Nyquist, como o caso do filtro 8. Observe que o filtro 3, o qual tem sua frequência de corte bem distante da frequência de Nyquist, 8,0 kHz, conseguiu atender bem as especificações de atenuação.

4.2 Filtro ALE

Assim como no Filtro Configurável, as simulações do filtro ALE tiveram como estímulos os sinais criados a partir do *script* do apêndice C. No caso do filtro ALE, além de sinais senoidais, foram utilizados sinais criados a partir do Aledemo (o Xi, que é uma parte da música "Aleluia" de Handel, e o Xin, que é a composição do Xi e um ruído senoidal de 1,0 kHz). Foram realizadas duas simulações diferentes no filtro ALE, uma utilizando os sinais Xi e Xin como sinais de estímulo e outra utilizando sinais senoidais puros.

Para o Filtro ALE, a análise do número de ciclos de *clock* necessários para que o filtro realize todas as operações é independente do número de coeficientes, pois a sua arquitetura é paralela e todas as multiplicações são realizadas em apenas um ciclo de *clock*. Para a completa operação do filtro ALE são necessários 11 ciclos de *clock*.

4.2.1 Sinais Aplicados e Respostas

Com o intuito de comprovar a funcionalidade do código implementado, foi escrito em Verilog um *testbench* que simula o comportamento do filtro ALE operando com um DSP. O filtro recebe como parâmetros o valor de μ e a direção de saída da resposta do filtro ALE. Para cada nova amostra que o DSP gera, ele envia o sinal de *sample_in* para que o Filtro ALE inicie sua operação. Diferente do Filtro Configurável, o Filtro ALE não recebe os coeficientes como parâmetros de entrada, sendo que estes são calculados pelo próprio filtro durante sua

operação. Os valores das amostras dos sinais foram enviados ao filtro sob uma taxa de 16,0 kHz e as operações do coprocessador foram realizadas a frequência de *clock* de 1,0 MHz.

Para o primeiro teste do Filtro ALE, foram instanciados no *testbench* dois Filtros ALE, para que fosse possível a comparação do ALE tendo um sinal puro, Xi, e um sinal com ruído, Xin, aplicados à entrada. Na instancia chamada de instancia 1 foi aplicado o sinal Xi, para que ele fosse processado pelo filtro e assim sofresse os efeitos de atraso e de distorção. Na instancia chamada de instancia 2, foi aplicado o sinal Xin, no qual será observada a atenuação do sinal senoidal inserido como ruído. Os sinais Xi e Xin são mostrados na figura 35.

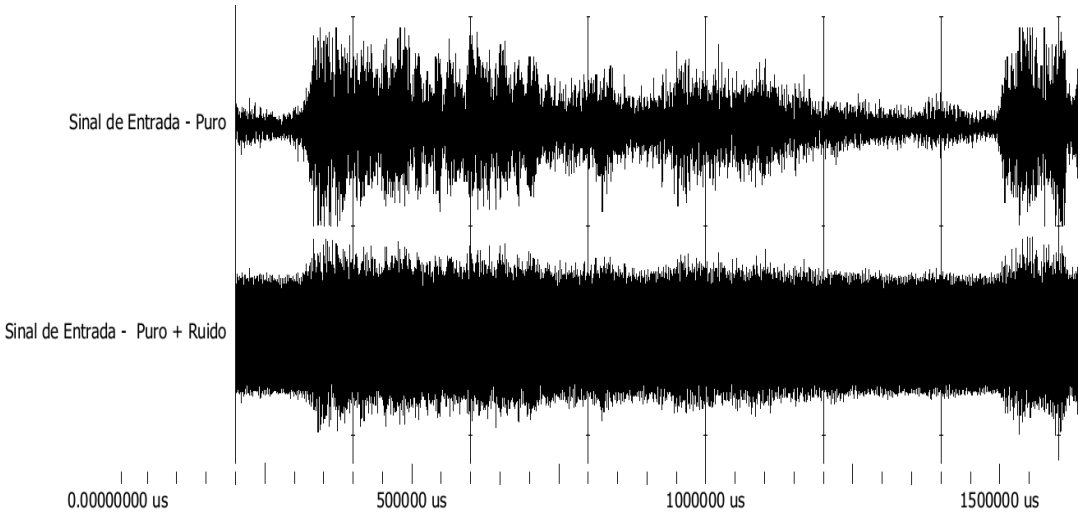

Figura 35: Sinais de entrada dos Filtros ALE. Sinal Xi criado pelo Aledemo (puro) e Xin (puro + ruido) (Modelsim)

Os sinais de saída de cada instancia do *testbench* são mostrados na figura 36.

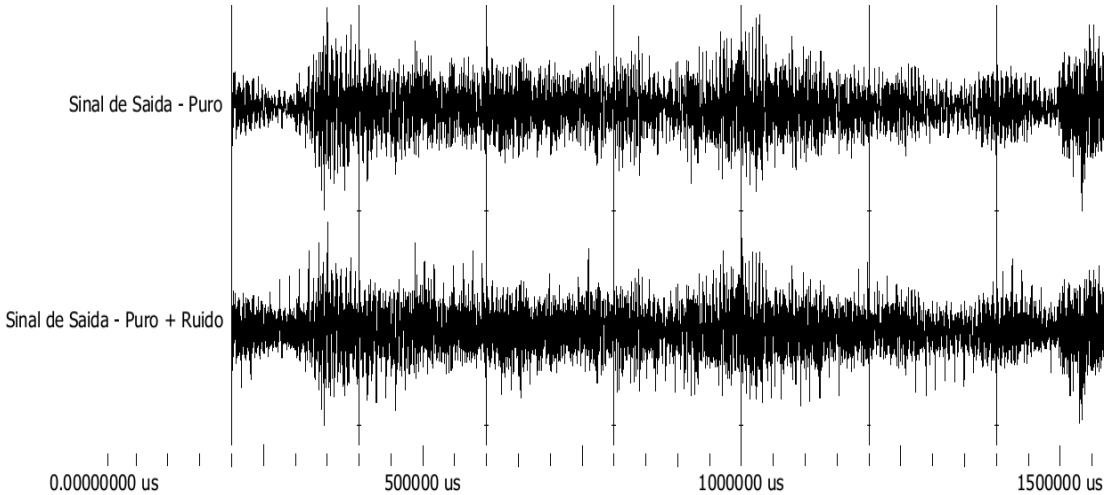

Figura 36: Sinais de saída dos Filtros ALE (Modelsim)

A tabela 9 contêm os valores dos sinais de entrada e saída dos filtros ALE. Os valores da senóide de 1,0 kHz, criada para ser o ruído somado ao sinal Xi, estão entre $4{,}194X10^6$ e $-4{,}194X10^6$. Observa-se que o sinal com ruído (Xin) tem, aproximadamente, valores dobrados se comparados aos valores do sinal puro (Xi).

Tabela 9: Valores máximos e mínimos dos sinais Xi e Xin

Sinal	**Valor de Entrada**		**Valor deSaída**	
	Máximo	**Mínimo**	**Máximo**	**Mínimo**
Instancia 1 - Xi	$3{,}355x10^6$	$-3{,}355x10^6$	$3{,}160x10^6$	$-3{,}188x10^6$
Instancia 2 - Xin	$7{,}432x10^6$	$-7{,}518x10^6$	$3{,}890\ x10^6$	$-3{,}866\ x10^6$

Como os sinais de saída das duas instancias sofreram os mesmos atrasos, devido às operações de filtragem, é possível a comparação ponto a ponto entre eles. Desta forma, foi gerado o sinal *diference* que é a saída da instancia 2 menos a saída da instancia 1. Pode-se considerar esse sinal como o sendo o sinal de erro entre a resposta desejada e a resposta obtida. Na figura 37 é mostrado parte deste sinal, que foi ampliada para melhor observação.

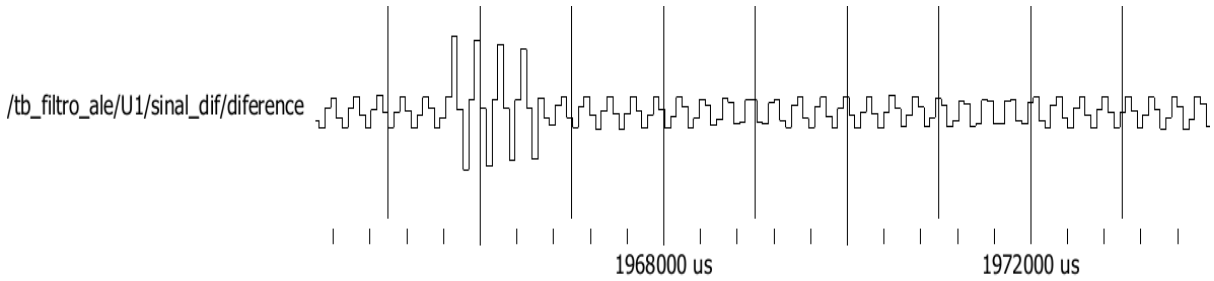

Figura 37: Diferença entre as saídas das instancias do Filtro ALE (Modelsim)

Pode ser observado que a diferença entre as saídas das duas instancias é uma senóide muito pequena, o que está de acordo com a resposta esperada, uma vez que de fato a diferença entre Xi e Xin é um sinal senoidal. Porém, de forma não esperada, foram observados intervalos de instabilidade nas saídas dos Filtros ALE, o qual gerou valores amplificados (ou não atenuados) do sinal de erro. Tais instabilidades foram também observadas na segunda simulação.

Os valores máximos e mínimos do sinal *diference* foram de 730 e de -678 para os instantes de instabilidade. Os valores da senóide propriamente dita, que foi atenuada, ficaram entre 200 e -200. O calculo da atenuação do filtro ALE é mostrado na próxima seção.

Para o segundo teste do filtro ALE, é aplicada de uma senóide composta. O intuito deste teste é mostrar que o filtro ALE atenua mais de um sinal periódico ao mesmo tempo. Foi utilizado apenas uma instancia do filtro ALE. O sinal composto é formado de uma senóide de 2,0 kHz e outra de 3,0 kHz. O apêndice C contem o *script* que gerou este sinal. Na figura 38 são apresentados os sinais de entrada e saída do filtro ALE.

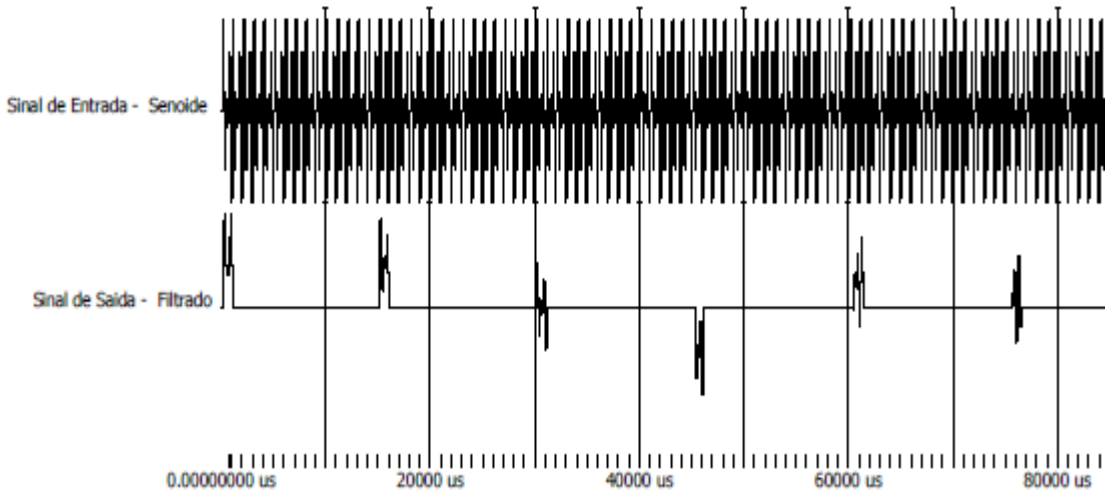

Figura 38: Simulação do Filtro ALE com um sinal senoidal composto (Modelsim)

Nesta figura novamente observa-se os intervalos de instabilidade na saída do filtro, que se mostram periódicos e sempre com a mesma duração. Na tabela 10 são mostrados os valores máximos e mínimos para este segundo teste do filtro ALE.

Tabela 10: Valores máximos e mínimos da saída do Filtro ALE

Sinal	**Valor de Entrada**		**Valor de Saída**	
	Máximo	**Mínimo**	**Máximo**	**Mínimo**
Senóide	$4{,}725 \times 10^{6}$	$-4{,}725 \times 10^{6}$	870	- 850

Pode ser observado que as senóides foram muito atenuadas, sendo que os valores de saída, sem considerar as instabilidades, ficaram novamente entre 200 e -200.

4.2.2 Análise dos Resultados

Foram realizadas diversas simulações com sinais de entrada senoidais com diferentes frequências, variando na ordem de kHz, e constatou-se que a frequência e duração dos intervalos de instabilidade foram sempre as mesmas.

Diversas causas podem ter afetado o comportamento do filtro para causar os momentos de instabilidade verificados. As mais prováveis são:

- o truncamento do sinal de saída e truncamento dos pesos de atualização dos coeficientes (ajustes);

- o atraso aplicado para a autocorrelação, de apenas uma amostra, que pode não ser suficiente para o correto funcionamento do Filtro ALE.

- o valor de μ utilizado não foi meticulosamente analisado, o que pode ter impactado de forma negativa na funcionalidade do filtro. Foi visto em Haykin (2002) que tipicamente o valor de μ está entre $1/2^7$ e $1/2^4$. Para as simulações aqui realizadas escolheu-se o valor de $1/2^5$.

Porém, mesmo com as instabilidades e dificuldades para a correta configuração dos parâmetros do filtro ALE, as suas respostas foram satisfatórias ao se considerar a pequena atenuação no sinal de interesse e a grande atenuação do ruído senoidal.

A partir dos valores da tabela 9 é possível calcular a atenuação que o filtro causa em Xi:

Atenuação no Sinal de Interesse (Xi): 0,5 dB

Pela tabela 10, com os valores do sinal senoidal, é possível calcular a atenuação que o filtro causa no sinal periódico, que mostrou ser alto o suficiente para eliminar ruídos senoidais.

Atenuação no Sinal Periódico: 87 dB

Ao se comparar o valor de atenuação obtido pela implementação em Verilog com o valor obtido pelo algoritmo do filtro ALE implementado em Matlab, cujo valor foi de aproximadamente 100 dB, comprova-se que o funcionamento do filtro descrito em Verilog se aproximou da implementação ideal.

4.3 Síntese dos coprocessadores - *backend*

O fluxo comum para um projeto ASIC pode ser dividido em duas etapas: a etapa de especificação e implementação de códigos, conhecido como *frontend*, e a etapa que consiste em passar os códigos implementados para o *hardware*, conhecido como *backend.* Algumas restrições do projeto, principalmente as de área e consumo, são cumpridas na etapa de *backend* enquanto a funcionalidade tem de estar garantida já na etapa de *frontend.*

A etapa de *backend* realizada para os dois coprocessadores implementados foi dividida em duas partes, a primeira realizada pelo RTL Compiler e a segunda pelo SoC Encounter.

4.3.1 RTL Compiler

O RTL Compiler tem como objetivo fazer o mapeamento do código RTL (Verilog comportamental) para o *hardware* através de uma biblioteca de células digitais, criando um Verilog estrutural, chamado de *netlist.* É também no RTL compiler que se aplicam comandos de restrição (*constraints*), definidas por um *script* de síntese, para se obter diferentes resultados de consumo, área, número de células, etc. Por exemplo, é neste *script* que se define se o coprocessador terá ou não *clock-gating*[7], o que causa grande impacto no consumo de potência, com um pequeno acréscimo de área. O RTL compiler também cria o arquivo de definição da árvore de *clock* que será utilizada no *hardware* sintetizado.

Inicialmente, realizou-se a síntese independente de cada coprocessador, primeiro um Filtro Configurável e depois um Filtro ALE. Para ambos, utilizou-se o mesmo *script* de síntese. Na tabela 11 são mostrados os valores obtidos para o número de células de cada coprocessador.

Tabela 11: Resultados das Sínteses

Coprocessador	Número de células digitais
Filtro Configurável	37642
Filtro ALE	69099

[7] *Clock-gating* é a desativação de parte da árvore de *clock*, usada por um conjunto de *flip-flops*, para reduzir o consumo de potência. Pode ser realizado pela adição de uma porta AND com um sinal de habilita em série com as árvores de *clock* (Frank Emnett e Biegel, 2000).

Ao se somar o número de células, tem se o total de 106741 células digitais.

Contudo, como os coprocessadores fazem parte do mesmo sistema (SoC), é uma prática melhor e mais eficiente para o *backend* realizar a compilação dos códigos Verilog para um único *netlist*. Assim sendo, utilizando o *script* de síntese que gerou os resultados anteriores, porém agora colocando os códigos Verilog dos dois coprocessadores (Filtro Configurável e Filtro ALE), realizou-se a síntese, gerando o *netlist* chamado de Top_filtro.

O número total de células deste *netlist* foi de 73814, uma redução da ordem de 30% do número de células ao se comparar com as sínteses separadas.

A figura 39 é apenas uma representação de como ficaram as ligações entre os dois coprocessadores (Filtro configurável e Filtro ALE), mostrando que uma das entradas de dados foi compartilhada.

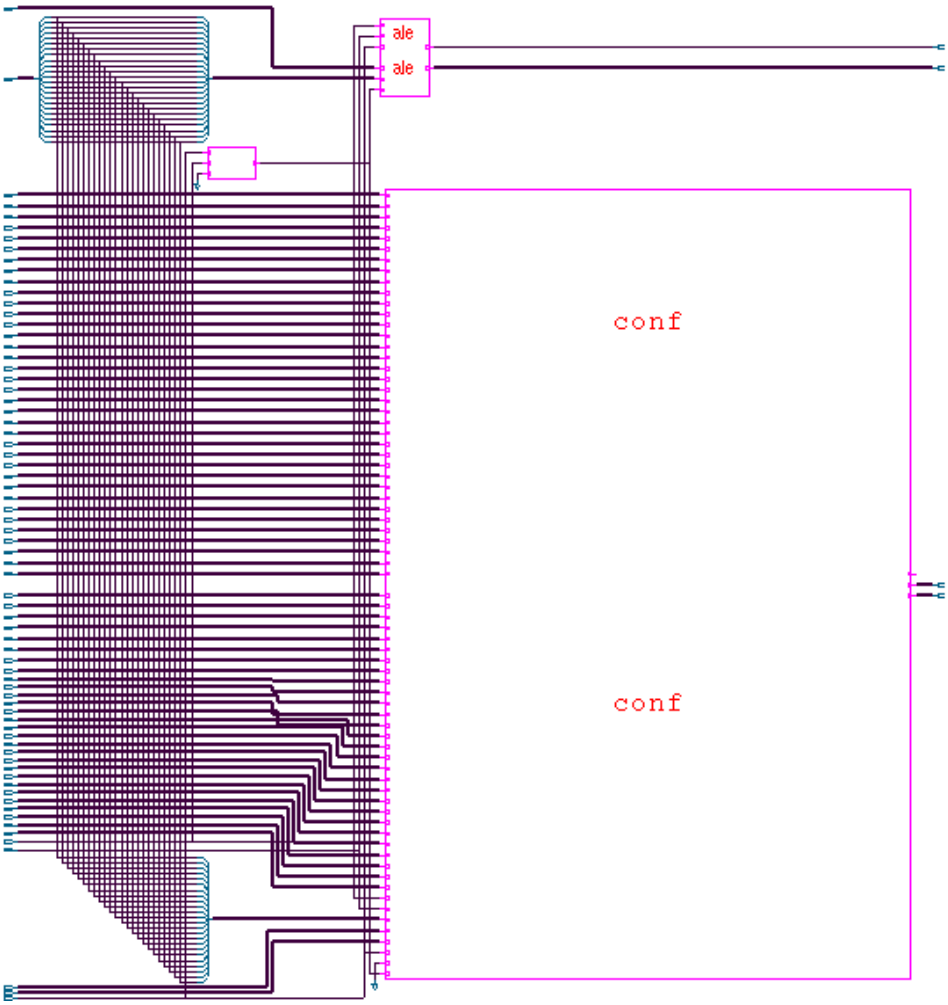

Figura 39: Diagrama gerado da síntese do Filtro Configurável e do Filtro ALE (RLT Compiler)

Na figura 40 é mostrado o esquemático gerado pelo RTL Compiler da máquina de estado do filtro ALE. Nela é possível observar os 8 registradores de saída (ff1 até ff8) que controlam as operações do filtro.

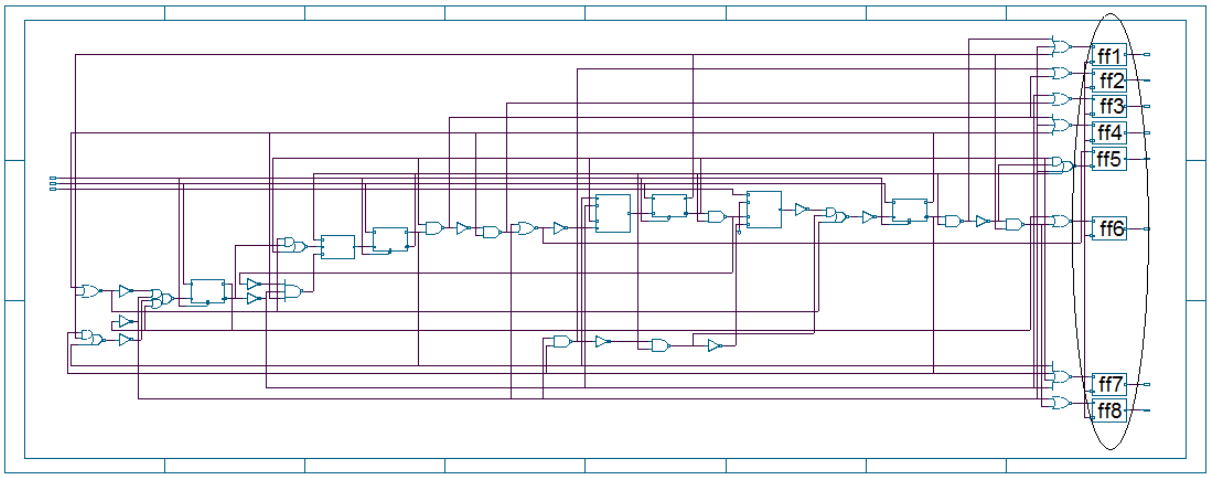

Figura 40: Esquemático da máquina de estado do Filtro ALE (RTL compiler)

4.3.2 SoC Encounter

Uma vez obtido o *netlist* Top_filtro, com a síntese dos dois coprocessadores implementados em Verilog, a segunda parte do *backend* é o posicionamento e o roteamento das células, o chamado *place&route*, realizado pelo SoC Encounter. Além disso, o SoC Encounter realiza a criação dos anéis de alimentação e da árvore de *clock* a partir de arquivos gerados pelo RTL Compiler.

É possível, após cada passo realizado no SoC Encounter, verificar se o *layout* está correto, analisando se há problema de *overlap*, se há curtos circuitos entre as *nets*, etc. O *overlap*, que é a sobreposição de células, ocorre na etapa de posicionamento das células *(place)*. O *overlap* é comum quando se define uma área pequena para o *layout*, muitas vezes por não se considerar a área de roteamento para a alimentação das células (vdd e gnd) e para as ligações com os contatos das células.

Após a etapa de posicionamento das células do *netlist* Top_filtro, foi gerada a figura 41, que mostra a disposição dos coprocessadores sintetizados, possibilitando reconhecer a área do Filtro Configurável e do Filtro ALE.

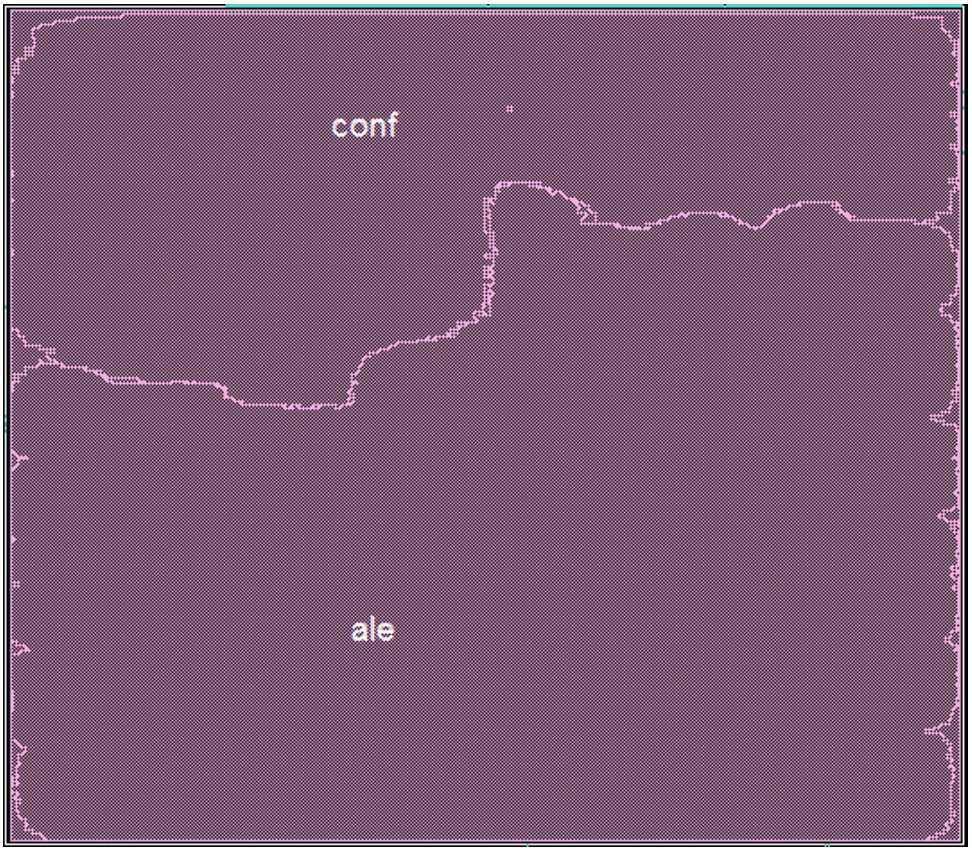

Figura 41: Posicionamento das células do *netlist* Top_filtro (Soc Encounter)

Pode-se constatar que a área do Filtro ALE (ale) é aproximadamente o dobro da área do Filtro Configurável (conf), como era esperado pelo resultado do número de células, apresentado na tabela 11. A razão do maior número de células é o maior número de multiplicadores presentes no Filtro ALE.

Na figura 42 é mostrado parte das células posicionadas do *netlist* Top_filtro. Foi selecionada a vista *layout* de cada célula para ser possível observar detalhes tais como o espaçamento que existe entre as células, o que será discutido a seguir.

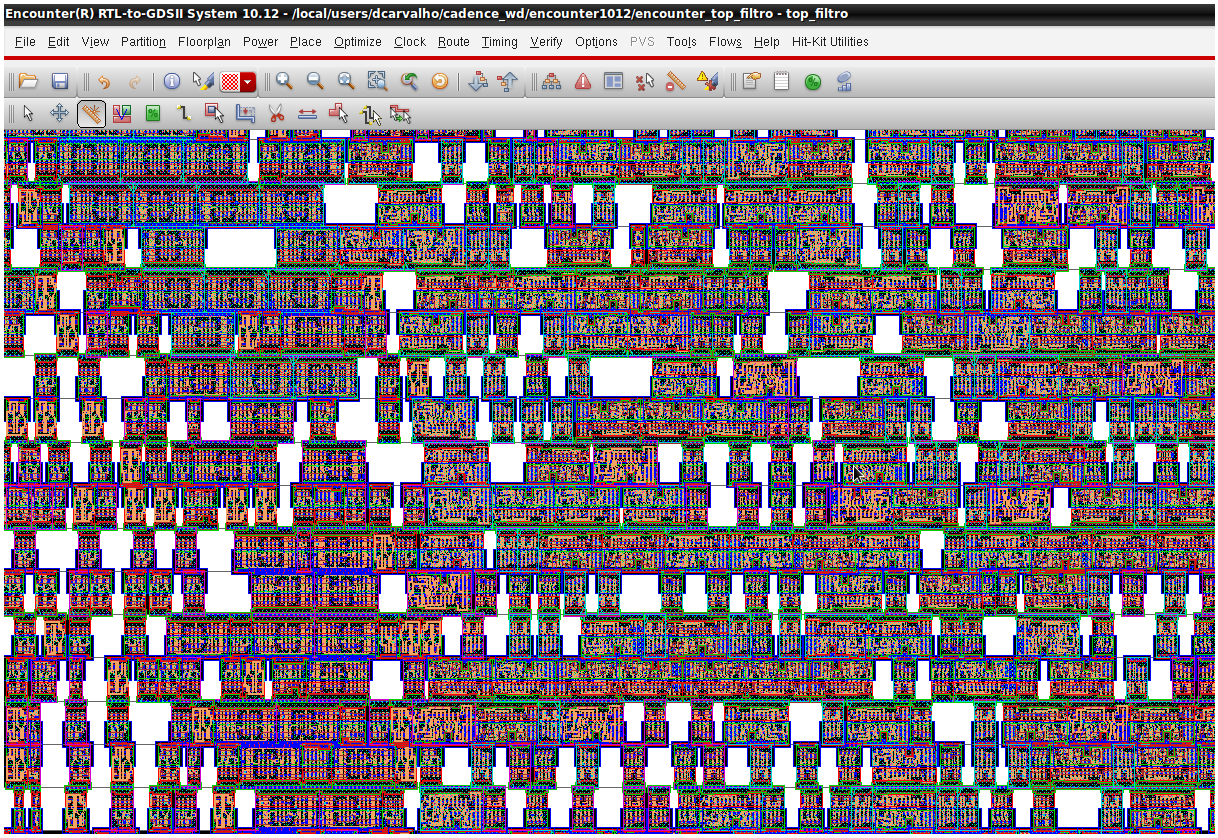

Figura 42: Células posicionadas para o roteamento (SoC Encounter)

O espaçamento visto entre algumas células na figura 42 foi proposital, pois se definiu uma área consideravelmente maior para o *layout* que a área ocupada somente pelas células. A razão disso é que estão disponíveis apenas apenas três níveis de metal para o roteamento. O metal um foi utilizado para a alimentação das células (vdd e gnd), para a árvore de *clock* e para os contatos das células Com isso restaram apenas dois níveis de metal para o roteamento propriamente dito, sendo o metal dois utilizado para roteamento horizontal e metal três, para roteamento vertical. Em consequência não é possível se obter maior densidade de células sem que haja curtos circuitos nas *nets* de roteamento.

O resultado final é um circuito com área de 14 mm² mostrado na figura 43. O metal três, representado na cor verde, é o metal mais alto, por isso a figura ficou apenas como um “quadrado verde”. Foi inserido um pedaço da figura 42 apenas para mostrar que abaixo destes metais de roteamento, estão as células digitais.

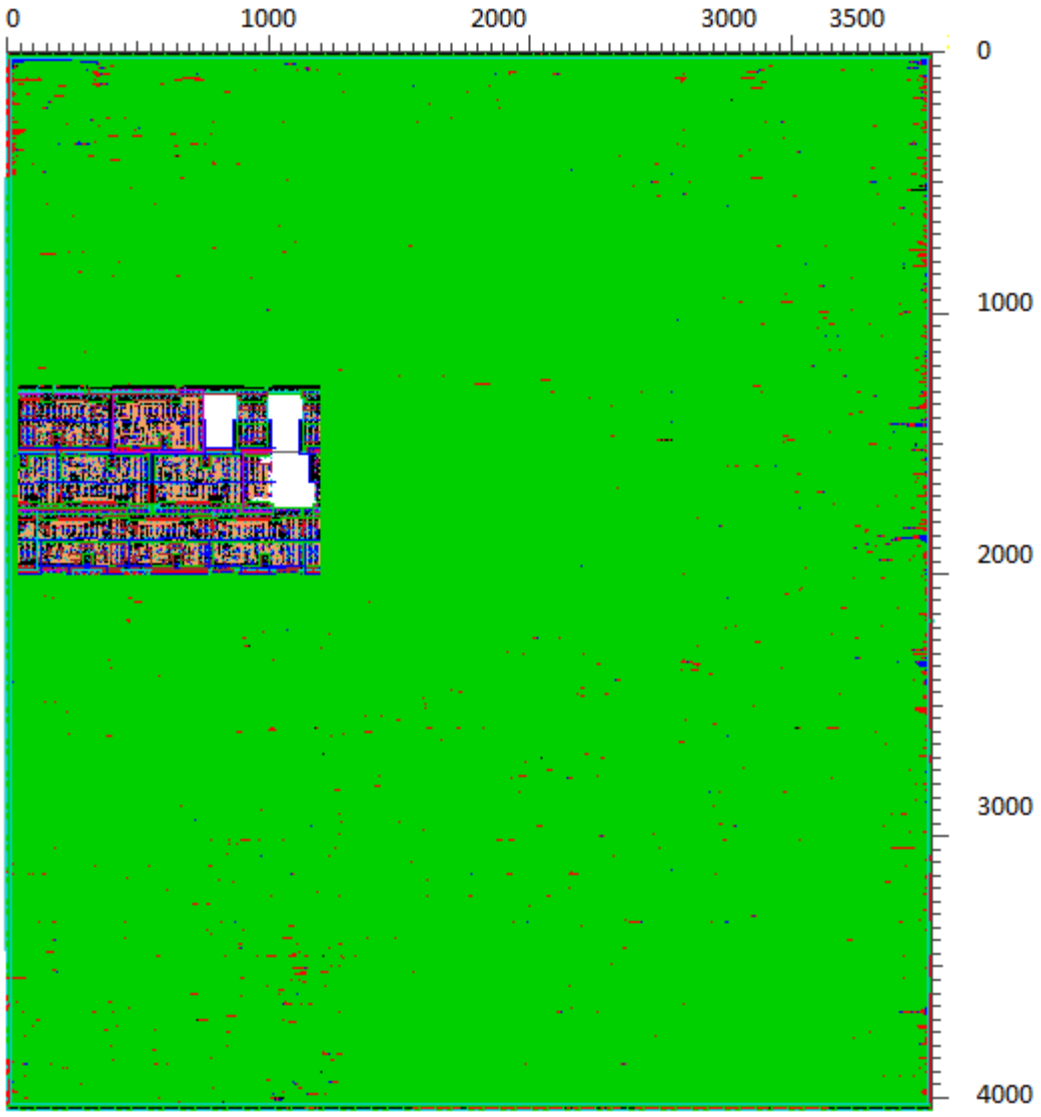

Figura 43: *Layout* final dos coprocessadores, Filtro Configurável e Filtro ALE (SoC Encounter)

Como mencionado na seção 2.3, a dimensão desejada para um CI que implementa um AAA é de cerca de 20 mm^2 e, baseado neste valor, este *layout* excede o tamanho razoável para coprocessadores que sejam parte de um AAA.

Uma das razões para a grande área é que a tecnologia utilizada para a síntese foi de 0,35µm, que não é uma tecnologia moderna. Outra razão é que o roteamento com três níveis de metal não é suficiente para se conseguir um circuito altamente integrado. Estima-se que aproximadamente 30% da área deste *layout* é preenchida com *fillers*. *Fillers* são células com os contatos de alimentação (vdd e gnd), porém sem nenhuma lógica. São inseridas para preencher os espaços vazios entre células usados para *roteamento*, dando continuidade nas *nets* de alimentação.

Para redução da área, deve-se realizar uma implementação otimizada em área para os filtros. Uma alternativa para isso, por exemplo, é a utilização de apenas um multiplicador para realizar todas as operações de multiplicação do Filtro ALE. Assim, em vez de se ter 24 multiplicadores de 39 *bits*, ter-se-ia apenas um. Porém, o consumo do Filtro ALE seria maior.

4.3.3 Análise do consumo de potência

Para finalizar a análise do *layout*, foram feitas estimativas de consumo de potência para diferentes frequências do *clock* e com atividade dos sinais de 0,1. O valor da atividade está relacionado com a taxa de mudança dos valores dos sinais de entrada e dos sinais internos do coprocessador, como a ativação dos sinais de controle dos *clock gating* e ativação dos *flip-flops*. A tabela 12 mostra os valores obtidos.

Tabela 12: Estimativa de consumo de potência para diferentes frequências de *clock* aplicados aos coprocessadores implementados

Clock	300 kHz	600 kHz	1,0 MHz	5,0 MHz	
Atividade					
0,1	**0,30**	**0,61**	**1,01**	**5,06**	mW

Esta estimativa de consumo realizada pelo SoC Encounter leva em consideração o consumo interno de cada célula (*Internal Power*), o consumo de *leakage* (*Leakage Power*) e o consumo associado às capacitâncias das *net*s (*Switching Power*), sendo que o consumo total é a soma destas três componentes para todas as células. No consumo é considerado um Filtro Configurável e um Filtro ALE operando simultaneamente. Pela tabela 12 está evidente que o consumo está diretamente relacionado com a frequência do *clock* de operação do coprocessador.

Com a busca de sistemas de menor consumo de potência e tendo em mente a relação entre consumo de potência e frequência do *clock* de operação dos coprocessadores, é interessante que o sistema apresentado em 3.1 seja configurado em *pipeline* com seis estágios, sendo cada estágio responsável por uma etapa do processamento de áudio. Na figura 44 é mostrado os seis estágios propostos para o SoC. Com a utilização de *pipeline*, é possível aplicar *clocks* com menor frequência nas etapas de processamento que requeiram reduzido

número de ciclos de *clock*, como será mostrado a seguir. O *pipeline* é adequado também porque cada coprocessador contém memória local para o armazenamento das amostras.

Os estágios do *pipeline* são: o codec de entrada, o Filtro Configurável passa baixas (PB), o DSP, o Filtro ALE, o Filtro Configurável passa altas (PA) e o Codec de Saída.

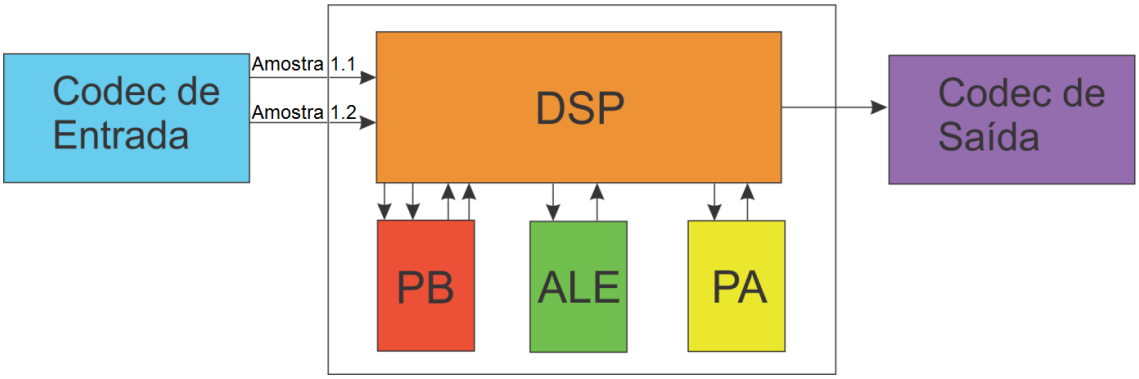

Figura 44: Proposta de *Pipeline* para processamento em um AAA

A tabela 13 mostra como será o fluxo das amostras no SoC para o processamento em um AAA.

Tabela 13: Fluxo das amostras no *Pipeline* para processamento em um AAA. Os estágios de codec, PB e DSP trabalham com um conjunto de duas amostras, uma de cada canal de entrada

Tempo	**1**	**2**	**3**	**4**	**5**	**6**
Codec de Entrada	**Amostra 1.1** **Amostra 1.2**	**Amostra 2.1** **Amostra 2.2**	**Amostra 3.1** **Amostra 3.2**	**Amostra 4.1** **Amostra 4.2**	**Amostra 5.1** **Amostra 5.2**	**Amostra 6.1** **Amostra 6.2**
PB		**Amostra 1.1** **Amostra 1.2**	**Amostra 2.1** **Amostra 2.2**	**Amostra 3.1** **Amostra 3.2**	**Amostra 4.1** **Amostra 4.2**	**Amostra 5.1** **Amostra 5.2**
DSP			**Amostra 1.1** **Amostra 1.2**	**Amostra 2.1** **Amostra 2.2**	**Amostra 3.1** **Amostra 3.2**	**Amostra 4.1** **Amostra 4.2**
ALE				**Amostra 1**	**Amostra 2**	**Amostra 3**
PA					**Amostra 1**	**Amostra 2**
Codec de Saída						**Amostra 1**

Observa-se que após o estágio de processamento no DSP, há apenas o processamento de uma única amostra. Isto é porque o DSP realiza a operação de microfone direcional e tal operação tem como resultado apenas uma amostra.

Segundo Herbig (2009), o tempo de atraso no processamento de um AAA deve ser inferior a 5,0 ms. Com o codec de entrada realizando a amostragem com uma frequência de 16,0 kHz, tem-se uma nova amostra a cada 62,5 µs. Com os seis estágios de processamento

(Figura 44), serão necessários 6 amostras para preencher o *pipeline* e gerar uma amostra de saída, o que demanda apenas 0,375 ms (6x62,5 μs). Este tempo de preenchimento não será perceptível pelo usuário do AAA, visto é inferior ao limite permitido Após o preenchimento dos estágios de processamento do *pipeline*, será gerada uma amostra de saída para cada nova amostra de entrada. Considerando ainda o tempo requerido para que o Filtro Configurável entre em regime de operação, pode-se afirmar que o atraso total não será perceptível pelo usuário do aparelho.

Com a utilização do Filtro ALE como um estágio de *pipeline* e recebendo amostras com frequência de 16 kHz, é possível a sua implementação utilizando apenas um multiplicador, o que possibilitaria a redução de sua área. Na implementação com apenas um multiplicador, o filtro ALE requereria no máximo 38 ciclos de *clock* para processar cada amostra. Com a utilização do *clock* de 600 kHz, o filtro ALE realizaria todas as operações dentro dos 62,5 μs.

Considerando as duas opções de configuração para o sistema apresentado em 3.1, com ou sem *pipeline*, temos os seguintes cenários:

- Sem *Pipeline* :

Com o codec de entrada trabalhando com 16,0 kHz, todo o sistema de processamento digital tem no máximo 62,5 μs para executar todos os algoritmos para correção do sinal auditivo do AAA. Considerando o *clock* de 5,0 MH, o Filtro ALE requereria 2,2 μs e o Filtro Configurável requereria 14,0 μs, para o pior caso de um FIR de 128 coeficientes. Considerando dois Filtros Configuráveis, a soma dos tempos de processamento dos coprocessadores seria de 30,2 μs, quase 50 % de todo tempo disponível. É um tempo alto se considerarmos que o DSP ainda tem que executar outros algoritmos como microfone direcional, cancelamento de realimentação acústica, etc..

Considerando o DSP Coolflux, podemos estimar a potência consumida pelo processamento digital (DSP + coprocessadores). Para isso faremos algumas hipóteses:

- que o DSP Coolflux tenha capacidade para realizar todas as operações de processamento, exceção daquelas realizadas pelos coprocessadores, no intervalo de 30 μs trabalhando em 5,0 MHz. Neste caso seu consumo é de 0,4 mW para tecnologia de 130 nm e tensão de alimentação de 1,2 V (Tabela 4, seção 2.2.4).

- que a estimativa de potência dos coprocessadores realizada com atividade de 0,1 e *clock* 5,0 MHz, 5,06 mW, é maior que a potência real.

- que os coprocessadores sejam implementados numa tecnologia mais moderna que tenha, como o ColFlux, tensão de alimentação de 1,2 V. Nesse caso a potencia consumida pode ser, em primeira ordem, estimada em 2,25 mW ($5{,}06*(1{,}2/1{,}8)^2$).

- que o Filtro Configurável tenha um consumo equivalente a do Filtro ALE (pelas dimensões dos circuitos, espera-se que seja de fato menor). Dessa forma podemos considerar que o consumo total dos coprocessadores seria de 3,4 mW

Dado as hipóteses feitas, o valor do consumo de potência de todo o sistema com: dois Filtros FIR (PA e PB), um Filtro ALE e o DSP, será então próximo de 4 mW. Este valor está além do limite de consumo, como visto em 2.3.

- Com *Pipeline* :

Ainda com o codec de entrada trabalhando com 16,0 kHz, porém agora com o sistema configurado como *pipeline*, cada estágio de processamento tem 62,5 μs para executar sua funcionalidade. Com este tempo, podemos utilizar um *clock* de 1,0 MHz nos coprocessadores e assim Filtro ALE requereria 11 μs e o Filtro Configurável requereria 60,0 μs, para uma implementação com 104 coeficientes, para processar cada amostra.

Considerando novamente o DSP Coolflux, podemos novamente estimar a potência consumida pelo processamento digital (DSP + coprocessadores). Para isso, temos as seguintes hipóteses:

- que o DSP Coolflux tenha capacidade para realizar todas as operações de processamento, exceção daquelas realizadas pelos coprocessadores, no intervalo de 60 μs trabalhando em 5,0 MHz. O consumo permanece 0,4 mW para tecnologia de 130 nm e tensão de alimentação de 1,2 V.

- que a estimativa de potência dos coprocessadores realizada com atividade de 0,1 e *clock* 1,0 MHz, 1,01 mW, é maior que a potência real.

- que os coprocessadores sejam implementados numa tecnologia mais moderna que tenha, como o ColFlux, tensão de alimentação de 1,2 V. Nesse caso a potência consumida pode ser estimada em 0,45 mW.

- que o Filtro Configurável tenha um consumo equivalente a do Filtro ALE. Dessa forma podemos considerar que o consumo total dos coprocessadores seria de 0,67 mW

Dados as hipóteses, o valor do consumo de potência de todo o sistema, dois Filtros FIR (PA e PB), um Filtro ALE e o DSP, será então próximo de 1,1 mW.

Com a opção de configuração do sistema 3.1 com *pipeline*, além de apresentar o valor de consumo de potência menor quando comparado com o sistema sem *pipeline*, pode-se afirmar que o Coolflux será capaz de realizar mais rotinas de processamento do AAA, pois tem um tempo maior de processamento comparado a configuração sem *pipeline*. O Coolflux terá aproximadamente 60,0 μs para processar rotinas de um AAA, o que são 300 ciclos do *clock* de 5,0 MHz.

Os resultados das duas configurações (sem e com *pipeline*) considerando as hipóteses mencionadas são resumidos na tabela 14.

Tabela 14: Resultados de área e consumo para o sistema proposto configurado com e sem *pipeline*

Sem Pipeline		Com Pipeline	
Tempo de processamento dos coprocessadores (μs) – *clock* de 5 MHz		Tempo de processamento dos coprocessadores (μs) – *clock* de 1 MHz	
Filtro Configurável	14,0 (128 coeficientes)	Filtro Configurável	60,0 (104 coeficientes)
Filtro ALE	2,2	Filtro ALE	11,0
Total (2 Filtros Configurável e 1 ALE)	30,2	Total (2 Filtros Configurável e 1 ALE)	-
Consumo (mW)– *clock* de 5 MHz		Consumo (mW) – *clock* de 1 MHz	
DSP Coolflux	0,4	DSP Coolflux	0,4
Filtro Configurável + Filtro ALE (Alimentação 1,2V)	2,25	Filtro Configurável + Filtro ALE (Alimentação 1,2V)	0,45
2° Filtro Configurável	1,12	2° Filtro Configurável	0,22
Consumo Total (2 Filtros Configuráveis + 1 Filtro ALE + DSP)	4,0	Consumo Total (2 Filtros Configuráveis + 1 Filtro ALE + DSP)	1,1

CONCLUSÕES E TRABALHOS FUTUROS

Neste capítulo serão apresentadas as conclusões do trabalho realizado e possíveis trabalhos futuros como continuidade deste.

É evidente que a evolução da microeletrônica associada com a constante melhoria dos algoritmos de PDS para os AAA's tem tornado melhor a vida das pessoas com alguma deficiência auditiva, possibilitando maior inclusão social.

5.1 Conclusões

Foi proposto um SoC para ser utilizado como Aparelhos de Auxílio Auditivo (AAA) digital, onde o processamento de sinais digitais foi realizado por um DSP em conjunto com coprocessadores. A especificação do SoC com a utilização de coprocessadores para o processamento *hardwired*, substituindo o processamento por *software* no DSP, foi a solução buscada e trabalhada para fazer um sistema com consumo reduzido e que atendesse a demanda de processamento. Consumo de potência é um dos fatores importantes que afetam as escolhas em projetos de CI's, no geral, e mais ainda em aplicações de aparelhos portáveis. O uso de DSP implementado como um ASIP é a melhor opção para se ter coprocessadores acoplados em busca da otimização do consumo de potência.

No SoC proposto, três algoritmos de tratamento de sinal digital foram propostos para serem processados por *hardwire*, gerando assim coprocessadores. Os três algoritmos foram: um filtro digital FIR, chamado de Filtro Configurável; o cancelador de realimentação acústica; e o filtro para remoção de sinais senoidais, chamado de filtro ALE. Destes três algoritmos, dois foram implementados em Verilog, sintetizados e testados. Os dois coprocessadores implementados foram o Filtro Configurável e o Filtro ALE.

A implementação do Filtro Configurável, coprocessador que permite a configuração de diferentes filtros FIR para o processamento por *hardware*, se mostrou factível e os resultados das simulações mostraram o correto funcionamento do código Verilog desenvolvido. Através da análise dos valores das repostas obtidas a partir do **Modelsim**, constatou-se que a redução do tamanho da palavra para a representação dos valores dos coeficientes tem um impacto no SNR do sinal, reduzindo alguns dBs para alguns casos. Nos

casos em que o filtro projetado tinha suas especificações no domínio da frequência muito próximas dos limites definidos pelo conversor ADC, houve redução do valor da atenuação na banda de rejeição. A frequência de amostragem do ADC define a frequência máxima dos sinais de entrada que podem ser convertidos para o domínio digital

A implementação do filtro ALE, utilizado para eliminar sinais periódicos, se mostrou factível e os resultados das simulações foram satisfatórios. Através da análise dos valores das repostas obtidas a partir do **Modelsim**, constatou-se alto fator de atenuação para os sinais periódicos, chegando a 87 dB. Porém, as simulações apresentaram picos de instabilidade devido a características particulares da implementação em *hardware*, como truncamento e o pequeno valor de atraso para a autocorrelação. Como possível causa das instabilidades, podemos também apontar o valor de μ, o qual não foi meticulosamente estudado.

Foi realizado o fluxo *backend* dos coprocessadores com a tecnologia CMOS 0,35μm da AMS (*Austria Micro Systems*), tensão de alimentação 1,8 V e três níveis de metal. O *layout* digital gerado apresentou consumo de potência dentro da restrição de consumo dos AAA's caso os coprocessadores operem com frequência de *clock* de até 1,0 MHz. Porém, apresentou grande área, chegando a 14,0 mm^2 para a síntese conjunta do Filtro Configurável e do Filtro ALE.

Como razões para a grande área podemos apontar a tecnologia de 0,35μm e também o fato que três níveis de metal para roteamento não são suficientes para alta densidade de integração das células. Caso utilizassemos uma tecnologia mais moderna, como a CMOS 90 nm, a redução da área, apenas pelo resultado da síntese, pode chegar em 75 %, resultando em uma área de 3,5 mm^2. Se considerarmos ainda, a possibilidade de mais de três níveis de metal para o roteamento, podemos obter maior densidade de célula, reduzindo os 30% de área estimada com *fillers*, obtendo áreas menores de 3,5 mm^2. É possível, ainda, a redução da área com otimizações dos códigos Verilog dos coprocessadores, como por exemplo, a implementação do Filtro ALE com apenas um multiplicador.

A proposta de fazer o SoC como um *pipeline* de seis estágios é vista como a melhor solução para um sistema de baixo consumo de potência, como analisado em 4.3.3. Pode-se concluir que o valor estimado de 1,1 mW para o sistema satisfaz a restrição de consumo de potência para um AAA e que este valor está muito próximo ou mesmo menor comparado a outros trabalhos como: (Mosch, Oerle *et al.*, 2000), que obtive o consumo de 1 mW para uma tensão de alimentação de 1,2 V, e (Neuteboom, Kup *et al.*, 1997) que obtive o consumo de 2 mW para tensão de alimentação de 0,9 V.

Uma importante conclusão para otimizar o consumo de potência dos coprocessadores, é a utilização de diferentes frequências de *clock* para os coprocessadores. Para isso, na etapa de *backend*, o projeto da árvore de *clock* deve considerar o projeto de diferentes frequências. Por exemplo, o Filtro ALE pode perfeitamente ter com frequência de *clock* de 300 kHz, enquanto o Filtro Configurável a frequência de 2,0 MHz. Com estas frequências, obter-se-ia um consumo ainda menor para o sistema proposto.

5.2 Trabalhos Futuros

Os trabalhos futuros são listados a seguir:

- Fabricar os coprocessadores implementados e testar o sistema com um DSP real ou um DSP sintetizado em uma FPGA;
- Fazer uma estimativa do consumo de potência requerido por um DSP para executar os algoritmos implementados em *hardware*. Dessa forma é possível comparar, de forma direta, o consumo entre implementações por *software* e *hardware*;
- Investigar melhor o algoritmo do Filtro ALE e tentar resolver os problemas observados;
- Implementar o filtro ALE com apenas um multiplicador.
- Implementar o algoritmo para Cancelamento de Realimentação Acústica em Verilog e realizar o fluxo *backend*;
- Analisar a capacidade de processamento de DSPs e a possibilidade/ necessidade de se ter outras funcionalidades implementadas em *hardware*.

REFERÊNCIAS BIBLIOGRÁFICAS

ALISOM. (2011). **Exame de audiometria**. Disponível em:<**http://www.alisom.com.br/portal/aparelhos-auditivos/exame-de-audiometria**>. Acesso em: 03/06/ 2011.

ANGELINI, C. **Intel core i7-3960x review**: sandy bridge-E and x79 express. Disponível em:<http://www.tomshardware.com/reviews/core-i7-3960x-x79-sandy-bridge-e,3071.html>. Acesso em: 31/07/2013.

ANGOLETTA, M.E. (2008). **Digital signal processor fundamentals and system design**. p.167-229. Disponível em:<http://cds.cern.ch/record/1100536/files/p167.pdf>. Acesso em: 31/07/2013.

AUDIBLE DIFFERENCE. (2012). **Digital hearing aids**. Disponível em:<**http://www.audible-difference.com/Articles/DSP2.htm**>. Acesso em: 31/07/2013.

AVALOS, J.G.; SANCHEZ, J.C.; VELAZQUEZ, J. (2011). Applications adaptive filtering. In: GARCIA,L. (Ed.). **Adaptive filtering applications**. Rijeko: InTech. Cap.1. Disponível em:< http://www.intechopen.com/books/adaptive-filtering-applications/applications-of-adaptive-filtering>. Acesso em: 31/07/2013.

CARVALHO, D.D.; NAVARRO, J. (2013). A Power optimized decimator for sigma-delta data converters. In: IEEE LATIN AMERICAN SYMPOSIUM ON CIRCUITS AND SYSTEMS, 4., 2013. **Proceedings…** Piscataway: IEEE, 2013. **LASCAS**, 2012.

CEITEC S.A. SEMICONDUCTORS. Disponível em:<**http://www.ceitec-sa.com/**>. Acesso em: 03/06/2011.

CSEM. (2008). **IcyFlex4 user's manaul**. [S.l.]: CSEM S.A.

CUBAS, H.G.A. (2013). **Projeto de um modulador sigma-delta de baixo consumo para sinais de áudio**. Dissertação (Mestrado) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2013.

CUBAS, H.A.; NAVARRO SOARES JR., J. (2012). Top-down design for low power multi-bit sigma-delta modulator. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 25., 2012, Brasília. **Proceedings...** [S.l.:s.n.].

DiCRISTINA, J. (2010). **Introduction to hearing aids and important design considerations**. Disponível em<http://www.maximintegrated.com/app-notes/index.mvp/id/4691>. Acesso em: 31/07/2013.

DINIZ, P.S.R. (2008). **Adaptive filtering -** algorithms and practical implementation. Berlin: Springer.

EMNETT, F.; BIEGEL, M. (2000). Power reduction through RTL clock gating. In: SYNOPSES USER´S GROUP, 2000, Santa Clara. [S.l.:s.n].

ERKAN, F. (2009). **Design and implementation of a fixed point digital active noise controller headphone**. Dissertation (MSc) - School of Natural and Applied Science, Middle East Trchnical University, 2009.

FISHER, J.A.; FARABOSCHI, P.; YOUNG, C. (2005). **Embedded computing** - a VLIW approach to architecture, compilers and tools. New York: Elsevier.

FRANTZ, G. (2000). Digital signal processor trends. **IEEE Micro**, v.20, n.6, p.52-59, Nov./Dec.

GENNUM CORPORATION. (2007). **Voyageu µDSP programmer's manual**. San Francisco.

_____. (2013). **GA3280, "voyager open platform dsp system for ultra low power audio processing"**. San Francisco.

HAYKIN, S.S. (2002). **Adaptive filter theory**. 4th ed. New Jersey: Prentice Hall.

_____. (2005). **Adaptive filter theory**. London: Pearson Education.

HERBIG, R. (2009). **Acceptable delay in digital hearing aids**. [S.l.]: Siemens.

INSTITUTO BRASILEIRO DE GEOGRAFIA E ESTATÍSTICA. (2005). **Sistema IBGE de recuperação automática**. Disponível em:<**http://www.sidra.ibge.gov.br/**>. Acesso em: 03/06/2011.

JACOBS, D. (2006). **Digital pulse width modulation for class-D audio amplifiers**. Dissertation (MSc) - University of Stellenbosch, Stellenbosh, 2006.

KATES, J. M. (2008). **Digital hearing aids**. San Diego: Plural.

LABORATÓRIO DE SISTEMAS INTEGRAVEIS TECNOLÓGICO. (2011). Disponível em:<**http://www.lsitec.org.br/dh/**>. Acesso em: 03/06/2011.

LAMENZA, A.R.D.O. (2005). **Análise compartiva do desempenho de aparelhos auditivos analógicos versus digitais no ruído**. Dissertação (Mestrado) - Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2005.

LAPSLEY, P. et al. (1997). **DSP processor fundamentals**: architectures and features. New York: Wiley-IEEE Press.

LIU, D. (2008). **Embedded DSP processor design**: application specific instruction set processors. Amsterdam: Morgan Kaufmann.

LUO, H.; ARNDT, H. (2002). Digital signal processing technology and applications in hearing aids. In: INTERNATIONAL CONFERENCE ON SIGNAL PROCESSING, 6., 2002. **Proceedings…** New York: IEEE. v.2, p.1727-1730.

MALUENDA, Y.R.M. (2009). **Análise estatística do algoritmo LMS aplicado ao cancelamento de retroalimentação acústica em aparelhos auditivos**. Tese (Doutorado) - Universidade Federal de Santa Catarina, Florianópolis, 2009.

MANOLAKIS, D.G.; INGLE, V.K.; KOGON, S.M. (2005). **Statistical and adaptive signal processing**: spectral estimation, signal modeling, adaptive filtering and array processing. Noewood: Artech House.

MATHWORKS. (2013). Signal Processing toolbox. Disponível em:<http://www.mathworks.com/products/signal/examples.html?file=/products/demos/shipping/signal/introfdatooldemo.html>. Acesso em: 31/07/2013.

MATOS, G.G.O. (2010). **Processamento temporal em perdas auditivas sensorioneurais**. Dissertação (Mestrado) - Universidade Veiga de Almeida, Rio de Janeiro, 2010.

MITRA, S.K. (2001). **Digital signal processing a computer-based approach**. New York: McGraw-Hill.

MOONEN, M.; WATERSCHOOT, T.V. (2009). Adaptive feedback cancellation for audio applications. **Signal Processing**, v.89, n.11, p.2185-2201, Nov.

MOSCH, P. et al. (2000). A 660 uW 50-Mops 1-V DSP for a hearing aids chip set. **IEEE Journal of Solid State Circuits**, v.35, n.11, p.1705-1712, Nov.

NAGEL, J.L. et al. (2009). **The Icyflex4 processor, a scalable DSP architecture**. [S.l.:s.n.] (CSEM Scientific and Technical Report 2009).

NEUTEBOOM, H.; KUP, B.M.J.; JANSSENS, M. (1997). A DSP-based hearing instrument IC. **Journal of Solid-State Circuits**, v.32, n.11, p.1790-1806, Nov.

NVIDIA´S next generation CUDA™ compute architecture: kepler™ Gk110. Disponível em:<http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>. Acesso em: 31/07/2013.

NXP. (2009). **Assembly programmer's manual**. [S.l.:s.n.].

_____. (2013). **CoolFlux DSP the embedded ultra low power C-programmable DSP core**. Disponível em:<**http://www.coolflux.com/nxp-coolflux-dsp**>. Acesso em: 31/07/2013.

OLIVEIRA, E.G.D. (2007). **Apostila de Telefonia**. São Paulo: [s.n.].

PARHI, K.K. (1999). **VLSI digial signal processing systems**. New York: Wiley.

PATTERSON, D.A.; HENNESSY, J.L. (2007). **Computer architecture a quantitative approach**. New York: Elsevier.

PENTEADO, S.P. (2009). **Desenvolvimento de prótese auditiva retroauricular, digital e de especificações mínimas**. Tese (Doutorado) - Faculdade de Medicina, Universidade de São Paulo, São Paulo, 2009.

PRONAK. (2013). Disponível em:<http://www.phonak.com/uk/b2c/en/home.html>. Acesso em: 31/07/2013.

RAMLI, R.M.; ABID NOOR, A.; SAMAD, S.A. (2012). A Review of adaptive line enhancers for noise cancellation. **Australian Journal of Basic and Applied Sciences**, v.6, n.6, p.337-352.

SCHREIER, R.; TEMES, G.C. (2004). **Understanding delta-sigma data converters**. New York: Wiley-Interscience.

SIXSEMICONDUTORES. (2013). Disponível em: < http://www.sixsemicondutores.com.br/?lang=pt.> Acesso em: 31/07/2013.

ON SEMICONDUCTOR. (2013). Ezairo ™ 5900 series high precision sound for next-generation digital hearing aids. Disponível em:<**http://www.onsemi.com/site/pdf/ezairo5900_fs.pdf**>. Acesso em: 31/07/2013.

SMITH, S.W. (1998). **The Scientist and engineer's guide to digital signal processing**. San Diego: California Technical.

STETZLER, T. et al. (2000). **Low-power real-time programmable DSP development platform for digital hearing aids**. Texas: Texas Instruments. (Application Report, SPRA 657). Disponível em:<**http://www.ti.com/lit/an/spra657/spra657.pdf**>. Acesso em: 31/07/2013.

TEXAS INSTRUMENTS. (2001). Whats is DSP + analog?. Disponível em:<http://www.ti.com/corp/docs/investor/dsp/>. Acesso em: 03/06/2011.

HAMACHER, V. et al. (2006). **Applications of adaptive signal processing methods in high-end**. [S.l.:s.n.]

VAHID, F.; GIVARGIS, T. (1999). **Embedded system design: a unified hardware/software approach**. New York: John Wiley.

VANHOOK, J. et al. (1993). **High-level synthesis for real-time digital signal processing**. Berlin: Springer.

YAN, Z. (2006). **Background noise cancellation for acoustic detection of manatee vocalizations**. Tesis (PhD.) - University of Florida, Florida, 2006.

FASTI, H.; ZWICKER, E. (2007). **Psychoacoustics**: facts and models. Berlin: Springer.

APÊNDICE A – Código verilog filtro configurável

```
/* ---------------------------------------------------------------
---------------------------------------------
 Universidade de São Paulo - Sao Paulo
-------------------------------------------------------------------
----------------------------------------
PROJECT NAME:Mestrado
PROJECT MODULE:Filtro Configurável
FILE NAME:filtro_conf.v
AUTHOR: Dionisio de Carvalho
E-MAIL: dionisio.carvalho@usp.br
Propósito : Implementação RTL - Verilog de filtro FIR com diferentes
respostas em frequência ( número de coeficientes variáveis) e com
memória circular para deslocamento de amostras.
Implementado com arquitetura do multiplicador serial
----------------------------------------------------------------*/
`timescale 1 ns / 1 ns

module filtro_conf
#(
parameter WORD_WIDTH_IN = 'd24, //Dados do codec de entrada
parameter WORD_WIDTH_OUT = 'd24, //Dados para o DSP
parameter COEF_WIDTH = 'd16 //tamanho do coeficiente
)
//////////////////////////////////////////////////////////////////
//Input and Output Declaration                                 ///////
//////////////////////////////////////////////////////////////////
(
// Coefficients
  input signed [COEF_WIDTH-1:0] coeff0_127,
  input signed [COEF_WIDTH-1:0] coeff1_126,
  input signed [COEF_WIDTH-1:0] coeff2_125,
  input signed [COEF_WIDTH-1:0] coeff3_124,
  input signed [COEF_WIDTH-1:0] coeff4_123,
  input signed [COEF_WIDTH-1:0] coeff5_122,
  input signed [COEF_WIDTH-1:0] coeff6_121,
  input signed [COEF_WIDTH-1:0] coeff7_120,
  input signed [COEF_WIDTH-1:0] coeff8_119,
  input signed [COEF_WIDTH-1:0] coeff9_118,
  input signed [COEF_WIDTH-1:0] coeff10_117,
  input signed [COEF_WIDTH-1:0] coeff11_116,
  input signed [COEF_WIDTH-1:0] coeff12_115,
  input signed [COEF_WIDTH-1:0] coeff13_114,
  input signed [COEF_WIDTH-1:0] coeff14_113,
  input signed [COEF_WIDTH-1:0] coeff15_112,
  input signed [COEF_WIDTH-1:0] coeff16_111,
  input signed [COEF_WIDTH-1:0] coeff17_110,
  input signed [COEF_WIDTH-1:0] coeff18_109,
  input signed [COEF_WIDTH-1:0] coeff19_108,
  input signed [COEF_WIDTH-1:0] coeff20_107,
  input signed [COEF_WIDTH-1:0] coeff21_106,
  input signed [COEF_WIDTH-1:0] coeff22_105,
  input signed [COEF_WIDTH-1:0] coeff23_104,
```

```
  input signed [COEF_WIDTH-1:0] coeff24_103,
  input signed [COEF_WIDTH-1:0] coeff25_102,
  input signed [COEF_WIDTH-1:0] coeff26_101,
  input signed [COEF_WIDTH-1:0] coeff27_100,
  input signed [COEF_WIDTH-1:0] coeff28_99,
  input signed [COEF_WIDTH-1:0] coeff29_98,
  input signed [COEF_WIDTH-1:0] coeff30_97,
  input signed [COEF_WIDTH-1:0] coeff31_96,
  input signed [COEF_WIDTH-1:0] coeff32_95,
  input signed [COEF_WIDTH-1:0] coeff33_94,
  input signed [COEF_WIDTH-1:0] coeff34_93,
  input signed [COEF_WIDTH-1:0] coeff35_92,
  input signed [COEF_WIDTH-1:0] coeff36_91,
  input signed [COEF_WIDTH-1:0] coeff37_90,
  input signed [COEF_WIDTH-1:0] coeff38_89,
  input signed [COEF_WIDTH-1:0] coeff39_88,
  input signed [COEF_WIDTH-1:0] coeff40_87,
  input signed [COEF_WIDTH-1:0] coeff41_86,
  input signed [COEF_WIDTH-1:0] coeff42_85,
  input signed [COEF_WIDTH-1:0] coeff43_84,
  input signed [COEF_WIDTH-1:0] coeff44_83,
  input signed [COEF_WIDTH-1:0] coeff45_82,
  input signed [COEF_WIDTH-1:0] coeff46_81,
  input signed [COEF_WIDTH-1:0] coeff47_80,
  input signed [COEF_WIDTH-1:0] coeff48_79,
  input signed [COEF_WIDTH-1:0] coeff49_78,
  input signed [COEF_WIDTH-1:0] coeff50_77,
  input signed [COEF_WIDTH-1:0] coeff51_76,
  input signed [COEF_WIDTH-1:0] coeff52_75,
  input signed [COEF_WIDTH-1:0] coeff53_74,
  input signed [COEF_WIDTH-1:0] coeff54_73,
  input signed [COEF_WIDTH-1:0] coeff55_72,
  input signed [COEF_WIDTH-1:0] coeff56_71,
  input signed [COEF_WIDTH-1:0] coeff57_70,
  input signed [COEF_WIDTH-1:0] coeff58_69,
  input signed [COEF_WIDTH-1:0] coeff59_68,
  input signed [COEF_WIDTH-1:0] coeff60_67,
  input signed [COEF_WIDTH-1:0] coeff61_66,
  input signed [COEF_WIDTH-1:0] coeff62_65,
  input signed [COEF_WIDTH-1:0] coeff63_64,
  //clock
  input   clk,//clock do sistema,
  input   rst_n,
  input   signed [WORD_WIDTH_IN-1:0] input_ch1,
  input   signed [WORD_WIDTH_IN-1:0] input_ch2,
  input   [7:0] filter_lenght, //máximo 128
  input   sample_in, //sinal que indica nova amostra
  input   num_channel, //configura para 1 ou 2 canais

  output reg  sample_out, //sinal para dsp, amostra calculada!
  output reg  signed [WORD_WIDTH_OUT-1:0] data_to_dsp_ch1,
  output reg  signed [WORD_WIDTH_OUT-1:0] data_to_dsp_ch2
);
```

```
//*****************************************************************
***********************
//Internal Registers and wires declarations
//*****************************************************************
***********************


     reg [7:0] wr_ptr; //Ponteiro de escrita - até 128 posições

     //Dois ponteiros de leitura devido simetria
     reg [7:0] rd_ptr1; //Ponteiro de leitura 1
     reg [7:0] rd_ptr2; //Ponteiro de leitura 2
 reg  signed [WORD_WIDTH_IN-1:0]  circ_buffer_ch1[0:127];
 reg  signed [WORD_WIDTH_IN-1:0]  circ_buffer_ch2[0:127];
 reg signed [45:0] accu_ch1;
 reg signed [45:0] accu_ch2;
 reg signed [WORD_WIDTH_IN:0] sum_samples_ch1 [0:63]; //25 bits
tamanho da palavra e 64 amostras
 reg signed [WORD_WIDTH_IN:0] sum_samples_ch2 [0:63]; //25 bits
tamanho da palavra e 64 amostras
 reg [7:0] k;
 reg [6:0] filter_half; //64
 reg done_cal,ctr_output1,ctr_output2,ctr_output3,
     ctr_output4,ctr_output5;

 // Accumulator for add the multiplications results:

wire signed [COEF_WIDTH-1:0] COEFF [0:63];//coeficientes simétricos
reg signed [45:0] round_ch1;
reg  signed [45:0] round_ch2;
integer index;

//_-----------------------------------------------------------------
------
assign COEFF [0] = coeff0_127;
assign COEFF [1] = coeff1_126;
assign COEFF [2] = coeff2_125;
assign COEFF [3] = coeff3_124;
assign COEFF [4] = coeff4_123;
assign COEFF [5] = coeff5_122;
assign COEFF [6] = coeff6_121;
assign COEFF [7] = coeff7_120;
assign COEFF [8] = coeff8_119;
assign COEFF [9] = coeff9_118;
assign COEFF [10] = coeff10_117;
assign COEFF [11] = coeff11_116;
assign COEFF [12] = coeff12_115;
assign COEFF [13] = coeff13_114;
assign COEFF [14] = coeff14_113;
assign COEFF [15] = coeff15_112;
assign COEFF [16] = coeff16_111;
assign COEFF [17] = coeff17_110;
assign COEFF [18] = coeff18_109;
assign COEFF [19] = coeff19_108;
assign COEFF [20] = coeff20_107;
assign COEFF [21] = coeff21_106;
```

```
assign COEFF [22] = coeff22_105;
assign COEFF [23] = coeff23_104;
assign COEFF [24] = coeff24_103;
assign COEFF [25] = coeff25_102;
assign COEFF [26] = coeff26_101;
assign COEFF [27] = coeff27_100;
assign COEFF [28] = coeff28_99;
assign COEFF [29] = coeff29_98;
assign COEFF [30] = coeff30_97;
assign COEFF [31] = coeff31_96;
assign COEFF [32] = coeff32_95;
assign COEFF [33] = coeff33_94;
assign COEFF [34] = coeff34_93;
assign COEFF [35] = coeff35_92;
assign COEFF [36] = coeff36_91;
assign COEFF [37] = coeff37_90;
assign COEFF [38] = coeff38_89;
assign COEFF [39] = coeff39_88;
assign COEFF [40] = coeff40_87;
assign COEFF [41] = coeff41_86;
assign COEFF [42] = coeff42_85;
assign COEFF [43] = coeff43_84;
assign COEFF [44] = coeff44_83;
assign COEFF [45] = coeff45_82;
assign COEFF [46] = coeff46_81;
assign COEFF [47] = coeff47_80;
assign COEFF [48] = coeff48_79;
assign COEFF [49] = coeff49_78;
assign COEFF [50] = coeff50_77;
assign COEFF [51] = coeff51_76;
assign COEFF [52] = coeff52_75;
assign COEFF [53] = coeff53_74;
assign COEFF [54] = coeff54_73;
assign COEFF [55] = coeff55_72;
assign COEFF [56] = coeff56_71;
assign COEFF [57] = coeff57_70;
assign COEFF [58] = coeff58_69;
assign COEFF [59] = coeff59_68;
assign COEFF [60] = coeff60_67;
assign COEFF [61] = coeff61_66;
assign COEFF [62] = coeff62_65;
assign COEFF [63] = coeff63_64;

// Ponteiro para fazer  o deslocamento para cada nova amostra

always @( posedge clk, negedge rst_n) // 32 kHz
begin
      if (!rst_n) begin
      filter_half <='d0;
                  end
      else begin
        if(!filter_lenght[0]) filter_half <= filter_lenght[7:1];
    else                   filter_half <= filter_lenght[7:1] +1;
       end
end
```

```
always @( posedge clk, negedge rst_n) // 32 kHz
     begin

          //zerando ponteiro
     if (!rst_n)     begin
                wr_ptr <= 'd0;
                                        end

          else  begin
                if(sample_in) begin
                circ_buffer_ch1[wr_ptr] <= input_ch1;
                circ_buffer_ch2[wr_ptr] <= input_ch2;
                     if (wr_ptr < filter_lenght-1)
                          wr_ptr <= wr_ptr + 1;
                     else
                          wr_ptr <= 'd0;
                                  end
                end
end // end always

always @(posedge clk,negedge rst_n)
begin: somas
  if(sample_in) begin
      index = 0;
                end
  else begin
      if(index < filter_half) begin
rd_ptr1 = (wr_ptr >= index+1)?(wr_ptr-index-1):(filter_lenght +
wr_ptr-1-index ); //le da amostra mais nova
rd_ptr2 = (wr_ptr < filter_lenght-index)?(wr_ptr+index):
(wr_ptr+index-filter_lenght); //le da amostra mais antiga

sum_samples_ch1[index] = circ_buffer_ch1[rd_ptr1] +
circ_buffer_ch1[rd_ptr2];
sum_samples_ch2[index] = circ_buffer_ch2[rd_ptr1] +
circ_buffer_ch2[rd_ptr2];
 index = index +1;
                           end
     end //else
end //always
always @(posedge clk,negedge rst_n)
begin: multiplications
  if(!rst_n)  begin
        accu_ch1 <= 46'd0;
        accu_ch2 <= 46'd0;
         k <= 'd0;
         done_cal <= 1'b0;
               end // end if rst_n ==1
     else       begin

          if (sample_in) k <=0;
          if(k < filter_half +2) begin

                if(k==8'd0)begin
                 accu_ch1 <= 'd0;
```

```
                 accu_ch2 <= 'd0;
                 round_ch1 <= 'd0;
                 round_ch2 <= 'd0;
                                 end

     if(k>8'd0 && k < filter_half+1) begin
    accu_ch1 <= accu_ch1 + sum_samples_ch1[k-1] * COEFF[k-1];
    accu_ch2 <= accu_ch2 + sum_samples_ch2[k-1] * COEFF[k-1];
                                    end
     if (k == filter_half+1)    begin
round_ch1<=({accu_ch1[45:0]}+{accu_ch1[22], {21{~accu_ch1[22]}}});
round_ch2<=({accu_ch2[45:0]}+{accu_ch2[22], {21{~accu_ch2[22]}}});
                done_cal <=1;
                                 end
             k <= k+1;
                                              end
                else begin
                     done_cal <=0;
                          end

     end //else
end //always
// Register to hold the final value of the accumulated sum
always @ (posedge clk,negedge rst_n)
    begin: final_output
      if (!rst_n) begin
         data_to_dsp_ch1 <= 0;
            data_to_dsp_ch2 <= 0;
             end
      else begin
        if(sample_out) begin
           data_to_dsp_ch1 <= round_ch1[38:15];
          data_to_dsp_ch2 <= round_ch2[38:15];
                     end
                end
end

always @(posedge clk, negedge rst_n)
begin
 if (!rst_n) begin
       ctr_output1 <= 1'b0;
       ctr_output2 <= 1'b0;
       ctr_output3 <= 1'b0;
       ctr_output4 <= 1'b0;
       ctr_output5 <= 1'b0;
                         end
   else begin
          ctr_output1 <= done_cal;
           ctr_output2 <= ctr_output1;
           ctr_output3 <= ctr_output2;
           ctr_output4 <= ctr_output3;
           ctr_output5 <= ctr_output4;
           sample_out  <= ctr_output1 & (~ctr_output5);
      end
end
endmodule
```

APÊNDICE B – Código Verilog filtro ALE

```
/* ----------------------------------------------------------------
-----------------------------------------------
 Universidade de São Paulo - Sao Paulo
-------------------------------------------------------------------
----------------------------------------
PROJECT NAME:Mestrado
PROJECT MODULE:Filtro ALE
FILE NAME:filtro_ale.v
AUTHOR: Dionisio de Carvalho
E-MAIL: dionisio.carvalho@usp.br

//  Propósito  :  Implementação  RTL  -  Verilog  de  um  filtro  ale
(adaptive linear estimate) para remoção de sinais periódicos.
Implementado com arquitetura do multiplicador paralelo.
Implementado com registradores de deslocamento
----------------------------------------------------------------*/

`timescale 1 ns / 100 fs
module filtro_ale
(
  input clk, //clock do sistema
  input rst_n,
  input sample_in,
  input out_filter,
  input [3:0]mu, //Valor de µentre 1/2^7 e 1/2^4;
  input signed [23:0] data_from_dsp,
  output reg sample_out,
  output reg signed [23:0]data_to_dsp,
  output reg signed [23:0]data_to_filter
);
// Registradores que armazenam os coeficientes atualizados.
// 24 regs com precisão de 16 bits.
reg signed [15:0] w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11,
w12, w13,w14, w15,w16, w17, w18, w19, w20, w21,w22, w23;
//Registradores de deslocamento que armazenam as amostras
reg   signed[23:0]   input_atual,   input_delay_1,   input_delay_2,
input_delay_3,    input_delay_4,    input_delay_5,    input_delay_6,
input_delay_7,input_delay_8,input_delay_9,input_delay_10,
input_delay_11,input_delay_12,    input_delay_13,    input_delay_14,
input_delay_15,         input_delay_16,input_delay_17,input_delay_18,
input_delay_19,    input_delay_20,    input_delay_21,input_delay_22,
input_delay_23, input_delay_24;
//Registradores que armazenam amostra*coenficiente.
reg  signed[38:0]  multiplicador1,  multiplicador2,  multiplicador3,
multiplicador4,    multiplicador5,multiplicador6,    multiplicador7,
multiplicador8,   multiplicador9,   multiplicador10,multiplicador11,
multiplicador12,          multiplicador13,          multiplicador14,
multiplicador15,multiplicador16,  multiplicador17,  multiplicador18,
multiplicador19, multiplicador20,                   multiplicador21,
multiplicador22, multiplicador23, multiplicador24;

//Registradores para calcular ajuste dos coeficientes
```

```
reg     signed     [46:0]     prod_erro_x0,prod_erro_x1,prod_erro_x2,
prod_erro_x3,prod_erro_x4,prod_erro_x5,prod_erro_x6,prod_erro_x7,pro
d_erro_x8,prod_erro_x9,prod_erro_x10,prod_erro_x11,
prod_erro_x12,prod_erro_x13,prod_erro_x14,prod_erro_x15,
prod_erro_x16,prod_erro_x17,prod_erro_x18,prod_erro_x19,
prod_erro_x20,prod_erro_x21,prod_erro_x22,prod_erro_x23;

//Registradores para ajuste truncado
reg   signed   [15:0]   ajust_w0,ajust_w1,ajust_w2,ajust_w3,ajust_w4,
ajust_w5,ajust_w6,ajust_w7,ajust_w8,ajust_w9,ajust_w10,ajust_w11,aju
st_w12,ajust_w13,ajust_w14,ajust_w15,ajust_w16,            ajust_w17,
ajust_w18,ajust_w19,ajust_w20,ajust_w21,ajust_w22,ajust_w23;

//define internal reg
reg signed  [23:0] n_hat_k, erro;
reg signed  [41:0] accu;

wire [3:0] mu_reg;

assign mu_reg = mu-1;//ja realiza a multiplicação por 2 para o
calculo do novos valores de w
//instancia da maquina de controle usada para gerar os sinais de
controle das operações
wire mul_coefs,  trunc_prods,  sum_stg_a,  trunc_out,  feed_back_nk,
feed_back_mul, sum_stg_b, trunc_fb_prods, sample_out_state;
control_sm  control_sm    ( .clk(clk),
                            .rst_n(rst_n),
                            .sample_in(sample_in),
                            .mul_coefs(mul_coefs),
                            .trunc_prods(trunc_prods),
                            .sum_stg_a(sum_stg_a),
                            .trunc_out(trunc_out),
                            .feed_back_nk(feed_back_nk),
                            .feed_back_mul(feed_back_mul),
                            .sum_stg_b(sum_stg_b),
                            .trunc_fb_prods(trunc_fb_prods),
                            .sample_out_state(sample_out_state)
                           );

always @( posedge clk, negedge rst_n)
begin : shift_samples
           if(!rst_n) begin
                 input_atual   <= 24'd0;
                 input_delay_1 <= 24'd0;
                 input_delay_2 <= 24'd0;
                 input_delay_3 <= 24'd0;
                 input_delay_4 <= 24'd0;
                 input_delay_5 <= 24'd0;
                 input_delay_6 <= 24'd0;
                 input_delay_7 <= 24'd0;
                 input_delay_8 <= 24'd0;
                 input_delay_9 <= 24'd0;
                 input_delay_10 <= 24'd0;
                 input_delay_11 <= 24'd0;
                 input_delay_12 <= 24'd0;
                 input_delay_13 <= 24'd0;
```

```
                input_delay_14 <= 24'd0;
                input_delay_15 <= 24'd0;
                input_delay_16 <= 24'd0;
                input_delay_17 <= 24'd0;
                input_delay_18 <= 24'd0;
                input_delay_19 <= 24'd0;
                input_delay_20 <= 24'd0;
                input_delay_21 <= 24'd0;
                input_delay_22 <= 24'd0;
                input_delay_23 <= 24'd0;
                input_delay_24 <= 24'd0;
                               end
                else begin
                if(sample_in) begin
                      input_atual <= data_from_dsp;
                      input_delay_1 <= input_atual;
                      input_delay_2 <= input_delay_1;
                      input_delay_3 <= input_delay_2;
                      input_delay_4 <= input_delay_3;
                      input_delay_5 <= input_delay_4;
                      input_delay_6 <= input_delay_5;
                      input_delay_7 <= input_delay_6;
                      input_delay_8 <= input_delay_7;
                      input_delay_9 <= input_delay_8;
                      input_delay_10 <= input_delay_9;
                      input_delay_11 <= input_delay_10;
                      input_delay_12 <= input_delay_11;
                      input_delay_13 <= input_delay_12;
                      input_delay_14 <= input_delay_13;
                      input_delay_15 <= input_delay_14;
                      input_delay_16 <= input_delay_15;
                      input_delay_17 <= input_delay_16;
                      input_delay_18 <= input_delay_17;
                      input_delay_19 <= input_delay_18;
                      input_delay_20 <= input_delay_19;
                      input_delay_21 <= input_delay_20;
                      input_delay_22 <= input_delay_21;
                      input_delay_23 <= input_delay_22;
                     input_delay_24 <= input_delay_23;
                         end //sample_in
           end //else
end //always

//Multiplicando amostras pelos coeficientes atuais
always @( posedge clk , negedge rst_n)
     begin
           if (!rst_n) begin
                         multiplicador1 <= 'd0;
                      multiplicador2 <= 'd0;
                      multiplicador3 <= 'd0;
                      multiplicador4 <= 'd0;
                      multiplicador5 <= 'd0;
                      multiplicador6 <= 'd0;
                      multiplicador7 <= 'd0;
                      multiplicador8 <= 'd0;
                      multiplicador9 <= 'd0;
```

```
                    multiplicador10 <= 'd0;
                    multiplicador11 <= 'd0;
                    multiplicador12 <= 'd0;
                    multiplicador13 <= 'd0;
                    multiplicador14 <= 'd0;
                    multiplicador15 <= 'd0;
                    multiplicador16 <= 'd0;
                    multiplicador17 <= 'd0;
                    multiplicador18 <= 'd0;
                    multiplicador19 <= 'd0;
                    multiplicador20 <= 'd0;
                    multiplicador21 <= 'd0;
                    multiplicador22 <= 'd0;
                    multiplicador23 <= 'd0;
                    multiplicador24 <= 'd0;
                                        end
                else begin
                   if (mul_coefs)begin
                    multiplicador1 <=  w0 * input_delay_1;
                    multiplicador2 <=  w1 * input_delay_2;
                    multiplicador3 <=  w2 * input_delay_3;
                    multiplicador4 <=  w3 * input_delay_4;
                    multiplicador5 <=  w4 * input_delay_5;
                    multiplicador6 <=  w5 * input_delay_6;
                    multiplicador7 <=  w6 * input_delay_7;
                    multiplicador8 <=  w7 * input_delay_8;
                    multiplicador9 <=  w8 * input_delay_9;
                    multiplicador10 <=  w9 * input_delay_10;
                    multiplicador11 <=  w10 * input_delay_11;
                    multiplicador12 <=  w11 * input_delay_12;
                    multiplicador13 <=  w12 * input_delay_13;
                    multiplicador14 <=  w13 * input_delay_14;
                    multiplicador15 <=  w14 * input_delay_15;
                    multiplicador16 <=  w15 * input_delay_16;
                    multiplicador17 <=  w16 * input_delay_17;
                    multiplicador18 <=  w17 * input_delay_18;
                    multiplicador19 <=  w18 * input_delay_19;
                    multiplicador20 <=  w19 * input_delay_20;
                    multiplicador21 <=  w20 * input_delay_21;
                    multiplicador22 <=  w21 * input_delay_22;
                    multiplicador23 <=  w22 * input_delay_23;
                    multiplicador24 <=  w23 * input_delay_24;
                                end //multiplicações
                            end //else
           end //always
//Soma de todos os multiplicadores
always @(posedge clk , negedge rst_n)
     begin
           if(!rst_n)  accu <= 42'd0;
           else begin
                 if(sum_stg_a) begin
accu  <=  multiplicador1  +  multiplicador2  +  multiplicador3  +
multiplicador4 + multiplicador5 + multiplicador6 +
multiplicador7 + multiplicador8 + multiplicador9 + multiplicador10 +
multiplicador11    +    multiplicador12    +    multiplicador13    +
multiplicador14 + multiplicador15+ multiplicador16 + multiplicador17
```

```
+    multiplicador18    +    multiplicador19    +    multiplicador20+
multiplicador21    +    multiplicador22    +    multiplicador23    +
multiplicador24;
                                    end
                end  //else
     end //always
//Truncamento da saída
always @(posedge clk , negedge rst_n)
begin
    if(!rst_n) begin
      data_to_out <= 24'd0;
      n_hat_k <= 24'd0;
              end
    else begin
                if (trunc_out)begin
                      if(out_filter)
                      data_to_filter <= accu[41:18];
                      else
                      data_to_dsp <= accu[41:18];

                      n_hat_k <= accu[41:18];
                                  end
                end  //else
end //always

//subtraindo entrada atual menos saida atual para estimativa de erro
e multiplicando por mu
always @(posedge clk , negedge rst_n)
begin
    if(!rst_n)
      erro <= 24'd0;
    else begin
                      if (feed_back_nk)
                      erro <= (input_atual - n_hat_k)>>>mu_reg;
                  end //else
end //always
always @(posedge clk , negedge rst_n)
begin
    if(!rst_n) begin
        prod_erro_x0 <= 47'd0;
        prod_erro_x1 <= 47'd0;
        prod_erro_x2 <= 47'd0;
        prod_erro_x3 <= 47'd0;
        prod_erro_x4 <= 47'd0;
        prod_erro_x5 <= 47'd0;
        prod_erro_x6 <= 47'd0;
        prod_erro_x7 <= 47'd0;
        prod_erro_x8 <= 47'd0;
        prod_erro_x9 <= 47'd0;
        prod_erro_x10 <= 47'd0;
        prod_erro_x11 <= 47'd0;
        prod_erro_x12 <= 47'd0;
        prod_erro_x13 <= 47'd0;
        prod_erro_x14 <= 47'd0;
        prod_erro_x15 <= 47'd0;
        prod_erro_x16 <= 47'd0;
```

```
        prod_erro_x17 <= 47'd0;
        prod_erro_x18 <= 47'd0;
        prod_erro_x19 <= 47'd0;
        prod_erro_x20 <= 47'd0;
        prod_erro_x21 <= 47'd0;
        prod_erro_x22 <= 47'd0;
        prod_erro_x23 <= 47'd0;
                                end
                else begin
        if (feed_back_mul) begin
        prod_erro_x0 <= erro*input_delay_1;
        prod_erro_x1 <= erro*input_delay_2;
        prod_erro_x2 <= erro*input_delay_3;
        prod_erro_x3 <= erro*input_delay_4;
        prod_erro_x4 <= erro*input_delay_5;
        prod_erro_x5 <= erro*input_delay_6;
        prod_erro_x6 <= erro*input_delay_7;
        prod_erro_x7 <= erro*input_delay_8;
        prod_erro_x8 <= erro*input_delay_9;
        prod_erro_x9 <= erro*input_delay_10;
        prod_erro_x10 <= erro*input_delay_11;
        prod_erro_x11 <= erro*input_delay_12;
        prod_erro_x12 <= erro*input_delay_13;
        prod_erro_x13 <= erro*input_delay_14;
        prod_erro_x14 <= erro*input_delay_15;
        prod_erro_x15 <= erro*input_delay_16;
        prod_erro_x16 <= erro*input_delay_17;
        prod_erro_x17 <= erro*input_delay_18;
        prod_erro_x18 <= erro*input_delay_19;
        prod_erro_x19 <= erro*input_delay_20;
        prod_erro_x20 <= erro*input_delay_21;
        prod_erro_x21 <= erro*input_delay_22;
        prod_erro_x22 <= erro*input_delay_23;
        prod_erro_x23 <= erro*input_delay_24;
                                end
          end
end //always

//aplicando µe truncando
always @(posedge clk , negedge rst_n)
begin
    if(!rst_n) begin
         ajust_w0 <= 16'd0;
         ajust_w1 <= 16'd0;
         ajust_w2 <= 16'd0;
         ajust_w3 <= 16'd0;
         ajust_w4 <= 16'd0;
         ajust_w5 <= 16'd0;
         ajust_w6 <= 16'd0;
         ajust_w7 <= 16'd0;
         ajust_w8 <= 16'd0;
         ajust_w9 <= 16'd0;
         ajust_w10 <= 16'd0;
         ajust_w11 <= 16'd0;
         ajust_w12 <= 16'd0;
         ajust_w13 <= 16'd0;
```

```
          ajust_w14 <= 16'd0;
          ajust_w15 <= 16'd0;
          ajust_w16 <= 16'd0;
          ajust_w17 <= 16'd0;
          ajust_w18 <= 16'd0;
          ajust_w19 <= 16'd0;
          ajust_w20 <= 16'd0;
          ajust_w21 <= 16'd0;
          ajust_w22 <= 16'd0;
          ajust_w23 <= 16'd0;
                     end
           else begin
              if(trunc_fb_prods)   begin
          ajust_w0 <= prod_erro_x0[46:31];
          ajust_w1 <= prod_erro_x1[46:31];
          ajust_w2 <= prod_erro_x2[46:31];
          ajust_w3 <= prod_erro_x3[46:31];
          ajust_w4 <= prod_erro_x4[46:31];
          ajust_w5 <= prod_erro_x5[46:31];
          ajust_w6 <= prod_erro_x6[46:31];
          ajust_w7 <= prod_erro_x7[46:31];
          ajust_w8 <= prod_erro_x8[46:31];
          ajust_w9 <= prod_erro_x9[46:31];
          ajust_w10 <= prod_erro_x10[46:31];
          ajust_w11 <=  prod_erro_x11[46:31];
          ajust_w12 <=  prod_erro_x12[46:31];
          ajust_w13 <=  prod_erro_x13[46:31];
          ajust_w14 <=  prod_erro_x14[46:31];
          ajust_w15 <=  prod_erro_x15[46:31];
          ajust_w16 <=  prod_erro_x16[46:31];
          ajust_w17 <=  prod_erro_x17[46:31];
          ajust_w18 <=  prod_erro_x18[46:31];
          ajust_w19 <=  prod_erro_x19[46:31];
          ajust_w20 <=  prod_erro_x20[46:31];
          ajust_w21 <=  prod_erro_x21[46:31];
          ajust_w22 <=  prod_erro_x22[46:31];
          ajust_w23 <=  prod_erro_x23[46:31];
                                             end
           end //else
     end //always
//ajuste dos pesos dos coeficientes
always @(posedge clk , negedge rst_n)
begin
    if(!rst_n) begin
                w0 <= 16'b0001000000000000;
                      w1 <= 16'b0000000000000000;
                      w2 <= 16'b00000000000000000;
                      w3 <= 16'b00000000000000000;
                      w4 <= 16'b00000000000000000;
                      w5 <= 16'b00000000000000000;
                      w6 <= 16'b00000000000000000;
                      w7 <= 16'b00000000000000000;
                      w8 <= 16'b00000000000000000;
                      w9 <= 16'b00000000000000000;
                      w10 <= 16'b000000000000000000;
                      w11 <= 16'b000000000000000000;
```

```
                    w12 <= 16'b00000000000000000;
                    w13 <= 16'b00000000000000000;
                    w14 <= 16'b00000000000000000;
                    w15 <= 16'b00000000000000000;
                    w16 <= 16'b00000000000000000;
                    w17 <= 16'b00000000000000000;
                    w18 <= 16'b00000000000000000;
                    w19 <= 16'b00000000000000000;
                    w20 <= 16'b00000000000000000;
                    w21 <= 16'b00000000000000000;
                    w22 <= 16'b00000000000000000;
                    w23 <= 16'b00000000000000000;
                end
          else begin
                    if (sum_stg_b) begin
                    w0 <= w0 + ajust_w0;
                    w1 <= w1 + ajust_w1;
                    w2 <= w2 + ajust_w2;
                    w3 <= w3 + ajust_w3;
                    w4 <= w4 + ajust_w4;
                    w5 <= w5 + ajust_w5;
                    w6 <= w6 + ajust_w6;
                    w7 <= w7 + ajust_w7;
                    w8 <= w8 + ajust_w8;
                    w9 <= w9 + ajust_w9;
                    w10 <= w10 + ajust_w10;
                    w11 <= w11 + ajust_w11;
                    w12 <= w12 + ajust_w12;
                    w13 <= w13 + ajust_w13;
                    w14 <= w14 + ajust_w14;
                    w15 <= w15 + ajust_w15;
                    w16 <= w16 + ajust_w16;
                    w17 <= w17 + ajust_w17;
                    w18 <= w18 + ajust_w18;
                    w19 <= w19 + ajust_w19;
                    w20 <= w20 + ajust_w20;
                    w21 <= w21 + ajust_w21;
                    w22 <= w22 + ajust_w22;
                    w23 <= w23 + ajust_w23;

                              end //if
          end //else
end //always
//sinal para interface de saída
always @(posedge clk , negedge rst_n)
begin    if(!rst_n)
      sample_out <= 'd0;
    else
     sample_out <= sample_out_state;
end //always

endmodule
```

```
/* ----------------------------------------------------------------
---------------------------------------------
 Universidade de São Paulo - Sao Paulo
-------------------------------------------------------------------
-----------------------------------------
PROJECT NAME:Mestrado
PROJECT MODULE:Filtro Ale
FILE NAME:control_sm.v
AUTHOR: Dionisio de Carvalho
E-MAIL: dionisio.carvalho@usp.br

// Propósito :
----------------------------------------------------------------*/
/*
// data path flags
reg  start,mul_coefs, trunc_prods, sum_stg_a, trunc_out,
feed_back_nk, feed_back_mul, sum_stg_b, trunc_fb_prods;
*/

`timescale 1 ns / 100 fs

module control_sm
(
   input clk, //clk
   input sample_in, //nova amostra
   input rst_n, //
   output reg mul_coefs,
      output reg trunc_prods,
      output reg     sum_stg_a,
      output reg trunc_out,
      output reg feed_back_nk,
      output reg feed_back_mul,
      output reg trunc_fb_prods,
      output reg sum_stg_b,
      output reg sample_out_state
);


localparam [3:0] IDLE = 4'd0,
                 WAIT = 4'd1,
                 START = 4'd2,
                 SUM_1 = 4'd3,
                 TRUNC_OUT = 4'd4,
                 FB_NK =  4'd5,
                 FB_MUL =  4'd6,
                 TRUNC_FB = 4'd7,
                 SUM_2 = 4'd8,
                 DONE = 4'd9;

reg [3:0] state, next_state;

//FSM sequential procedure
always @(posedge clk, negedge rst_n)
 begin : seq_procedure
   if(!rst_n) begin
      state <= IDLE;
```

```
   end
   else begin
      state <= next_state;
   end
end

//FSM combinational procedure
always @(*)
begin : comb_procedure
    next_state = state;
   case (state)
      IDLE: begin //0
         if (sample_in == 1) next_state = WAIT;
         else next_state = IDLE;
             end
      WAIT: begin //1
         next_state = START;
             end

      START: begin //2
         next_state = SUM_1;
             end
      SUM_1: begin //3
         next_state = TRUNC_OUT;
                end
      TRUNC_OUT: begin //4
         next_state = FB_NK;
                end
      FB_NK: begin //5
         next_state = FB_MUL;
                end
      FB_MUL: begin //6
         next_state = TRUNC_FB;
                end
      TRUNC_FB: begin //7
         next_state = SUM_2;
                end
      SUM_2: begin //8
         next_state = DONE;
                end
      DONE: begin //9
         next_state = IDLE;
                end
      default: next_state = IDLE;
   endcase
end


always @(state)
begin: output_signal

if(state == IDLE)
begin
  mul_coefs = 1'b0;
      sum_stg_a = 1'b0;
      trunc_out = 1'b0;
```

```
      feed_back_nk = 1'b0;
      feed_back_mul = 1'b0;
      trunc_fb_prods = 1'b0;
      sum_stg_b = 1'b0;
      sample_out_state = 1'b0;
  end

if(state == WAIT )
begin
  mul_coefs = 1'b0;
      sum_stg_a = 1'b0;
      trunc_out = 1'b0;
      feed_back_nk = 1'b0;
      feed_back_mul = 1'b0;
      trunc_fb_prods = 1'b0;
      sum_stg_b = 1'b0;
      sample_out_state = 1'b0;
end

if(state == START)
begin
  mul_coefs = 1'b1;
      sum_stg_a = 1'b0;
      trunc_out = 1'b0;
      feed_back_nk = 1'b0;
      feed_back_mul = 1'b0;
      trunc_fb_prods = 1'b0;
      sum_stg_b = 1'b0;
      sample_out_state = 1'b0;
end

if(state == SUM_1)
begin
  mul_coefs = 1'b0;
      sum_stg_a = 1'b1;
      trunc_out = 1'b0;
      feed_back_nk = 1'b0;
      feed_back_mul = 1'b0;
      trunc_fb_prods = 1'b0;
      sum_stg_b = 1'b0;
      sample_out_state = 1'b0;
end

if(state == TRUNC_OUT)
begin
  mul_coefs = 1'b0;
      sum_stg_a = 1'b0;
      trunc_out = 1'b1;
      feed_back_nk = 1'b0;
      feed_back_mul = 1'b0;
      trunc_fb_prods = 1'b0;
      sum_stg_b = 1'b0;
      sample_out_state = 1'b0;
end

if(state == FB_NK)
```

```
begin
  mul_coefs = 1'b0;
      sum_stg_a = 1'b0;
      trunc_out = 1'b0;
      feed_back_nk = 1'b1;
      feed_back_mul = 1'b0;
      trunc_fb_prods = 1'b0;
      sum_stg_b = 1'b0;
      sample_out_state = 1'b0;
end

if(state == FB_MUL)
begin
  mul_coefs = 1'b0;
      sum_stg_a = 1'b0;
      trunc_out = 1'b0;
      feed_back_nk = 1'b0;
      feed_back_mul = 1'b1;
      trunc_fb_prods = 1'b0;
      sum_stg_b = 1'b0;
      sample_out_state = 1'b0;
end
if(state == TRUNC_FB)
begin
  mul_coefs = 1'b0;
      sum_stg_a = 1'b0;
      trunc_out = 1'b0;
      feed_back_nk = 1'b0;
      feed_back_mul = 1'b0;
      trunc_fb_prods = 1'b1;
      sum_stg_b = 1'b0;
      sample_out_state = 1'b0;
end
if(state == SUM_2)
begin
  mul_coefs = 1'b0;
      sum_stg_a = 1'b0;
      trunc_out = 1'b0;
      feed_back_nk = 1'b0;
      feed_back_mul = 1'b0;
      trunc_fb_prods = 1'b0;
      sum_stg_b = 1'b1;
      sample_out_state = 1'b0;
end
if(state == DONE)
begin
  mul_coefs = 1'b0;
      sum_stg_a = 1'b0;
      trunc_out = 1'b0;
      feed_back_nk = 1'b0;
      feed_back_mul = 1'b0;
      trunc_fb_prods = 1'b0;
      sum_stg_b = 1'b0;
      sample_out_state = 1'b1;
end  end //always
endmodule
```

APÊNDICE C – Códigos Matlab

Geração de sinais para os filtros Configurável e ALE

```
%DIONISIO DE CARVALHO - 7389935
%UNIVERSIDADE DE SÃO PAULO - USP

fs = 16000; %taxa de amostragem
eta = 0.66*sin(2*pi/fs*3400*(0:7999)); %sinal para canal 1
eta2 = 0.66*sin(2*pi/fs*4000*(0:7999)); %sinal para canal 2

for i =1:8000
eta_ch1(i) = eta(i)*2^23; %24 bits - 1 de sinal
eta_ch2(i) = eta2(i)*2^23; %24 bits - 1 de sinal
end

%fd e fd2 variável temporária que será usada para salvar os sinais nos
arquivos txt.
 fd  = fopen('C:\mestrado\filtro_configuravel\mem_ch1.txt','wt');
 fd2 = fopen('C:\mestrado\filtro_configuravel\mem_ch2.txt','wt');

%Salvando os sinais em complemento de 2 com 24 bits
 for i=1:8000
 fprintf(fd,'%s\n', dec2bin(mod((eta_ch1(i)),2^24),24));
 fprintf(fd2,'%s\n', dec2bin(mod((eta_ch2(i)),2^24),24));
 end

fclose(fd)
fclose(fd2)
```

Geração de sinais para o Filtro ALE (DEMO Matlab)

```
% DIONISIO DE CARVALHO
% UNIVERSIDADE DE SÃO PAULO - USP
%% Código para validação do Algoritmo ALE
load handel
Xi = y'/2;
V = 0.5*sin(2*pi/Fs*1000*(1:length(Xi)));
Sinal_composto = Xi + V;
N = length(Xi);
L = 24; %tamanho do filtro
mu = 0.0001;
w = zeros(L,1);  %Pesos para ajustes dos coeficientes
xin = zeros(L,1);     %Registrados do filtro
D = sinal_composto;
for i=1:N
for j=L:-1:2
    xin(j)=xin(j-1); % fazendo o deslocamento das amostras
end
    xin(1)=d(i);                %Insere uma nova amostra no começo
    y(i)=w'*xin;                %Saída do filtro Adaptativo
    error=d(i)-y(i);            %Erro
    e(i)=error;                 %Armazenamento do erro
    wtemp = w + 2* mu *error*xin; %Calculando os novos valores dos
coeficientes
    w=wtemp;                      %Atualizando os valores
end
```

```
%DIONISIO DE CARVALHO - 7389935
%UNIVERSIDADE DE SÃO PAULO - USP

fs = 16000; %taxa de amostragem
eta1 = 0.33*sin(2*pi/fs*2000*(0:73112)); %sinal para canal 1
eta = 0.33*sin(2*pi/fs*3000*(0:73112)); %sinal para canal 1
eta2 = eta1 + eta;
for i =1:73113
eta_ch2(i) = eta2(i)*2^23; %24 bits
end fd  =
fopen('C:\mestrado\filtro_ale_validacao_final\estimulos\senóide.txt','wt');

%Salvando os sinais em complemento de 2 com 24 bits
 for i=1:73112
 fprintf(fd,'%s\n', dec2bin(mod((eta_ch2(i)),2^24),24));
 end
      fclose(fd)
```