Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Coprojeto de um Decodificador de Áudio AAC-LC em FPGA

Renato Coral Sampaio

Dissertação apresentada como requisito parcial
para a conclusão do Mestrado em Informática

Orientador
Prof. Dr. Ricardo Pezzuol Jacobi

Coorientador
Prof. Dr. Pedro de Azevedo Berger

Brasília
2013

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenador: Profa. Dra. Alba Cristina M. A. de Melo

Banca examinadora composta por:

Prof. Dr. Ricardo Pezzuol Jacobi (Orientador) — CIC-UnB
Prof. Dr. Ivan Saraiva Silva — DEI-UFPI
Prof. Dr. Marcus Vinicius Lamar — CIC-UnB

CIP — Catalogação Internacional na Publicação

Coral Sampaio, Renato.

Coprojeto de um Decodificador de Áudio AAC-LC em FPGA / Renato Coral Sampaio. Brasília : UnB, 2013.
93 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2013.

1. MPEG-4, 2. AAC, 3. FPGA, 4. Coprojeto, 5. Áudio,
6. Processamento de Sinais Digitais

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Coprojeto de um Decodificador de Áudio AAC-LC em FPGA

Renato Coral Sampaio

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Prof. Dr. Ricardo Pezzuol Jacobi (Orientador)
CIC-UnB

Prof. Dr. Ivan Saraiva Silva
DEI-UFPI

Prof. Dr. Marcus Vinicius Lamar
CIC-UnB

Profa. Dra. Alba Cristina M. A. de Melo
Coordenador do Mestrado em Informática

Brasília, 5 de julho de 2013

Dedicatória

Dedico este trabalho a todos os interessados nas áreas de codificação de áudio e nas técnicas de aceleração de soluções computacionais que utilizam a abordagem de coprojeto entre hardware e software.

Agradecimentos

Agradeço a todas as consciências amigas intra e extrafísicas que contribuíram com este trabalho. Aos meus pais, Carlos Alberto e Marinêz, e aos meus irmãos, Thaís e André, pelo apoio e incentivo recebido. À minha esposa Graça Dantas pelo inestimável amor e companheirismo. Ao amigo Moacir Maurício Dantas, companheiro de engenharia, pelo exemplo de vida. Ao meu orientador Prof. Dr. Ricardo Jacobi pela orientação esclarecedora e oportunidades oferecidas. Ao meu co-orientador, Prof. Dr. Pedro Berger pelos oportunos esclarecimentos sobre a codificação de áudio. Aos demais professores do programa de mestrado do CIC na UnB e a todos os alunos colegas de laboratório por ajudarem a tornar o ambiente sempre agradável de se trabalhar. 3

Abstract

Audio Coding is present today in many electronic devices. It can be found in radio, tv, computers, portable audio players and mobile phones. In 2007 the Brazilian Government defined the brazilian Digital TV System standard (SBTVD) and adopted the AAC - Advanced Audio Coding as the audio codec.

In this work we use the co-design of hardware and software approach to implement a high performance and low energy solution on an FPGA, able to decode up to 6 channels of audio in real-time. The solution architecture and details are presented along with performance and quality tests. Finally, hardware usage and performance results are presented and compared to other solutions found in literature.

Keywords: MPEG-4, AAC, FPGA, Co-Design, Audio, Digital Signal Processing

Resumo

A Codificação de áudio está presente hoje nos mais diversos aparelhos eletrônicos desde o rádio, a televisão, o computador, os tocadores de música portáteis e nos celulares. Em 2007, o governo do Brasil definiu o padrão do Sistema Brasileiro de TV Digital (SBTVD) que adotou o AAC *Advanced Audio Coding* para codificação de áudio.

Neste trabalho, utilizamos a abordagem de coprojeto combinando software e hardware para implementar uma solução de alto desempenho e baixo consumo de energia em um FPGA, capaz de decodificar até 6 canais de áudio em tempo real. Apresentamos os detalhes da solução bem como os testes de desempenho e qualidade. Por fim, apresentamos os resultados de utilização de hardware e performance juntamente com uma comparação com as demais soluções encontradas na literatura.

Palavras-chave: MPEG-4, AAC, FPGA, Coprojeto, Áudio, Processamento de Sinais Digitais

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Com a adoção cada vez mais expressiva de dispositivos móveis em nosso cotidiano, a cada dia aumentam as necessidades de projetos de circuitos integrados mais sofisticados. Dispositivos multimídia com capacidades de decodificar áudio e video como os telefones celulares e *tablets* atingiram o mercado de massa e necessitam cada vez mais de maior poder de processamento para expandir sua gama de aplicações, tamanho reduzido para maior portabilidade e maior eficiência energética para aumentar sua autonomia. Porém, considerando-se as restrições de desempenho e consumo de energia dos processadores embarcados atuais, verifica-se a necessidade do desenvolvimento de arquiteturas mais eficientes para atingir tais objetivos. Uma das abordagens consiste na combinação de hardware dedicado para funções específicas com software rodando em processadores de propósito geral, ambos funcionando de maneira integrada.

Como parte da modernização do sistema de telecomunicações no Brasil em 2006 foi estabelecido o novo padrão de televisão digital, normatizado em 2007 pela Norma Brasileira ABNT NBR 15602. Este novo padrão, chamado de Sistema Brasileiro de TV Digital (SBTVD) foi baseado no sistema japonês, o *Integrated Services Digital Broadcasting Terrestrial (ISDB-T)*, porém com padrões diferentes de codificação de áudio e vídeo em que adotou-se o MPEG-4. Neste caso, utiliza o codificador H.264 para vídeo e o codificador AAC para áudio. Especificamente no caso do áudio, são exigidas, no mínimo, a decodificação de áudio mono, stereo e multicanal (5.0 e 5.1) do padrão AAC-LC (*Low complexity*).

Dentre as diversas áreas de otimização de processamento, este trabalho se foca no processamento de áudio digital. Mais especificamente, optou-se por trabalhar com o *Advanced Audio Coding* (AAC) que é um padrão de codificação/decodificação de áudio digital com perdas. Padronizado pela ISO e pela IEC em 1997, o AAC faz parte das especificações MPEG-2 e MPEG-4 e foi concebido para ser o sucessor do padrão MP3 (MPEG-1 Layer 3). De 1997 a 2009 o padrão foi aprimorado, sendo hoje utilizado na maior parte dos tocadores de música portáteis, telefones celulares, computadores e TVs Digitais. Suas principais vantagens em relação ao MP3 são o suporte para frequências até 96kHz ao invés de 48kHz, suporte para até 48 canais ao invés de 6 e uma qualidade superior à mesma taxa de bits.

1.1 Motivação e Objetivo

Seguindo a necessidade atual do Brasil para a implantação do sistema de TV Digital, a proposta do trabalho é desenvolver uma arquitetura em um sistema híbrido hardware/-software implementando o decodificador AAC em sua versão LC – *Low Complexity* com desempenho rápido o suficiente para decodificar em tempo real até 6 canais de áudio na configuração 5.1, ou seja, dois pares de canais estéreo, um canal central e um canal de baixa frequência.

Mais especificamente, objetiva-se a implementação de um protótipo funcional em FPGA do AAC-LC de modo que possa ser expandido para versões mais sofisticadas como o *High Efficiency* AAC v1 e v2 e que futuramente possa ser adaptado em um SoC (*System-on-a-Chip*) para uso nos decodificadores de áudio do Sistema Brasileiro de TV Digital (SBTVD).

1.2 Metodologia

A primeira etapa de trabalho consistiu em estudar o funcionamento do algoritmo do AAC-LC a partir do software de referência em C baseado nas Normas ABNT NRB 15602-3 (1), ISO/IEC 13818-7 (18) de 2006 e ISO/IEC 14496-3 (20) de 2009.

Em seguida, avaliou-se a implementação em C realizada pelo grupo da UnB com respeito à qualidade, em comparação ao algoritmo de referência da ISO, e com respeito à performance em relação a implementações disponíveis no mercado. O algoritmo foi otimizado até atingir os requisitos de qualidade e performance desejados para processadores em um PC.

Considerando os requisitos de tempo real para decodificar áudio de 2 canais (one-seg no SBTVD) e 6 canais (full-seg no SBTVD) e a necessidade da posterior integração da solução em um SoC, a etapa seguinte foi a de definir uma plataforma de hardware para que fossem realizados testes de performance específicos. Neste caso, optou-se pelo uso de um FPGA onde seria possível o desenvolvimento de uma arquitetura a ser adaptada em um SoC. A plataforma escolhida para a implementação é a placa de desenvolvimento DE2-70 da Terasic contendo um FPGA Cyclone II da Altera. Esta placa nos dá subsídios para trabalhar com um processador e integrar módulos de hardware dedicados diretamente dentro do mesmo circuito integrado. Além disso, fornece saídas de áudio para realização de testes de qualidade. Como processador, optou-se pelo o NIOS II do próprio fabricante do FPGA que roda em *softcore* e nos fornece ferramentas para o desenvolvimento.

Com a plataforma alvo definida, iniciou-se a adaptação do algoritmo em C para o processador Nios II e em seguida foram realizados os testes de performance para descobrir a frequência necessária para alcançar a decodificação de áudio em tempo real para 2 e 6 canais. Os resultados apontaram frequências muito acima das suportadas pela arquitetura e com isso, definiu-se a necessidade do uso da abordagem de coprojeto combinando módulos em software com módulos em hardware dedicado para o desenvolvimento de uma arquitetura mais eficiente que pudesse alcançar a performance desejada.

Seguindo a abordagem do coprojeto, realizou-se o perfilamento do algoritmo onde os módulos foram classificados quanto a seu esforço computacional. A partir desta classificação, definiu-se os módulos que seriam desenvolvidos em hardware e iniciou-se um processo iterativo onde cada módulo desenvolvido foi testado quanto a sua conformidade

e performance juntamente com o restante do sistema funcionando no processador até que os requisitos de tempo real para 2 e 6 canais fossem atingidos. Além dos requisitos de performance, procurou-se manter o menor uso de área do FPGA e o menor consumo de energia possível.

A Figura 1.1 apresenta o fluxo adotado para o projeto iniciando com o software de referência e finalizando com o protótipo em hardware.

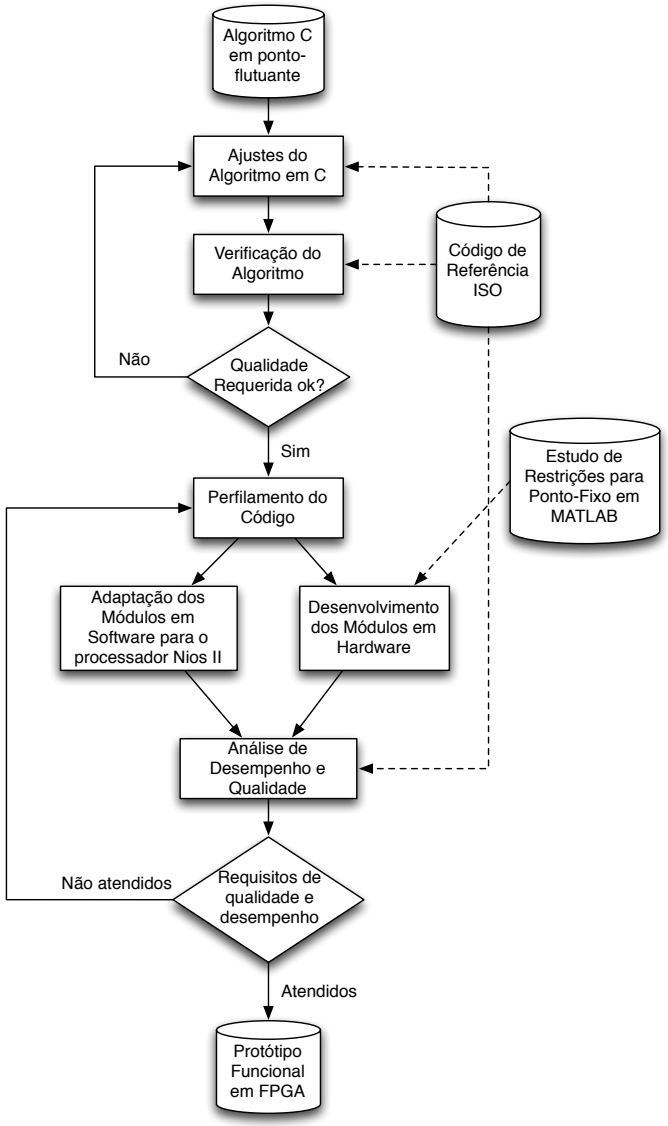

Figura 1.1: Diagrama de Fluxo do Projeto

1.3 Organização do Trabalho

Este trabalho está organizado em capítulos. No Capítulo 1 é feita uma introdução apresentando a motivação, objetivos e metodologia a utilizada. No Capítulo 2, são apresentadas a fundamentação teórica sobre codificação de áudio e sobre o padrão AAC juntamente com uma revisão de literatura sobre o tema e sobre outras abordagens de solução.

Em seguida, no Capítulo 3, apresentamos todo o processo de desenvolvimento da arquitetura do decodificador AAC-LC seguido do Capítulo 4 onde são apresentados os resultados finais e as comparações com arquiteturas com propósito equivalente encontradas na literatura. Finalmente, no Capítulo 5 são apresentadas as conclusões e sugestões de trabalhos futuros.

Capítulo 2

Fundamentação Teórica

2.1 Codificação de Áudio

A codificação ou compressão de áudio tem o objetivo de reduzir o tamanho da representação digital do áudio para uma maior eficiência tanto no armazenamento quanto na transmissão do sinal. Com este intuito, os algoritmos de codificação procuram reduzir ao máximo o tamanho em bits do sinal, preservando a qualidade do som ao ponto em que quando reproduzido novamente não haja diferença perceptível em relação ao som original.

Existem dois grandes grupos de algoritmos de codificação/decodificação (Codec) de áudio. O primeiro é o grupo dos algoritmos sem perdas em que o áudio digital passa por processos genéricos de compactação de dados e quando descompactado, todo o conteúdo digital original é recuperado. Geralmente, nestes casos a taxa de compressão fica em torno de 50% a 60% do tamanho original. O segundo é o grupo dos algoritmos com perdas em que, além da compressão padrão sem perdas, utilizam-se modelos psicoacústicos que procuram eliminar a parte inaudível do som com a utilização de técnicas de representação otimizada do áudio que possam manter a qualidade o mais próxima possível da original. Neste caso, as taxas de compressão variam com a qualidade requerida e tipicamente ficam entre 5% e 20% do tamanho original do arquivo. Algoritmos do segundo grupo são os mais utilizados em equipamentos de larga escala em que a qualidade máxima de som não é obrigatória. A Tabela 2.1 apresenta uma lista dos principais tipos de representação de áudio digital utilizados atualmente.

Tabela 2.1: Tipos de Representação Digital de Áudio e Codecs

	Representação Digital sem Compressão	Codificadores Sem Perda	Codificadores Com Perda
Taxa de Compressão	0%	50% a 60%	5% a 20%
Exemplos	LPCM - Linear Pulse Code Modulation PDM - Pulse-density modulation PAM - Pulse-amplitude modulation	FLAC, Apple Lossless, WMA Lossless, MPEG-4 ALS, RealAudio Lossless	MPEG-1 - Layer 3 (MP3) MPEG-4 (AAC) WMA AC3 DTS OPUS

Para se obter altas taxas de compressão de áudio é necessário primeiro compreender os mecanismos tanto do som quanto do sistema auditivo humano. Assim, é possível saber que tipos de som podem ou não ser suprimidos no processo de codificação.

2.1.1 Sistema Auditivo

O ouvido humano é composto de três partes distintas, o ouvido externo, o ouvido médio e o ouvido interno. O som que ouvimos chega primeiro ao ouvido externo, sendo captado pelo pavilhão auditivo (orelha) e conduzido até o tímpano através do canal auditivo. A pressão gerada pela onda de som faz o tímpano vibrar e por sua vez, este amplifica e transmite a vibração através dos ossículos (martelo, bigorna e estribo) do ouvido médio até a cóclea, órgão responsável pela audição no ouvido interno. A cóclea é um conjunto de três canais em formato de espiral preenchidos por líquidos. No canal médio, fica o órgão de Corti que é o órgão receptor da audição. Este órgão é composto por milhares de células ciliadas e é estimulado pelo movimento das membranas basilar e tectorial. O movimento destas membranas estimuladas pelas vibrações sonoras são transformadas em ondas de compressão que por sua vez ativam o órgão de Corti, responsável por sua transformação em impulsos nervosos que são enviados ao cérebro através do nervo auditivo.

A Figura 2.1 mostra a Anatomia do ouvido humano.

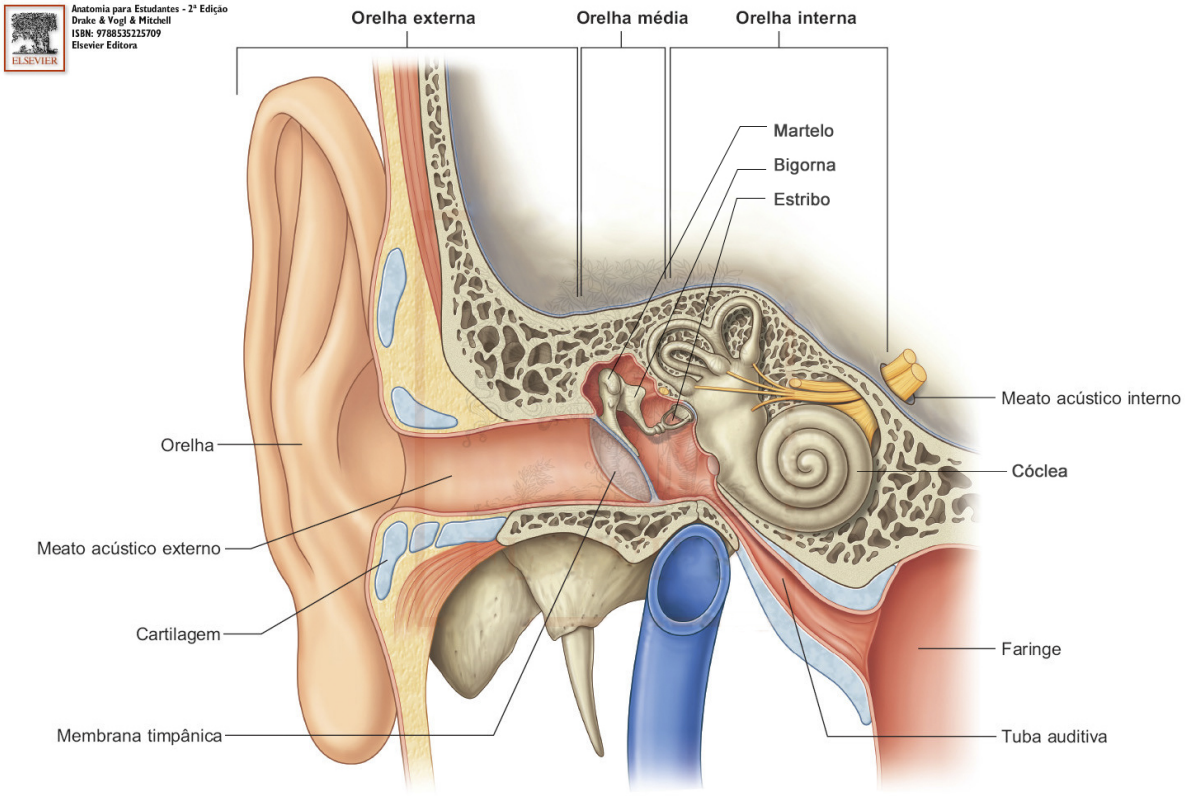

Figura 2.1: Anatomia do Ouvido Humano (Ref. (9))

2.1.2 Características do Som e o Limiar de Audição

O som é a propagação de uma onda mecânica em meios materiais. Sua intensidade pode ser medida por aparelhos e quantificada pela unidade de medida chamada ***sound pressure level*** (**SPL**), representada na escala logarítmica em decibéis (dB). A medida SPL é baseada na pressão mínima necessária para que o ouvido humano perceba um estímulo senoidal de 1 kHz, neste caso o valor da pressão, em Pascal, é $p_0 = 20\mu Pa$. Na Equação 2.1 podemos ver o valor correspondente SPL em dB para um pressão p qualquer.

$$L_{SPL} = 20 \log_{10} \left(\frac{p}{p_0} \right) \text{(dB SPL)} \tag{2.1}$$

Porém, a forma com a qual o ouvido humano percebe o som e sua intensidade diferem de uma escala pura de intensidade de pressão. Neste caso, define-se o conceito de ***loudness*** ou a intensidade com que o estímulo sonoro é percebido pelo ouvido. Esta escala, medida em ***phons***, depende da frequência e pressão sonora e não pode ser medida diretamente por aparelhos.

Além disso, para compreender a audição humana, utiliza-se o conceito de menor diferença perceptível (***Just Noticeable Difference*** - **JND**) que é caracterizada pela menor diferença de pressão que o ouvido humano pode perceber. Este comportamento também não é linear e depende tanto da intensidade quanto da frequência do tom.

Existe ainda a medida do **Limiar Absoluto de Audição** humana que é caracterizada pela energia necessária para que o ouvido humano perceba um tom puro na ausência de ruído. Este limite varia de pessoa para pessoa mas pode ser aproximado pela Equação 2.2 conforme apresentado por Spanias (29). A menor frequência percebida pelo ouvido humano está na faixa de 20 Hz, enquanto a maior frequência percebida é de 20 kHz.

$$T_q(f) = 3,64 \left(\frac{f}{1000} \right)^{-0,8} - 6,5 e^{-0,6\left(\frac{f}{1000}-3,3\right)^2} + 10^{-3} \left(\frac{f}{1000} \right)^4 \text{(dB SPL)} \tag{2.2}$$

A Figura 2.2 apresenta as curvas de contorno da percepção humana para diferentes intensidades e frequências de som. Cada uma das curvas é percebida com a mesma intensidade (*loudness*) que pode ser vista na escala de *phons* apresentada. A curva pontilhada apresenta o limiar absoluto de audição no qual se pode observar que a maior sensibilidade do ouvido é de aproximadamente 4 kHz.

2.1.3 Bandas Críticas

Aprofundando-se mais no funcionamento interno da cóclea, observa-se que diferentes frequências de ondas sonoras geram respostas mais acentuadas em diferentes posições da membrana basilar. Neste caso, os receptores nervosos estão ajustados para diferentes bandas de frequência conforme suas posições ao longo da membrana. De acordo com experimentos realizados (29), dado um estímulo senoidal, a onda sonora que atinge a membrana basilar se propaga desde a janela oval até a região que possua uma frequência ressonante próxima da frequência deste estímulo. A onda, então, desacelera e sua magnitude atinge um pico. A partir deste ponto a onda decai rapidamente. O local onde ocorre o pico é chamado de "o melhor local" ou "o local característico" para o estímulo daquela

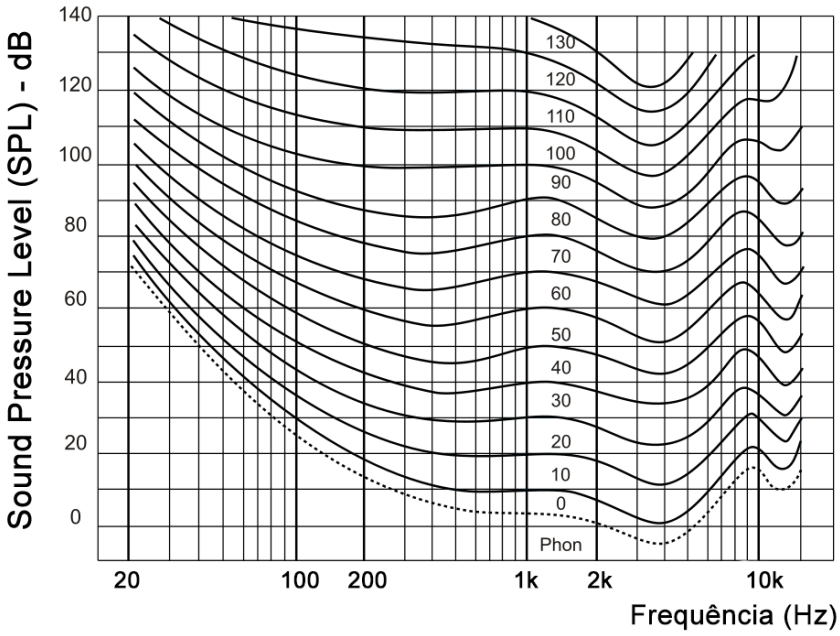

Figura 2.2: Curvas de contorno de igual percepção da audição humana (Ref. (14))

determinada frequência e a frequência que estimula este local é chamada de "a melhor frequência" ou a "frequência característica".

O que ocorre na prática é uma transformada de frequência para um local, o que pode ser observado na Figura 2.3 onde a resposta da membrana basilar a um estímulo com três tons é mapeada. Com este resultado, do ponto de vista do processamento de sinais, pode-se considerar a cóclea como um banco de filtros passa-faixa sobrepostos. Além disso, as respostas à magnitude são assimétricas, não lineares e as larguras de banda dos filtros da cóclea são não uniformes, sendo mais largas com o aumento da frequência.

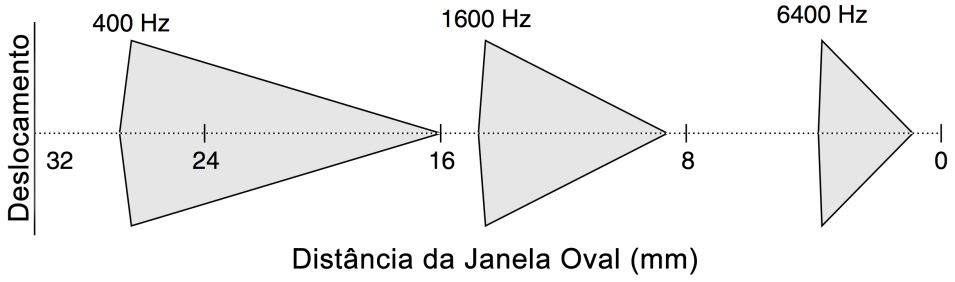

Figura 2.3: A transformada de frequência em local ao longo da Membrana Basilar (29)

Com isto, define-se o conceito de "Banda Crítica" como uma função da frequência que quantifica a largura de bandas dos filtros passa-faixas da cóclea. Uma banda crítica é uma faixa de frequências na qual um tom irá interferir na percepção de outro tom através do fenômeno de mascaramento. A existência da banda crítica demonstra a inabilidade do ouvido humano de perceber diferenças de sons cujas frequências sejam próximas e estejam dentro de um limiar. De acordo com Spanias (29), experimentos realizados com um

ouvinte jovem com audição dentro dos limites normais mostram que a banda crítica tende a se manter constante em 100 Hz para frequências centrais de até 500 Hz, Acima disto, vai aumentando e tende a se manter em aproximadamente 20% do valor da frequência central. A função pode ser aproximada pela Equação 2.3.

$$BW_c(f) = 25 + 75 \left[1 + 1,4 \left(\frac{f}{1000} \right)^2 \right]^{0,69} \text{(Hz)} \tag{2.3}$$

A Equação 2.4 é comumente utilizada para converter a escala de Hertz para Barks, onde cada Bark corresponde a uma banda crítica.

$$Z_b(f) = 13 \arctan(0,00076f) + 35 \arctan \left[\left(\frac{f}{7500} \right)^2 \right] \text{(Bark)} \tag{2.4}$$

Além do modelo aproximado pela equação 2.5, existem outros modelos empíricos como o *equivalent rectangular bandwidth (ERB)*. Este modelo é expresso pela equação e mostra que para frequências abaixo de 500 Hz, as larguras de banda são ainda menores do que 100 Hz.

$$ERB(f) = 24,7 \left(4,37 \left(\frac{f}{1000} \right) + 1 \right) \tag{2.5}$$

2.1.4 Mascaramento do Som

A partir da análise da Banda Crítica, observa-se o fenômeno do Mascaramento do Som. Neste caso, existem dois tipos, o mascaramento simultâneo e o mascaramento temporal. A consequência do mascaramento é que em um determinado som, existem diversos tons e ruídos misturados e, dependendo da amplitude e frequência em que cada um ocorre, haverá mascaramento tornando parte do som inaudível.

O mascaramento simultâneo ocorre quando dois ou mais estímulos sonoros ocorrem no mesmo instante. Neste caso, o sinal de maior intensidade irá estimular a membrana basilar em uma determinada região e bloquear a capacidade da mesma de detectar o outro sinal de menor intensidade. Os três tipos de mascaramento simultâneo mais usados em modelos de codificação são o ruído-mascarando-tom *(Noise-Masking-Ton NMT)*, o tom-mascarando-ruído *(Ton-Masking-Noise TMN)* e o ruído-mascarando-ruído *(Noise-Masking-Noise NMN)*.

No caso do NMT um ruído com largura de banda estreita mascara um tom dentro da mesma Banda Crítica desde que a intensidade deste tom esteja abaixo de um determinado limiar. Neste limiar, a razão mínima entre sinal e o sinal mascarante ocorre quando a frequência do tom mascarado está no centro da frequência do ruído mascarante. Experimentalmente este limiar tende a ficar entre -5 dB e +5 dB.

Para o TNM, o tom de intensidade maior é quem mascara um ruído de menor intensidade. Do mesmo modo, haverá um limiar mínimo quando a frequência do tom for igual à frequência central do ruído mascarante e neste caso, experimentos mostram esta razão do limiar entre 21 dB a 28 dB.

O terceiro tipo NMN é o mais difícil de ser caracterizado, mas experimentos apontam uma razão de aproximadamente 26 dB entre um ruído e outro para que ocorra o mascaramento.

A Figura 2.4 exemplifica os dois primeiros casos de mascaramento simultâneo.

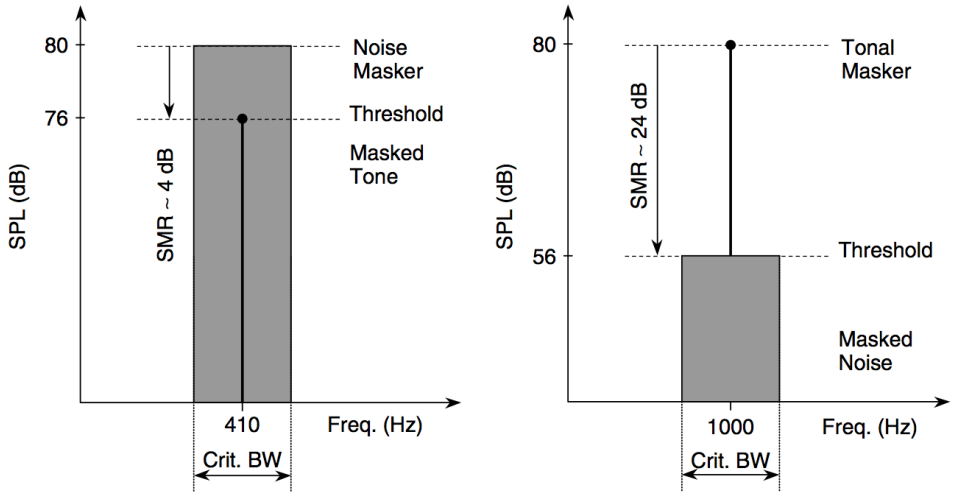

Figura 2.4: Exemplo de Mascaramento Simultâneto (NMT à esquerda e TMN à direita). (Ref. (29))

Ainda referente ao mascaramento simultâneo, outra característica importante é o espalhamento do efeito de mascaramento. Neste caso, observa-se que o efeito de mascaramento de um determinado sinal não se restringe à banda crítica na qual ele ocorre. Na prática, todo sinal pode ter efeito de mascaramento que se espalha pelas bandas críticas vizinhas. O decaimento deste efeito é maior para as frequências menores que a frequência central do sinal e menor para as frequências maiores. A Figura 2.5 mostra o efeito de espalhamento do mascaramento gerado por um determinado tom.

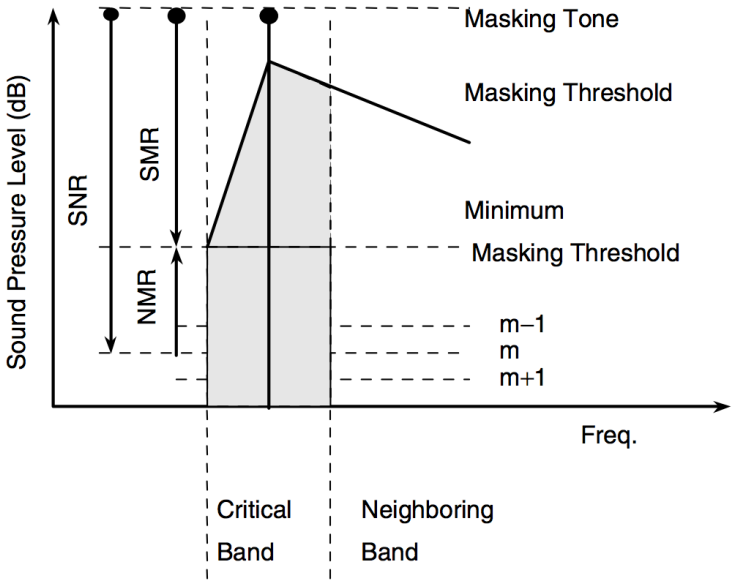

Figura 2.5: Efeito de espalhamento do mascaramento observado para um tom (Ref. (29))

O mascaramento também pode ser não-simultâneo ou temporal. Neste caso, o efeito de mascaramento ocorre ao longo do tempo podendo ser antes ou depois de um sinal mascarante. O pré-mascaramento normalmente ocorre entre 1 a 2 ms antes da ocorrência do sinal enquanto o pós-mascaramento tende a ser mais duradouro sendo válido de 50 até 300 ms após a ocorrência do sinal mascarante. Este efeito, que pode ser visto na Figura 2.6 é pouco compreendido e seus limites ainda são controvertidos na literatura.

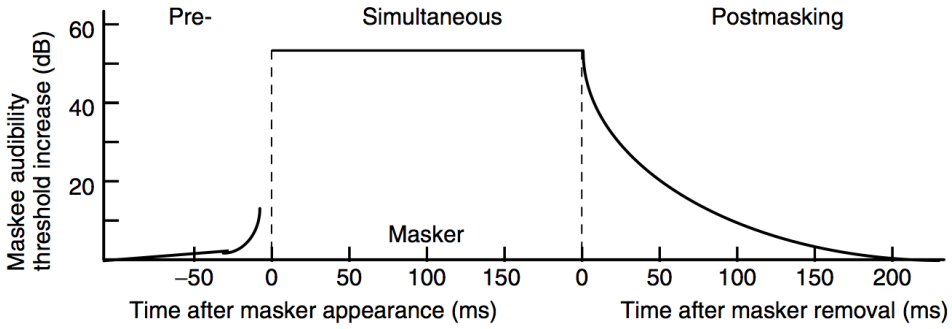

Figura 2.6: Exemplo de Mascaramento Temporal (Ref. (29))

2.1.5 Psicoacústica e Codificação Perceptual

A Psicoacústica é a área de estudo que reúne entre outros, os conceitos apresentados acima para criar um modelo que caracterize o modo como o ouvido humano percebe o som. A partir deste modelo psicoacústico, é possível definir as partes mais relevantes dos sinais sonoros e as partes que podem ser suprimidas por estarem fora dos limites de percepção.

Os codificadores de áudio com perdas também conhecidos como codificadores perceptuais são algoritmos que, a partir de um modelo psicoacústico, identificam as partes relevantes dos sinais sonoros controlando, assim, sua compressão visando não introduzir artefatos ou distorções perceptíveis.

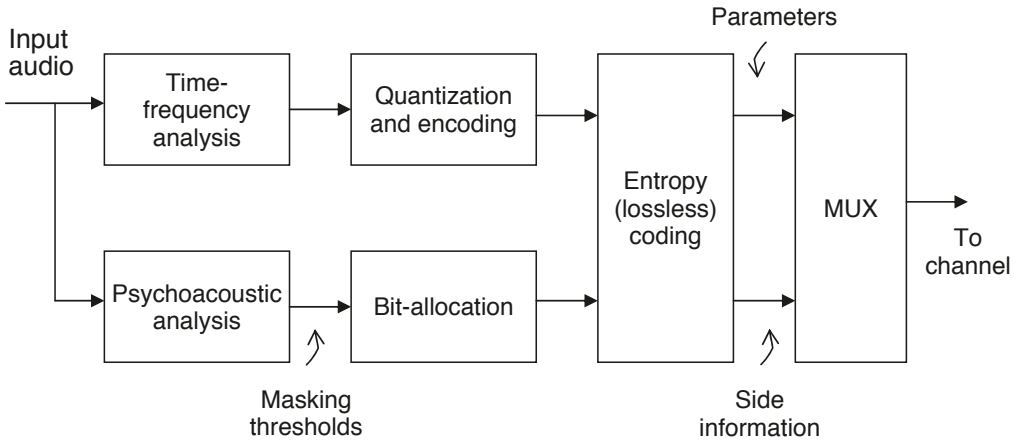

Figura 2.7: Diagrama de um Codificador Perceptual Genérico (Ref. (29))

A Figura 2.7 apresenta a arquitetura genérica de um codificador perceptual. Este tipo de codificador geralmente segmenta o sinal de áudio de entrada em janelas com larguras de 2 a 50 ms. Em seguida, efetua uma transformada do áudio no domínio do tempo para o domínio da frequência e estima os componentes temporais e espectrais de cada janela, o que é feito de acordo com as propriedades psicoacústicas. O objetivo é extrair parâmetros que orientem a quantização. Esta etapa pode ser realizada por: transformada unitária, banco de filtros com tempo variante ou invariante, análise senoidal, análise de predição linear ou uma combinação destes métodos.

O controle de distorção é realizado pelo analisador psicoacústico que estima o valor total de mascaramento e seus limiares máximos para que a quantização não introduza artefatos perceptíveis ao som original.

A etapa de quantização utiliza o modelo psicoacústico juntamente com os parâmetros calculados nas etapas anteriores para remover as partes do áudio não perceptíveis.

Finalmente, as redundâncias restantes no sinal são removidas a partir do codificador de entropia, que efetua uma compactação sem perdas adicionais. O sinal codificado é então organizado contendo os parâmetros usados juntamente com os dados de áudio.

2.1.6 Codificação de Áudio no Sistema Brasileiro de TV Digital

A partir da norma ABNT NBR 15602 (1) o governo Brasileiro definiu o *Advanced Audio Coding* (AAC) como padrão de codificação de áudio para o Sistema Brasileiro de TV Digital (SBTVD). Em sua Parte 2 (15602-2) a norma define que o áudio codificado deve seguir os padrões da norma ISO/IEC 14496-3 (20) com as configurações mínimas apresentadas na Tabela 2.2.

Tabela 2.2: Configurações de Áudio da norma ABNT NBR 15602-2 (1)

Parâmetro	**Restrição**
Frequência de amostragem do sinal de áudio	32 kHz, 44,1 kHz ou 48 kHz
Quantização	16 ou 20 bits
Mecanismos de transporte permitidos	LATM/LOAS (conforme ISO/IEC 14496-3)
Número de canais (One Seg)	Mono (1.0), 2 canais (estéreo ou 2.0)
Número de canais (Full Seg)	Mono (1.0), 2 canais (estéreo ou 2.0), ou multicanal (5.1)
Perfis e níveis permitidos	*Low complexity* AAC: nível 2 (LC-AAC@L2) para dois canais *Low complexity* AAC: nível 4 (LC-AAC@L4) para multicanal *High Efficiency* (HE): nível 2 (HE-AAC v1@L2) para dois canais *High Efficiency* (HE): nível 4 (HE-AAC v1@L4) para multicanal
Amostras por quadro	1024 amostras para AAC e 2048 quando utilizada a técnica SBR (*Spectral band replication*).

Em relação à configuração dos canais temos as opções apresentadas na Tabela 2.3 de acordo com o número de canais.

Tabela 2.3: Configuração de canais - MPEG-4 pela Norma ABNT NER 15602 (1)

Modo	**Núm. de Canais**	**Ordem de Transmissão do SE - *Syntatic Element*[a]**	**Elemento padrão para mapeamento de auto-falantes[b]**
Monoaural (1/0)	1	<SCE1><TERM>	SCE1 = C
Estéreo (2/0)	2	<CPE1><TERM>	CPE1 = L e R
3/0	3	<SCE1><CPE1><TERM>	SCE1 = C, CPE1 = L e R
3/1	4	<SCE1><CPE1><SCE2> <TERM>	SCE1 = C, CPE1 = L e R, SCE1 = MS
Multicanal 5.0 (3/2)	5	<SCE1><CPE1><CPE2> <TERM>	SCE1 = C, CPE1 = L e R, CPE2 = LS e RS
Multicanal 5.1 (3/2 + LFE)	6	<SCE1><CPE1><CPE2> <LFE><TERM>	SCE1 = C, CPE1 = L e R, CPE2 = LS e RS, LFE = LFE
[a] Abreviaturas dos elementos sintáticos: SCE = *Single Channel Element*, CPE = *Channel Pair Element*, LFE = *Low Frequency Channel Element*, TERM = *terminator* [b] Abreviaturas relacionadas ao arranjo de alto-falantes: L – alto-falante frontal esquerdo / R – alto-falante frontal direito / C – alto-falante frontal central / LFE – ênfase de baixa frequência / LS – alto-falante *surround* traseiro esquerdo / RS – alto-falante *surround* traseiro direito / MS – alto-falante *surround* monaural.			

2.2 *Advanced Audio Coding* (AAC) - MPEG-4

Em 1998 o *Moving Picture Experts Group* normatizou o MPEG-4 que é um padrão para codificação de dados digitais de áudio e vídeo. O MPEG-4 incorpora diversas soluções definidas nos padrões anteriores MPEG-1 e MPEG-2 e inclui novas funcionalidades e padrões de codificação de áudio e vídeo.

Especificamente relacionado à codificação de áudio, o MPEG-4 define padrões para codificação de voz e de som em geral com ou sem perdas. Em nosso caso, estamos interessados nas definições de *General audio coding tools* que define os codificadores com perdas e de alta qualidade incluindo o AAC. Neste caso, o padrão AAC vindo do MPEG-2, é aprimorado e ganha mais ferramentas aumentando sua eficiência na qualidade e na compressão dos dados, porém, ainda mantendo compatibilidade com o padrão anterior.

O *Advanced Audio Coding* (AAC) foi concebido para ser o sucessor do padrão MP3 (MPEG-1 Layer 3). Desde 1997 vem sendo aprimorado, e é hoje utilizado na maior parte dos tocadores de música portáteis, telefones celulares, computadores e TVs Digitais. Suas principais vantagens em relação ao MP3 são o suporte para frequências de até 96kHz ao invés de 48kHz, suporte para até 48 canais ao invés de 6 e uma qualidade superior à mesma taxa de bits.

Entre os perfis do Codificador AAC temos:

1. AAC - *Low Complexity* (LC): Utiliza Banco de Filtros baseado em MDCT com janelas de 256 (curta) ou 2048 (longa) amostras, 4 tipos diferentes de formas de janela (*Long, Short, Long-Start, Long-Stop*), modelo sofisticado de mascaramento baseado em modelo psicoacústico de alta qualidade, quantização e codificação do padrão

AAC com o algoritmo *Huffman* para codificação de entropia. Alem disso, possui suportes para ferramentas de processamento espectral como o *Coupling*, *Intensity Stereo* (IS), *Mid/Side Coding* (M/S), *Temporal Noise Shaping* (TNS) e *Perceptual Noise Substitution* (PNS).

2. *High Efficiency* (HE) - AAC versão 1: inclui todas as técnicas do AAC-LC e adiciona a técnica *Spectral band replication* (SBR) para sons com baixas taxas de bits.

3. *High Efficiency* (HE) - AAC versão 2: inclui as ferramentas do HE-AAC v1 e adiciona a função *Parametric Stereo* (PS), também para melhoria na qualidade de sons com baixas taxas de bits.

A Figura 2.8 apresenta a relação entre os diferentes perfis do AAC.

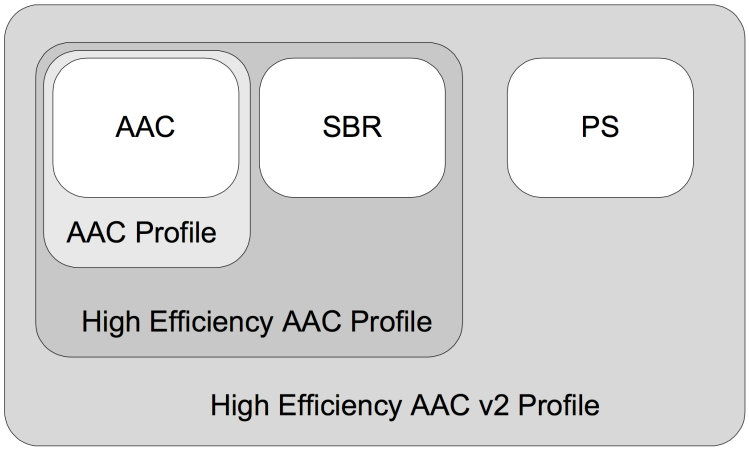

Figura 2.8: Perfis do Codificador AAC (Ref. (20))

Neste trabalho, estamos interessados no funcionamento específico da parte de decodificação do AAC-LC, implementando as ferramentas requeridas pela norma ABNT NBR 15602 de modo que a implementação possa ser facilmente expandida tanto para o padrão HE-AAC v1 quanto HE-AAC v2. A Figura 2.9 apresenta a sequência de etapas do Decodificador AAC-LC.

2.2.1 LATM/LOAS e MP4 *File Format*

A etapa inicial de decodificação consiste em desempacotar os dados de áudio que podem estar no formato de arquivo ADIF (Audio Data Interchange Format) envelopado no container MP4 ou no formato de streaming ADTS (Audio Data Transport Stream) envelopado no formato LATM/LOAS (LATM – Low-overhead MPEG-4 Audio Transport Multiplex / LOAS - Low Overhead Audio Stream), padrão para transmissão via internet ou pela TV Digital. Para cada um destes formatos, existe um algoritmo próprio que extrai os dados de áudio para a decodificação.

No formato de arquivo, os parâmetros gerais de decodificação do áudio são armazenados em um cabeçalho e os *raw data blocks* são armazenados em um espeço contíguo. No formato de transporte para transmissão (TS – Transport Stream), cada *raw data block* é empacotado com um cabeçalho próprio.

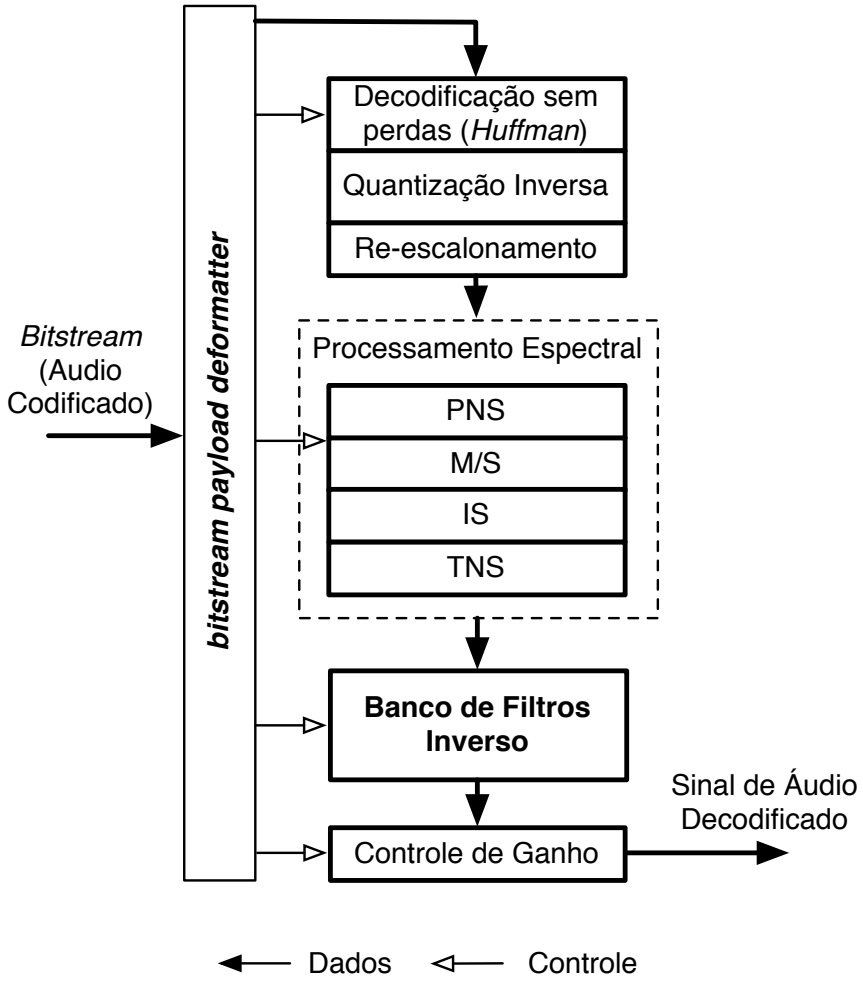

Figura 2.9: Diagrama de Blocos do Decodificador AAC-LC

2.2.2 *Bitstream Payload Deformater (Parser)*

A etapa inicial de decodificação consiste em separar os parâmetros definidos pelo codificador dos dados de áudio compactados. Esta tarefa é realizada por um *parser*, o *Bitstream Payload Deformater*.

O *bitstream* do áudio é organizado em blocos chamados de *raw data blocks*. Cada um destes blocos contém parâmetros de decodificação e dados de áudio compactados para todos os canais que estejam sendo transmitidos. Conforme citado anteriormente, o AAC suporta até 48 canais simultâneos, porém, no padrão do SBTVD somente são transmitidos áudios de 1 a 6 canais por bloco. Cada canal de um *raw data block* contém 1024 amostras de áudio organizadas em uma única janela longa ou em oito janelas curtas contendo 128 amostras cada. Deste modo, cada *raw data block* pode conter de 1024 (um canal) a 6144 (seis canais) amostras de áudio.

Sendo o *bitstream* de tamanho variável, a estrutura do *parser* consiste em uma árvore na qual, dependendo de cada parâmetro lido, segue-se um caminho diferente para a decodificação dos parâmetros e dos dados de áudio. A primeira classificação é feita pelo tipo de elemento sintático conforme a Tabela 2.4.

Tabela 2.4: Elementos Sintáticos dos *raw data blocks*

Id. do Elemento	**Código**	**Descrição**
SCE	000	*Single Channel Element*
CPE	001	*Channel Pair Element*
CCE	010	*Coupling Channel Element*
LFE	011	*Low Frequency Enhancement Channel*
DSE	100	*Data Stream Element*
PCE	101	*Program Config Element*
FIL	110	*Fill Element*
TERM	111	*End Element*

Cada um dos elementos sintáticos possui estrutura distinta exigindo um caminho próprio de decodificação. Descrevemos abaixo resumidamente a função e as especificidades de cada um deles.

1. **SCE** e **LFE**: Estes elementos são utilizados para armazenar dados de um único canal sendo o LFE específico para coeficientes espectrais de baixa frequência.

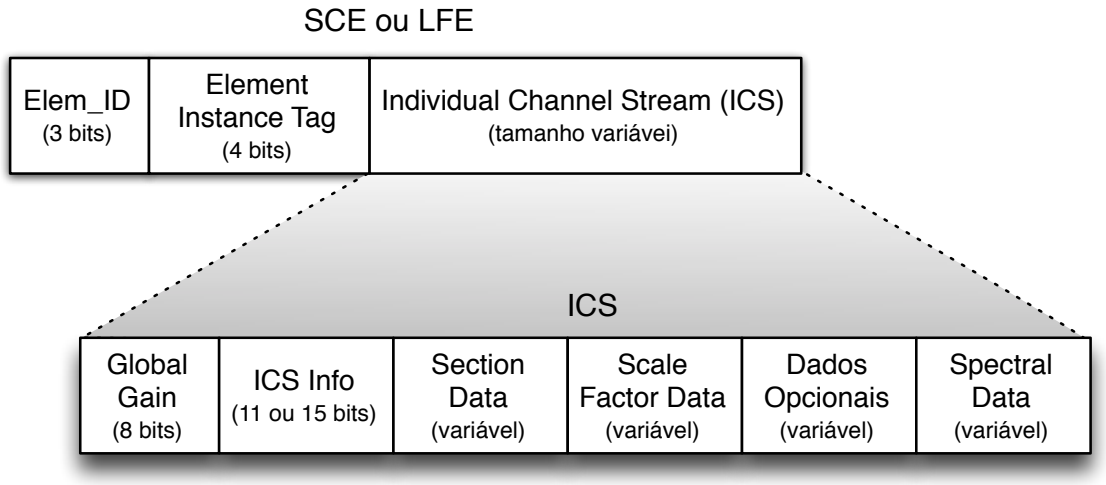

Figura 2.10: Diagrama dos Elementos Sintáticos SCE e LFE com detalhamento do ICS

Sua estrutura interna pode ser observada na Figura 2.10. Nela podemos observar que além do *ID* de 3 bits temos também o *Element Instance Tag* com 4 bits de largura que é o identificador único do canal. Em seguida, temos o *Individual Channel Stream* (ICS), estrutura de tamanho variável onde ficam armazenados os parâmetros de decodificação do canal como o *Global Gain*, que representa o valor do primeiro fator de escala, a estrutura ICS INFO, informando o modo como estão organizadas as janelas, o *Section Data* informando as tabelas Huffman usadas em cada trecho, o *Scale Factor Data*, os fatores de escala codificados com o algoritmo Huffman, os dados opcionais para o processamento espectral e finalmente o *Spectral Data*, contendo os coeficientes espectrais também codificados com o algoritmo Huffman.

2. **CPE**: Este elemento armazena dados de dois canais (stereo) podendo oferecer maior compactação quando ambos os canais possuem a mesma parametrização, *ICS Info*, que no caso é transmitida uma só vez. Esta opção é sinalizada pelo valor '1' em *Common Window* (Janela comum). Neste caso, um recurso adicional de processamento espectral, o *Mid/Side Coding* pode também ser utilizado e é sinalizado pelo campo *MS mask present*. Em caso positivo, seus parâmetros são encontrados em seguida em *MS used*. São ao todo dois elementos ICS, um para cada canal, que podem ou não conter os parâmetros de *ICS Info* caso o codificador não tenha utilizado janela comum.

CPE

Elem ID (3 bits)	Element Instance Tag (4 bits)	Common Window (1 bit)	ICS Info (11 ou 15 bits)	MS Mask Present (1 bit)	MS Used (variável)	Individual Channel Stream (ICS) (tamanho variável)	Individual Channel Stream (ICS) (tamanho variável)

Figura 2.11: Diagrama do Elemento Sintático CPE

3. **CCE**: Este elemento é formado pelo bloco ICS em conjunto com dados específicos para sua decodificação, seja ela o *Intensity Stereo* ou para mixagem com outro canal.

4. **DSE**: Este elemento é utilizado para transportar dados que não são parte do *stream* de áudio, não sendo utilizado no caso do SBTVD.

5. **CPE**: Este elemento é utilizado para transportar informações sobre a configuração dos canais e frequência de amostragem entre outros. No caso de *streaming* do SBTVD, o protocolo de transporte LATM já transporta estas informações.

6. **FIL**: Este elemento é utilizado tanto para dados do *Spectral Band Replication* (SBR) quanto para preencher o bitstream quando se objetiva manter uma taxa de bits constante.

7. **TERM**: Este é o elemento que define o final de um *raw data block*.

2.2.3 Decodificador sem Perdas (*Noiseless Decoder*)

Esta etapa consiste em recuperar tanto os valores dos Fatores de Escala quanto os dos Coeficientes Espectrais. Todos estes dados no caso do AAC são codificados pelo algoritmo *Huffman* que utiliza a probabilidade de ocorrência dos valores de entrada para determinar códigos de tamanho variável para representar os dados. Quanto maior a ocorrência do dado, menor será o tamanho de sua representação em bits.

No caso do AAC, tabelas pré-determinadas são utilizadas. Isto diminui a eficiência de compressão do algoritmo que normalmente cria a tabela de códigos durante a compactação, porém evita a necessidade de transmitir as tabelas juntamente com os dados de áudio, o que resulta em um ganho final.

Para a codificação dos fatores de escala, o padrão AAC determina uma única tabela fixa. Estes são utilizados pelo codificador para diminuir a representação em bits das frequências codificadas. As frequências são separadas por faixas ou bandas cujas larguras

se assemelham às Bandas Críticas do ouvido humano. Como a variação de magnitude entre as bandas é pequena, somente o valor do primeiro fator é transmitido e em seguida as diferenças entre eles são enviadas.

A decodificação dos **Fatores de Escala** presentes no elemento *Scale Factor Data* é, portanto, realizada a partir do primeiro valor, transmitido em *Global Gain*. Em seguida, o número de grupos de janelas (*num window groups*) e o número total de bandas (*max sfb*) são utilizados como referência para dois laços encadeados que varrem o vetor contendo (*sfb_cb - scale factor band codebook*) informações sobre como decodificar os fatores de escala.

A outra etapa da decodificação sem perdas é a dos **Coeficientes Espectrais**. Estes coeficientes são codificados em grupos de 2 ou 4 valores em que são utilizadas 11 tabelas. Mais uma vez, valores auxiliares como o número de grupos de janelas e a banda utilizada são usados como referência para os laços que irão acessar as tabelas e extrair os coeficientes.

2.2.4 Quantização Inversa (*Inverse Quantization*)

Após os coeficientes espectrais serem recuperados, a etapa seguinte é a de Quantização Inversa que utiliza um quantizador não-uniforme. Como em toda quantização, há um erro intrínseco ao processo que neste caso limita a representação dos coeficientes em 13 bits, ou seja, um valor absoluto máximo de 8191. Porém, a redução deste efeito é proporcionada justamente pela utilização dos fatores de escala que reduzem o tamanho da representação do coeficiente antes da quantização. Conforme a norma ISO/IEC 14496-3, a quantização inversa é descrita pela Equação 2.6 e utiliza laços encadeados para percorrer o vetor de coeficientes de acordo com o número de grupos de janelas, o número de bandas de fatores de escala e o tamanho de cada grupo. Na Equação, x_quant representa o valor do coeficiente espectral quantificado enquanto $x\_invquant$ representa o coeficiente de saída.

$$x\_invquant = sinal(x\_quant) \times |x\_quant|^{\frac{4}{3}} \tag{2.6}$$

2.2.5 Re-escalonador (*Rescaling*)

A etapa final de recuperação dos Coeficientes Espectrais é a do re-escalonador que utiliza os fatores de escala para multiplicar os coeficientes e os retornar à sua magnitude original. Neste caso o ganho a ser utilizado na multiplicação é obtido pela Equação 2.7 onde o valor de SF_OFFSET é uma constante de valor 100.

$$ganho = 2^{0,25 \times (scale\_factor - SF\_OFFSET)} \tag{2.7}$$

2.2.6 Processamento Espectral

Após recuperar os Coeficientes Espectrais, a próxima etapa de decodificação consiste na utilização de uma sequência de ferramentas espectrais para modificar as amostras de áudio ainda no domínio da frequência de acordo com os parâmetros de decodificação. O conjunto de módulos que compõem estas ferramentas é composto por: PNS - Perceptual Noise Substitution, M/S – Mid-Side Stereo, Intensity Stereo e TNS – Temporal Noise

Shaping. A utilização destas ferramentas por parte do codificador AAC é opcional, porém é requerida de acordo com a norma ABNT NBR 15602 (1).

PNS - *Perceptual Noise Substitution*

A ferramenta *Perceptual Noise Substitution* (PNS) explora a característica dos ruídos aleatórios de poder simular os coeficientes de trechos de subbandas ruidosas. Neste caso, o codificador identifica os trechos de sinal reconhecidos como ruído para cada uma das bandas dos fatores de escala e os agrupa em diferentes categorias. Os coeficientes espectrais destas categorias são omitidos do processo de codificação e em troca, um flag informando a substituição por ruído é habilitado e transmitido juntamente com o valor total da energia total dos mesmos para cada banda. Desse modo, diminui-se a complexidade da etapa de decodificação evitando o processamento destes coeficientes no codificador sem perdas.

O processo de decodificação neste caso é realizado a partir de um gerador de números pseudo-aleatórios que calcula coeficientes na quantidade equivalente à referida banda de fatores de escala. Em seguida, a energia equivalente destes novos coeficientes é calculada e a partir dela é gerado um valor de ganho a ser aplicado a cada coeficiente. Finalmente, os valores dos coeficientes já com o ganho são inseridos nas posições correspondentes no vetor de coeficientes espectrais.

Joint Stereo Coding

O *Joint Stereo Coding* ou codificação estéreo conjunta utiliza um conjunto de técnicas que se baseiam nas semelhanças entre os canais esquerdo e direito, nos casos de som estéreo, para diminuir o tamanho da representação de bits do áudio. Esta semelhança pode ser observada não no domínio do tempo mas sim no domínio da frequência e pode ser acentuada dependendo da resolução temporal do banco de filtros.

No caso do MPEG-4 AAC, duas técnicas são utilizadas, o *Mid/Side Stereo* (MS) e o *Intensity Stereo* (IS).

M/S – *Mid-Side Stereo*

O Mid/Side Stereo explora o fato de que quando um par de canais tem suas frequências muito próximas, a diferença entre os valores de seus coeficientes espectrais será mínima. Neste caso, a técnica consiste em transformar a soma dos canais esquerdo e direito em um só canal central (Mid) e a diferença entre eles em um outro canal (Side). Esta é uma técnica de compactação sem perda de informações.

$$\begin{aligned} m &= L + R \\ s &= L - R \end{aligned} \tag{2.8}$$

Na etapa de decodificação, a ferramenta M/S é utilizada quando os dois canais usam a mesma configuração de janela, ou seja, quando o parâmetro *common window* é igual a '1'. Durante o processamento, que varre todo o vetor de coeficientes espectrais, a matriz inversa apresentada na Equação 2.9 é utilizada sempre que o parâmetro *MS_used* for igual a '1'. Neste caso, l e r são os valores desejados para os canais esquerdo e direito

respectivamente e são obtidos pelos coeficientes m (mid) e s *side*. Assim, obtém-se $2l$ ou $2r$ e utiliza-se uma operação de shift para recompor o valor original.

$$\begin{bmatrix} l \\ r \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} m \\ s \end{bmatrix} \tag{2.9}$$

IS - *Intensity Stereo*

A ferramenta IS se baseia na incapacidade do ouvido humano de perceber a localização da origem de um som acima de uma certa frequência. Portanto, a técnica consiste em combinar frequências próximas em uma só no momento da codificação e transmitir um só coeficiente espectral juntamente com parâmetros de lateralidade para que na etapa de decodificação possa ser recuperado o lado direito e esquerdo, mesmo sendo diferentes do original. Esta é uma ferramenta que gera perdas na qualidade original e geralmente é empregada para taxas de bits mais baixas.

Similar ao MS, a decodificação do IS só é utilizada quando ambos os canais utilizam a mesma configuração (*common window* = '1'). A codificação IS pode ser em fase ou fora de fase de acordo com o tipo de *Scale Factor Band Codebook* (sfb_cb) indicado nos parâmetros de codificação. Quando habilitada, informações de decodificação (intensity stereo position) são transmitidas pelo canal direito no lugar dos fatores de escala. A partir desta informação, um valor de ganho é calculado para ser aplicado ao valor do coeficiente espectral do canal direito.

TNS – *Temporal Noise Shaping*

A ferramenta *Temporal Noise Shaping* (TNS) (16), é utilizada para controlar a forma temporal do ruído na quantização de cada janela de áudio, atuando nas distorções de pré-eco causadas pela baixa resolução temporal do banco de filtros. Os efeitos de pré-eco ocorrem geralmente quando um sinal apresenta uma transição abrupta após uma região de baixa energia, ou mais silenciosa. Um exemplo seriam as batidas de instrumentos de percussão (29).

A técnica envolve o uso de um filtro que conforma a onda de áudio para evitar picos indesejados dentro de uma mesma janela. Para isto, a predição linear é aplicada no domínio da frequência. Conforme pode ser observado no diagrama da Figura 2.12, parâmetros de um Filtro Preditivo Linear ($A(z)$) são estimados para os coeficientes espectrais $X(k)$ e a saída $e(k)$ é quantizada e codificada utilizando o codificador perceptual.

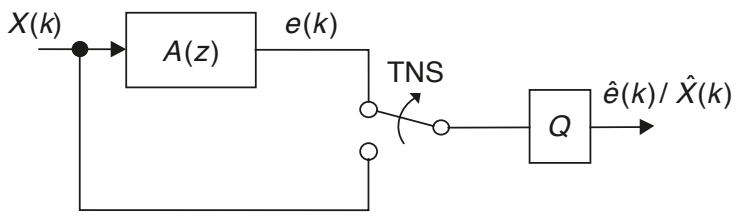

Figura 2.12: Esquema do TNS para controle dos efeitos de pré-eco ((Ref. (29))

Em seguida, os coeficientes do filtro quantizados são transmitidos como parâmetros no *bitstream* para que o mesmo sinal possa ser reconstituído pelo decodificador. A aplicação do TNS na decodificação constitui no cálculo dos coeficientes LPC (*Linear Predictive Coding*) e da posterior aplicação do filtro inverso utilizando os coeficientes LPC.

O algoritmo do TNS, aplicado individualmente para cada canal, utiliza dois Laços Externos, um com o número de janelas (*num_windows*) e outro com *tns.n_filt* para varrer os parâmetros específicos dedicados ao TNS presentes na estrutura ICS. A operação interna consiste em calcular os coeficientes LPC, em seguida calcular o tamanho do filtro (a quantas amostras de áudio o mesmo será aplicado) a partir das variáveis *max_tns_sfb* e *swb_offset*, e por fim aplicar o filtro em um Laço Interno.

Um exemplo do uso do TNS é apresentado na Figura 2.13.

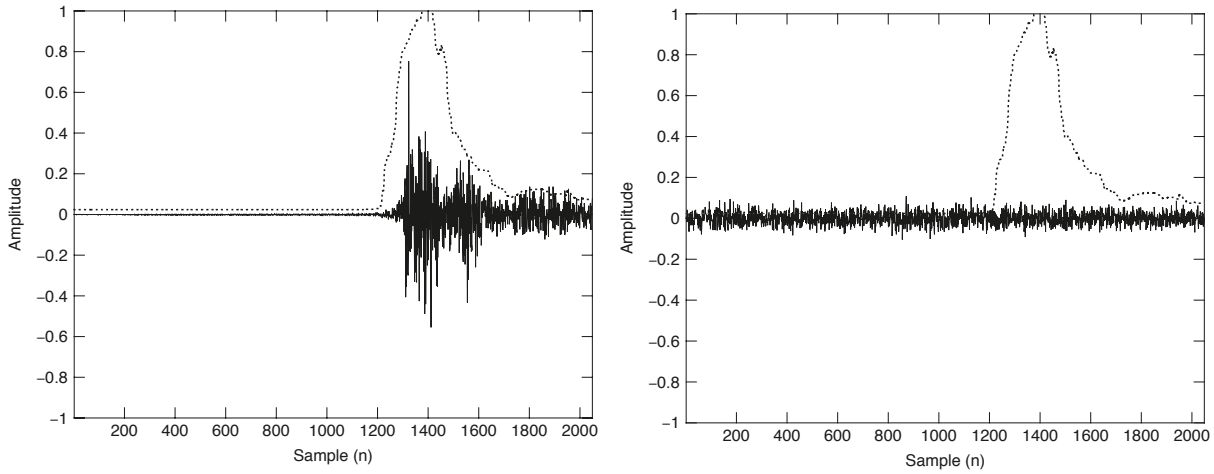

Figura 2.13: Exemplo de onda sonora ressaltando a diferença entre a codificação com TNS (esquerda) e sem TNS (direita). (Ref. (29))

2.2.7 Banco de Filtros (*Filterbank*) e *Block Switching*

Após passar pelo processamento espectral, o Banco de Filtros é a última etapa de processamento. Sua função é converter os coeficientes espectrais no domínio da frequência de volta em amostras de áudio no domínio do tempo. No caso do AAC, este Banco de Filtros é implementado utilizando a *Modified Discrete Cosine Transform* (MDCT) em sua etapa de codificação (análise) e o processo inverso, a *Inverse Modified Discrete Cosine Transform* (MDCT) para a decodificação (síntese).

Para melhorar a resolução temporal, o AAC emprega dois tamanhos de janela, uma longa com 2048 amostras de áudio e uma curta com 256 amostras. As janelas longas são mais adequadas para trechos de áudio com onda mais estacionária e ajudam a aumentar a capacidade de compressão do áudio. Já as janelas curtas são usadas para trechos de áudio onde ocorrem mudanças mais abruptas, melhorando a qualidade do áudio e ajudando na redução de efeitos de pré-eco.

Segundo Spanias (29), este modelo de Banco de Filtros possui características importantes como a reconstrução perfeita, amostragem crítica, eliminação dos artefatos entre

blocos, baixa complexidade, possibilidade de implementação por algoritmos rápidos, entre outras. Por estes motivos, é amplamente empregado nos atuais codificadores de áudio.

A MDCT direta é apresentada pela Equação 2.10 onde N é o número de amostras (2048 para janelas longas e 256 para curtas) e são gerados N/2 coeficientes no domínio da frequência. Já a Equação 2.11 apresenta o procedimento inverso, o da IMDCT onde os N/2 coeficientes são convertidos novamente em N amostras de áudio.

$$X(k) = \sum_{n=0}^{N-1} x(n) h_k(n), \;\; 0 \leq k \leq \frac{N}{2} - 1 \tag{2.10}$$

$$x(n) = \sum_{k=0}^{N/2-1} \left[X(k) h_k(n) + X^P(k) h_k \left(n + \frac{N}{2} \right) \right] \tag{2.11}$$

Conforme ressaltado por Spanias (29), a sobreposição de 50% da janela anterior com 50% da janela atual é utilizada neste processo, praticamente eliminando os artefatos oriundos da mudança entre os blocos. Este procedimento completo pode ser observado no diagrama da Figura 2.14.

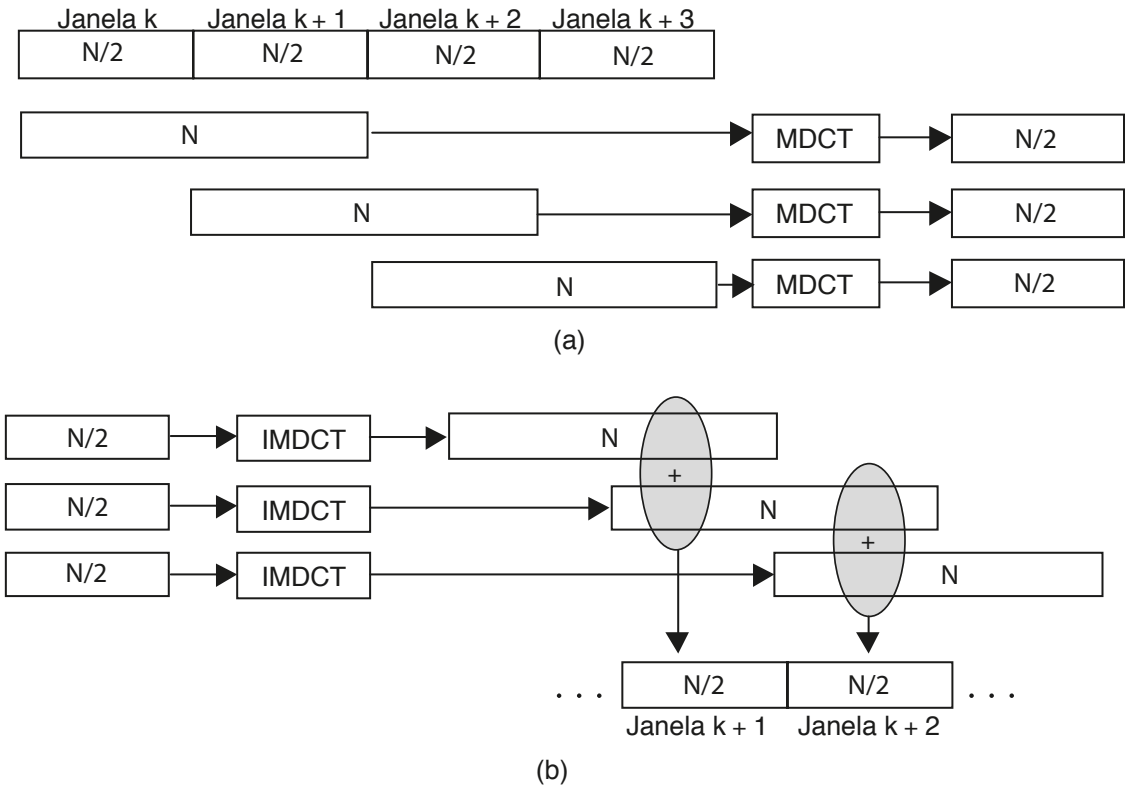

Figura 2.14: Diagrama do Processamento das Janelas do Banco de Filtros. Em (a) temos a etapa de codificação convertendo N amostras em grupos de N/2 coeficientes espectrais. Em (b) temos a etapa de decodificação convertendo os N/2 coeficientes espectrais em N amostras e aplicando o procedimento de sobreposição e adição (*Overlap and Add*). (Ref. (29))

Além da sobreposição de janelas, duas outras técnicas são empregadas no Banco de Filtros do AAC para manter a boa qualidade de áudio e evitar artefatos na transição entre janelas. A primeira delas é a utilização de duas funções distintas para janelas, a Senoidal e a *Kaiser-Bessel Derived* (KDB). A segunda técnica consiste na variação do formato da janela visando amenizar distorções entre os diferentes tipos e tamanhos de janela.

Em termos de função, a janela Senoidal possui uma banda de passagem mais estreita com pouca atenuação na região de rejeição enquanto a função KBD apresenta banda de passagem mais larga e maior atenuação em sua faixa de rejeição. A escolha da função é definida pelo codificador e transmitida ao decodificador por meio do parâmetro *window shape*.

Visando manter consistência entre as funções utilizadas nas janelas para que a reconstrução seja perfeita, o Banco de Filtros divide as funções Senoidal e KBD em duas partes (direita e esquerda) e controla sua aplicação armazenando a informação sobre a função utilizada na janela anterior para que a mesma seja usada na primeira metade da próxima janela.

A função Senoidal é caracterizada pelas Equações 2.12 e 2.13.

$$w_{sen\_esq}(n) = sen\left(\frac{\pi}{N}\left(n+\frac{1}{2}\right)\right),\ 0 \leq n < \frac{N}{2} \tag{2.12}$$

$$w_{sen\_esq}(n) = sen\left(\frac{\pi}{N}\left(n+\frac{1}{2}\right)\right),\ \frac{N}{2} \leq n < N \tag{2.13}$$

Já o janelamento utilizando a função KDB é representado nas Equações 2.14, 2.15, e o núcleo da função Kaiser-Bessel é apresentado em 2.16 e 2.17, conforme a norma (ISO/IEC, 2005). O valor de α vale 4 para janelas longas (N = 2048) e 6 para janelas curtas (N = 256).

$$w_{KBD\_ESQ,N}(n) = \sqrt{\frac{\sum_{p=0}^{n}\left[W'(p,\alpha)\right]}{\sum_{p=0}^{N/2}\left[W'(p,\alpha)\right]}} \quad para\ 0 \leq n < \frac{N}{2} \tag{2.14}$$

$$w_{KBD\_DIR,N}(n) = \sqrt{\frac{\sum_{p=0}^{N-n-1}\left[W'(p,\alpha)\right]}{\sum_{p=0}^{N/2}\left[W'(p,\alpha)\right]}} \quad para\ \frac{N}{2} \leq n < N \tag{2.15}$$

$$W'(n,\alpha) = \frac{I_0\left[\pi\alpha\sqrt{1-\left(\frac{n-N/4}{N/4}\right)^2}\right]}{I_0\left[\pi\alpha\right]} \quad para\ 0 \leq n \leq \frac{N}{2} \tag{2.16}$$

$$I_0\left[x\right] = \sum_{k=0}^{\infty}\left[\frac{\left(\frac{x}{2}\right)^k}{k!}\right]^2 \tag{2.17}$$

Uma comparação entre ambas as funções Senoidal e KBD pode ser vista na Figura 2.15.

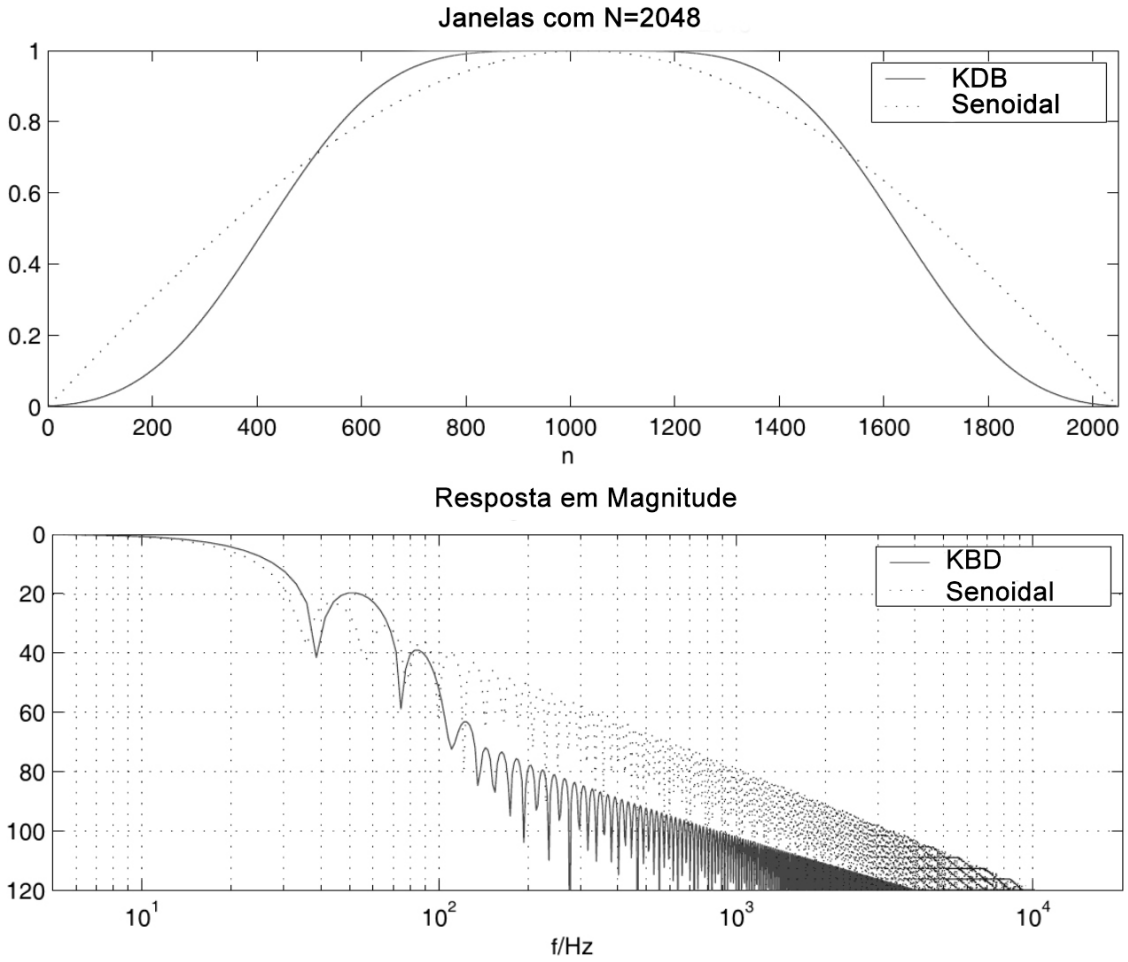

Figura 2.15: Funções Senoidal em KBD para janelas longas (Ref. (38)).

O Banco de Filtros do AAC aplica 4 formas de janelas: *Only Long Sequence*, *Eight Short Sequence*, *Long Start Sequence* e *Long Stop Sequence*. Os dois últimos formatos são usados para fazer a transição entre as janelas longas e curtas. A utilização de cada um destes formatos é controlada pelo parâmetro *window sequence*. Tanto o formato da janela atual quanto da janela anterior são usados pelo decodificador para efetuar a transição entre elas. Além disso, conforme citado anteriormente, a função aplicada a cada parte da janela depende da função da janela anterior, ou seja, do valor do *window shape* atual e anterior.

No caso das janelas longas no formato *Only Long Sequence* temos 2.18:

$$w(n) = \begin{cases} w_{shape\_anterior,N=1024}(n), & \text{para } 0 \leq n < 1024 \\ w_{shape\_atual,N=1024}(n), & \text{para } 1024 \leq n < 2048 \end{cases} \tag{2.18}$$

As janelas curtas são do formato *Eight Short Sequence* apesentado pelas Equações 2.19 e 2.20, enquanto a função de sobreposição e soma (Overlap and Add) interna é

apresentada pela Equação 2.21. Note que a função da janela anterior só afeta a primeira das oito janelas curtas (8 * 256 = 2048).

$$w_0(n) = \begin{cases} w_{shape\_anterior,N=256}(n), & \text{para } 0 \leq n < 128 \\ w_{shape\_atual,N=256}(n), & \text{para } 128 \leq n < 256 \end{cases} \tag{2.19}$$

$$w_{1\text{ a }7}(n) = \begin{cases} w_{shape\_atual,N=256}(n), & \text{para } 0 \leq n < 128 \\ w_{shape\_atual,N=256}(n), & \text{para } 128 \leq n < 256 \end{cases} \tag{2.20}$$

$$Z_{i,n} = \begin{cases} 0, & \text{para } 0 \leq n < 448 \\ x_{0,n-448} \cdot w_0(n-448), & \text{para } 448 \leq n < 576 \\ x_{j-1,n-(128j+320)} \cdot w_{j-1}(n-(128j+320)) & \\ +x_{j,n-(128j+448)} \cdot w_j(n-(128j+448), & \text{para } 1 \leq j < 8 \\ & \text{e } \ 128j+448 \leq n < 128j+576 \\ x_{7,n-1344} \cdot w_7(n-1344), & \text{para } 1472 \leq n < 1600 \\ 0, & \text{para } 1600 \leq n < 2048 \end{cases} \tag{2.21}$$

No caso das janelas de transição *Long Start Sequence* e *Long Stop Sequence* temos as Equações 2.22 e 2.23 respectivamente.

$$w(n) = \begin{cases} w_{shape\_anterior,N=2048}(n), & \text{para } 0 \leq n < 1024 \\ 1, & \text{para } 1024 \leq n < 1472 \\ w_{shape\_atual,N=256}(n), & \text{para } 1472 \leq n < 1600 \\ 0, & \text{para } 1600 \leq n < 2048 \end{cases} \tag{2.22}$$

$$w(n) = \begin{cases} 0, & \text{para } 0 \leq n < 448 \\ w_{shape\_anterior,N=256}(n-448), & \text{para } 448 \leq n < 576 \\ 1, & \text{para } 576 \leq n < 1024 \\ w_{shape\_atual,N=2048}(n), & \text{para } 1024 \leq n < 2048 \end{cases} \tag{2.23}$$

Para facilitar a compreensão, apresentamos na Figura 2.16 as quatro formas de janela utilizando a função senoidal.

A última etapa do Banco de Filtros é de sobreposição e adição das amostras com a metade da janela anterior, resultando, assim, nas amostras finais no domínio do tempo. Com isso, temos na saída do Banco de Filtros as 2048 amostras de áudio no domínio do tempo.

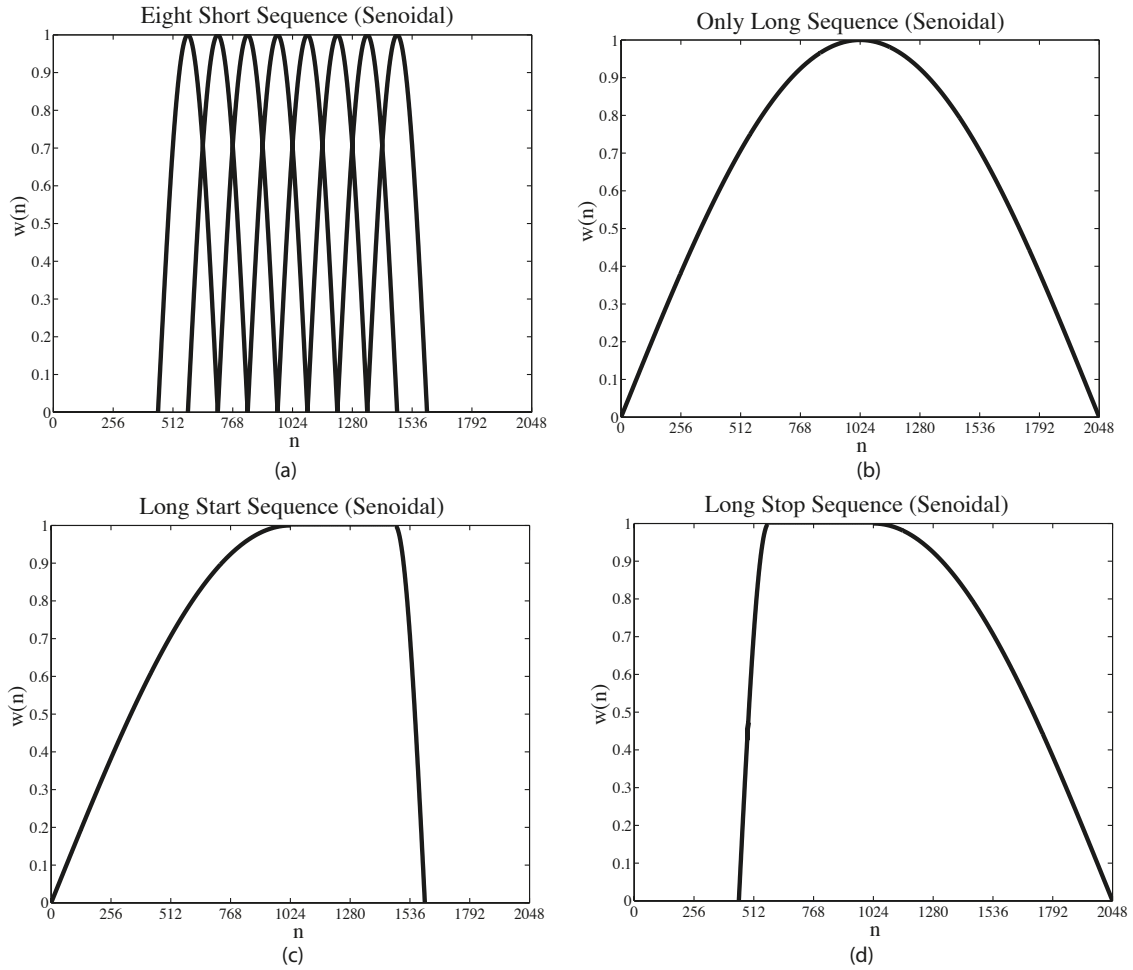

Figura 2.16: Representação das quatro formas de janela do Banco de Filtros (a) Eight Short Sequence, (b) Only Long Sequence, (c) Long Start Sequence, (d) Long Stop Sequence.

2.3 Coprojeto

Dentre as possíveis abordagens de implementação existem pelo menos 3 opções mais utilizadas para o desenvolvimento de aplicações em sistemas embarcados e sistemas dedicados. A primeira abordagem é a solução em software desenvolvida e otimizada para um processador embarcado. Esta, geralmente é mais rápida para ser implementada e atende uma grande parte de aplicações com mais baixo requisito computacional. Porém, para aplicações mais exigentes do ponto de vista de esforço computacional, a necessidade de acelerar a frequência dos processadores gera maior consumo de energia.

A segunda abordagem que vai ao outro extremo é a do desenvolvimento de toda a solução hardware dedicado. Esta abordagem normalmente fornece o maior desempenho e o menor consumo de energia possível, porém a custo de um tempo de desenvolvimento muito maior e de uma solução fixa com poucas opções de ajustes ou reconfiguração.

A terceira abordagem utilizada no caso de soluções com necessidade de grande esforço computacional é uma solução híbrida denominada coprojeto entre hardware e software. Neste caso, o perfil e a exigência computacional de cada parte da solução é avaliada sepa-

radamente e somente as funções mais críticas são implementadas em hardware dedicado, mantendo o restante em software. Assim, a solução é executada em uma arquitetura que integra um processador a módulos dedicados em hardware. Esta abordagem permite que a solução mantenha flexibilidade, acelere o processamento e mantenha o consumo de energia baixo. Além disso, o coprojeto permite que a solução seja implementada de maneira gradual partindo-se da solução em software, desenvolvendo cada módulo em hardware e testando o mesmo tanto para o desempenho esperado quanto para a conformidade com os resultados do algoritmo original.

A Tabela 2.5 faz um resumo da comparação entre as três abordagens.

Tabela 2.5: Configuração de canais - MPEG-4

	Consumo de Energia	Esforço de Implementação	Flexibilidade da Solução
Algoritmo em software	Alto	Baixo	Alto
Coprojeto HW/SW	Médio	Médio	Médio
Algoritmo em hardware	Baixo	Alto	Baixo

Ressalta-se ainda que a solução almejada para a decodificação de áudio possui um requisito de desempenho máximo baseado no esforço necessário para decodificar o áudio em tempo real. Portanto, não estamos buscando a solução mais rápida possível mas sim uma que atenda os requisitos de tempo real, que possua flexibilidade para ser expandida e que tenha um baixo consumo de energia.

2.4 FPGA

De acordo com a metodologia adotada, a validação do projeto passa por uma etapa de prototipação em arquiteturas reconfiguráveis. No caso, escolhemos a tecnologia FPGA (Field-programmable gate array) em que é possível programar um circuito lógico e depurar sua arquitetura até alcançarmos o desempenho e a área adequados. Neste caso, o FPGA pode ser utilizado tanto como tecnologia fim para a implementação do circuito desejado como meio para a validação do projeto a ser futuramente fabricado em um SoC (*System-on-a-Chip*).

Os FPGAs são circuitos integrados compostos por células lógicas que podem ser programadas para executar funções lógicas, armazenar dados e efetuar roteamento de sinais para outras células lógicas. Sua programação é realizada através de linguagens de descrição de hardware HDL (*Hardware Description Language*) sendo as mais comuns o VHDL (*VHSIC Hardware Description Language*) e Verilog.

Atualmente estão presentes no mercado diversos fabricantes de FPGA incluindo Achronix Semiconductor, Actel, Altera, AMI Semiconductor, Atmel, Cypress Semiconductor, Lattice Semiconductor, QuickLogic, e Xilinx. Cada fabricante oferece dispositivos com diversos tamanhos em termos do número total de portas lógicas, memórias embarcadas e elementos de processamento DSP embutidos.

Em relação aos processadores embarcados podemos observar exemplos dos dois maiores fabricantes tendo a Altera com o Nios II que é um *Soft Processor* ou *softcore* (processador configuráveis que utiliza os elementos lógicos do FPGA) e a Xilinx com o *Soft Processor* MicroBlaze. Também estão disponíveis processadores *hardcore* como o ARM na família Cyclone V da Altera, o que torna soluções de coprojeto mais eficientes.

Apesar de historicamente os FPGAs oferecerem um número limitado de células lógicas, baixa frequência e maior consumo de energia comparado às soluções em hardware dedicado (ASIC), temos anúncios recentes como o Virtex-7 da Xilinx (36) com até 2.000.000 de células lógicas fabricado com tecnologia de 20nm e o Stratix 10 da Altera (6) com 4.000.000 de células lógicas fabricado com tecnologia de 14nm podendo alcançar frequência de até 1 GHz e capaz de reduzir em até 70% o consumo de energia em comparação com dispositivos anteriores. Estas características, tornam estes dispositivos cada vez mais atrativos para uso em larga escala.

2.5 Revisão de literatura

Ao se pesquisar a literatura geral sobre codificação de áudio digital, estudou-se o artigo *Perceptual Coding of Digital Audio* (26) e o livro *Audio Signal Processing and Coding* (29) dos mesmos autores que inicialmente fazem uma introdução geral sobre a codificação de áudio e explicam suas principais características e atributos juntamente com os diversos tipos e técnicas utilizadas nos decodificadores. O capítulo 2 do livros apresenta as bases matemáticas e os conceitos essenciais sobre processamento de sinais. No capítulo 3 temos as diversas técnicas de quantização e codificação de entropia. No capítulo 5, são apresentados os princípios da psicoacústica. No capitulo 6 e 7, foram estudados os modelos de transformada do domínio do tempo para o domínio da frequência juntamente com os conceitos de Bancos de Filtros. No capítulo 10, diversos padrões de codificação de áudio, incluindo o AAC são discutidos juntamente com seus algoritmos e finalmente no capítulo 12 são apresentadas as técnicas para a medição da qualidade de áudio.

Além destes, também foi estudado o livro *Digital Audio Signal Processing* (38) que aborda assuntos gerais de codificação de áudio e em seu capítulo 9, fala especificamente dos conceitos envolvendo a codificação de áudio digital e os codificadores do padrão MPEG-1 e MPEG-2 explicando as diversas etapas de codificação juntamente com o funcionamento das ferramentas de processamento espectral.

Para a implementação da solução na plataforma FPGA foram consultados os manuais específicos do ambiente de desenvolvimento da Altera sobre o software Quartus II (5), sobre o FPGA da família Cyclone II (3), sobre o processador Nios II (4) e sobre o barramento Avalon em (2).

Os detalhes específicos do padrão AAC foram estudados a partir das normas ABNT NBR 15602 (1), norma brasileira que define o uso do AAC no padrão de Televisão Digital do Brasil com seus perfis obrigatórios e opcionais, da norma internacional ISO/IEC 13818-7 (18) que define o padrão AAC do MPEG-2 e da norma ISO/IEC 14496-3 (20) que define o AAC para o MPEG-4.

Praticamente todos as etapas da decodificação AAC estão descritas nestas normas, porém, para melhor compreender os desafios e detalhes sobre sua implementação, recorremos a literatura específica. O uso do métodos *Huffman* tanto para a etapa de decodificação dos fatores quanto para a decodificação dos coeficientes espectrais foi aprofundada pelo

artigo (7). As etapas de *Joint Stereo* do processamento espectral foram estudadas em (17). A implementação da ferramenta *Temporal Noise Shaping* foi estudada a partir do artigo original (16) que propôs o método TNS em que são discutidos os problemas decorrentes do mascaramento temporal como os efeitos de pré-eco que podem ser minimizados através da técnicas TNS.

As implementações da MDCT foram estudadas a partir de 5 artigos. Em dois casos (8, 25) são apresentadas estruturas baseadas em recursão em que o mesmo elemento de processamento é reutilizado diversas vezes com o intuito de reduzir ao máximo o tamanho do hardware necessário para o cálculo. Este casos, o número de ciclos necessários para processar um entrada de 1024 pontos chega a 156911. Em seguida temos Li et al. (22) que estuda algoritmos eficientes para implementar arquiteturas tanto da MDCT quanto da IMDCT em FPGA, de modo a reaproveitar os módulos para ambos os sentidos das transformadas. Ao invés de utilizar o algoritmo Radix-2, utiliza-se um algoritmo de N/8-pontos em que se reduz em até 50% o número de ciclos para 65538 com 7 multiplicadores e 11 somadores em hardware e mais memória. Em (28), Wu e Hwan implementam a DCT com o banco de filtros com grande eficiência utilizando 3 multiplicadores e 3 somadores e alcançam um processamento de 15360 ciclos para a janela de 1024 pontos. Em um quinto artigo, Du et al. (10) trata da implementação do Banco de Filtros Inverso como um todo incluindo a IMDCT para o decodificador AAC e consegue organizar as etapas de IMDCT, do Janelamento e Sobreposição de modo a utilizar 4 multiplicadores e 6 somadores e reduzir o número total de ciclos de 15360 para 12288 a custo de mais 1 multiplicador, 3 somadores e 15% mais de área de memória.

Pesquisando a literatura relacionada à implementação completa do AAC, encontramos diferentes abordagens para alcançar um arquitetura eficiente. Algumas soluções focam o aprimoramento do algoritmo em software para que o mesmo exija um menor esforço do processador. Outras abordagens focam a implementação puramente em hardware reduzindo ao máximo o consumo de energia. Além disso, temos abordagens de coprojeto combinando partes do algoritmo em software e outra parte em hardware dedicado.

Um exemplo de abordagem que foca a otimização de software é apresentada por Takamizawa et al.(30), que implementou uma solução puramente em software para o processador RISC de baixo consumo NEC V830 com capacidade de 158 MIPS. Nesta solução, são propostos métodos para acelerar os cálculos da IMDCT explorando a arquitetura específica do processador que possui memória cache limitada a 4 kB. Três técnicas combinadas permitiram uma redução de 42% do uso do processador. A primeira foi a utilização de uma memória RAM interna ao processador de 4 kB, exatamente o tamanho necessário para armazenar os coeficientes espectrais na IMDCT, que permite reduzir a zero o *cache miss* do processador. A segunda técnica foi a redução do tamanho dos coeficientes *twiddle* de 32 para 16 bits, que também reduz o tempo de acesso à memória. A terceira técnica envolveu o armazenamento dos coeficientes em seu modo diferencial, o que permitiu uma redução de 32 para 8 bits, também otimizando tempo de acesso à memória. Finalmente, temos um algoritmo que implementa um decodificador AAC-LC de nível 4 capaz de decodificar um áudio 5.1 a 432 kbps a uma frequência de 133 MHz que consome 300mV.

Tsai, Liu e Wang (34) implementam uma solução pura em ASIC para o decodificador AAC-LC de dois canais. A solução explora uma arquitetura que paraleliza o máximo possível de operações entre os canais. Neste caso, se alcança a decodificação em tempo

real a 3,3 MHz utilizando uma área de 3x3mm^2 com consumo de 158 mW na tecnologia TSMC de 0,25 μm. O número total de Logic Gates é de 82,2k porém, a quantidade de memória utilizada não é reportada. Em 2009 (32), uma nova proposta de *low-power* foi feita na qual se utilizam 102k *Logic Gates* em que é possível decodificar um áudio de dois canais a 1,3MHz que consome apenas 2,45mW. Desta vez, a tecnologia UMC 0,18 μm é utilizada ocupando a mesma área de 3x3mm^2.

Uma outra solução específica para o decodificador AAC-LC para dois canais utilizando a abordagem de coprojeto é apresentada por Liu e Tsai (23) na qual a plataforma base é composta por um processador ARM (ARM920T) em silício acoplado a um FPGA Xilinx (VertexE XCV2000E FG 680). Durante o desenvolvimento observou-se a necessidade de 175 MHz para atingir os requisitos de tempo real para dois canais na solução em software. O módulo crítico escolhido para implementação em hardware foi o Banco de Filtros, cuja comunicação foi feita por meio de um barramento AMBA. Neste caso, alcançou-se a decodificação em tempo real com o processador a 41 MHz e o módulo em hardware a 22,8 MHz. Em 2008 (35), a mesma equipe apresentou uma nova solução de coprojeto para os padrões AAC, AC3 e MP3 utilizando novamente o ARM acoplado a um Banco de Filtros em hardware. Desta vez, alcança-se a decodificação em tempo real a 5 MHz.

Ainda utilizando o coprojeto, temos a abordagem de Zhou et al. (37) implementada em um SoC que incorpora um processador OpenRISC acoplado a um Banco de Filtros reconfigurável baseado no CORDIC (*Coordinate Rotation Digital Computer*) capaz de ser utilizado para os padrões AAC, MP3, WMA, AC3 e Vorbis. Neste caso, são necessários 44,3 k *Logic Gates* e um total de 78 kB de memória. A decodificação de áudio AAC em tempo real para um alta qualidade de 256 kpbs ocorre a aproximadamente 16 MHz.

Em Tao et al. (31) uma solução de coprojeto é proposta para decodificar o AAC-LC de dois canais em conjunto com o MP3. Neste caso, tanto a decodificação Huffman quanto a IMDCT são implementadas em hardware. A implementação é realizada em FPGA na plataforma XUP VIIP da Xilinx com um processador PowerPC embarcado. A decodificação em tempo real só é alcançada, neste caso, com o processador a 300 MHz para um áudio de dois canais a 118 kbps.

O trabalho de Renner (27) apresenta uma arquitetura simplificada em FPGA para o Decodificador AAC-LC de dois canais. Neste caso, temos uma solução toda em hardware porém, a mesma não contempla nenhuma das ferramentas de processamento espectral ou mesmo a decodificação dos protocolos LATM/LOAS ou MP4. No total são utilizados 26.549 elementos Lógicos com 248.704 bits de memória sendo capaz de decodificar um áudio de dois canais a 128 kbps em tempo real com o relógio a 4 MHz.

A Tabela 2.6 apresenta um resumo das soluções citadas buscando colocar em perspectiva as diferentes abordagens com suas semelhanças e diferenças. Como podemos observar, temos uma solução em software, três soluções puramente em hardware sendo duas delas em ASIC e uma em FPGA. Além disso, temos três soluções que utilizam a técnica de coprojeto de hardware e software, uma delas implementa um hardware dedicado em silício e duas utilizam processadores VLSI combinados com módulos em hardware desenvolvidos em FPGA.

Tabela 2.6: Comparação das diversas soluções encontradas na literatura

Referência	**Takamizawa et al. (30)**	**Tsai, Liu e Wang (34)**	**Tsai e Liu (32)**	**Renner (27)**	**Liu and Tsai (23)**	**Zhou et al. (37)**	**Tao et. al. (31)**
Arquitetura da Solução	Software + 2 Áreas de memória RAM	Hardware-Puro	Hardware-Puro	Hardware-Puro	Coprojeto HW/SW	Coprojeto HW/SW	Coprojeto HW/SW
Decodificador Implementado	MPEG-2 AAC-LC	MPEG-2 AAC-LC	MPEG-2 AAC-LC	MPEG-2 AAC-LC	MPEG-2 AAC-LC	MPEG-2 AAC-LC / MP3	MPEG-2 AAC-LC / MP3
Tecnologia	Processador de Propósito Geral	ASIC (TSMC 0,25μm)	ASIC (UMC 0,18μm)	FPGA (Cyclone II Altera)	VLSI / FPGA (Xilinx VertexE)	VLSI	FPGA (Xilinx XUP V2P
Processador	RISC NEC V380	-	-	-	ARM (ARM920T)	OpenRisc 1200	PowerPC
Desempenho do Processador	158 MIPS	-	-	-	60 a 200 MIPS	250 MIPS @ 250MHz	N/A
Freq. do Processador	133 MHz	-	-	-	41 MHz	10,6 MHz	300 MHz
Freq. do Hardware	-	3,3 MHz	1,3 MHz	4 MHz	22,8 MHz	10,6 MHz	N/A
N. de Canais	6	2	2	2	2	2	2
Tx. Amost. Áudio	48 kHz	44,1 kHz	44,1 kHz	48 kHz	44,1 kHz	44,1 kHz	48 kHz
Bitrate	432 kbps	128 kbps*	128 kbps*	128 kbps*	128 kbps*	128 kbps*	118 kbps
Power	300 mW	158 mW	2,45 mW	N/A	N/A	N/A	N/A
LE (FPGA) / LG (ASIC)		82,2 k LG	102 k LG	26549 LE (248704 bits de memória)	6266 Slices	44,3 k LG + 16 k RAM + 4 k ROM	N/A

Capítulo 3

Desenvolvimento do AAC

3.1 Código de Referência em C

O trabalho de desenvolvimento do decodificador AAC teve início a partir do estudo e validação do código C desenvolvido pela equipe de áudio da UnB (Prof. Dr. Pedro Berger e o doutorando Tiago Trindade) a partir das Normas ABNT NBR 15602-2 (1) e da Norma ISO/IEC-14496-3 (20). Como implementações de referência foram utilizados tanto o decodificador da ISO quanto o decodificador de Código aberto FAAD2 disponível em http://www.audiocoding.com. A validação do código foi fundamental para a verificação de sua conformidade com a norma. Para isto, foi criado um algoritmo de comparação onde o Código C foi comparado ao Decodificador de Referência ISO.

Esta comparação foi realizada observando os resultados finais de saída de cada decodificador, ou seja, as amostras de áudio decodificadas. A avaliação de qualidade foi baseada nas métricas de *Signal-to-Noise Ratio* (SNR) conforme a Equação 3.1, onde s são os valores de saída do decodificador em teste e r representa as amostras de saída do decodificador de referência.

$$SNR = 10 \cdot log_{10} \left(\frac{\sum s_i^2}{\sum (s_i - r_i)^2} \right) \tag{3.1}$$

O Procedimento de testes seguiu o fluxo apresentado na Figura 3.1 e os resultados para 5 entradas mono e 5 entradas stereo são apresentados na Tabela 3.1.

Tabela 3.1: Configuração de canais - MPEG-4

Taxa de Amostragem (kHz)	**Stereo (dB)**	**Mono (dB)**
11	79,09	79,00
16	79,28	79,33
22	79,38	78,87
24	79,42	78,91
32	79,48	80,22
44,1	79,52	80,36
48	79,54	79,13

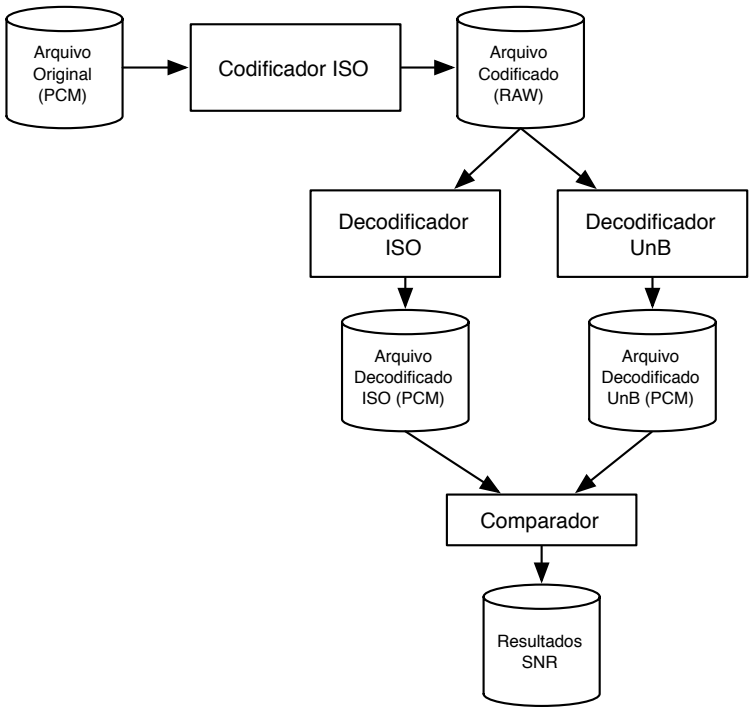

Figura 3.1: Fluxo de comparação das saídas dos decodificadores

Estes resultados mostraram que o algoritmo implementado estava dentro dos padrões de qualidade requeridos.

A segunda análise realizada foi a de performance do algoritmo que se iniciou com a comparação da performance da versão em C do decodificador em relação a outros decodificadores disponíveis no mercado rodando em processadores de arquitetura x86. O objetivo desta análise foi o de avaliar se o algoritmo implementado tinha performance aceitável em relação a outras soluções conhecidas. Utilizamos como referência os decodificadores FAAD2, Nero AAC Dec e iTunes. A Tabela 3.2 mostra as diferenças de performance de cada decodificador para 5 tipos de áudio diferentes, todos com duração de 1 hora. Nela podemos ver que nossa versão (Decodificador UnB) tem a performance de 2 a 3 vezes mais lenta que os demais. Todos os testes forma realizados em um processador Intel Core i7 2635QM de 2.2 GHz.

Tabela 3.2: Performance do Decodificador em Software comparada a outros decodificadores disponíveis.

Decodificador	**Tempo de Decodificação (seg)**
Decodificador UnB	46,23
Nero AAC Dec	16,20
FAAD2	25,39
iTunes	19,80

3.2 Configuração da Plataforma utilizada no desenvolvimento

Após ter o código validado, a etapa seguinte foi a de preparação do ambiente para os testes de perfilamento. Seguindo a filosofia do coprojeto, faz-se necessário definir uma arquitetura e uma plataforma alvo para realizar o perfilamento do código já que cada FPGA e processador possuem características diferentes e consequentemente apresentam desempenhos diferentes em cada etapa do algoritmo.

Para o FPGA, escolhemos a família Cyclone II do fabricante Altera com a placa DE2-70 desenvolvida pela Terasic. Este kit de desenvolvimento oferece fácil acesso a botões, LEDs, LCD, *switches* e saída de áudio com conversor D/A adequada para a saída do decodificador. Além disso, o FPGA oferece 70 mil elementos lógicos que oferecem tamanho suficiente para testes com arquiteturas variadas.

Em relação ao processador, a primeira escolha foi baseada em uma arquitetura aberta. Neste caso, utilizou-se o Plasma que é uma implementação aberta do MIPS disponível no OpenCores. Porém, apesar de fazê-lo funcionar adequadamente na plataforma alvo, o compilador disponível não suportava o uso de bibliotecas e, portando, não seria possível seu uso, sendo necessário criar uma *tool chain* para compilar o projeto como um todo.

Visando o início mais rápido dos testes, optou-se pela escolha do processador NIOS II que é um *softcore* disponibilizado juntamente com as ferramentas do próprio fabricante (Altera) e que oferece uma IDE pronta para desenvolvimento e testes integrada com o FPGA.

Neste caso, criou-se um projeto utilizando-se o ambiente *SOPC Builder (System on a Programmable Chip Builder)*, ferramenta integrada ao Quartus II que permite a definição de uma arquitetura com processador e periféricos conectados por meio do barramento Avalon da Altera. Em sua configuração inicial, utilizamos a versão do processador com maior desempenho disponível (Nios II/f) incluindo multiplicadores em hardware e unidade de ponto flutuante. Alem disso, utilizou-se a frequência de 100MHz que é a máxima pode ser alcançada com esta arquitetura nesta placa de desenvolvimento. A configuração detalhada é apresentada na Tabela 3.3 e a configuração da arquitetura no ambiente SOPC Builder apresentada na Figura 3.2.

Tabela 3.3: Detalhes da Configuração da Arquitetura Inicial do Processador, Memórias

Item	**Tipo**	**Observações**
Processador	Nios II/f	Floating Point Hardware Hardware Multiply (Embedded Multipliers) Hardware Divide (Embedded Multipliers) Instruction Cache - 4KB Data Cache - 4KB
Memória	SSRAM	Memória principal (2MB)
Memória	SDRAM	Memórias auxiliares (2x32MB)
Mem. Flash	Flash	Armazenamento do Bitstream a ser decodificado (8MB).

Para que o código original funcionasse no processador dentro do FPGA, foram necessárias adaptações nas funções de entrada e saída como a leitura do arquivo de entrada

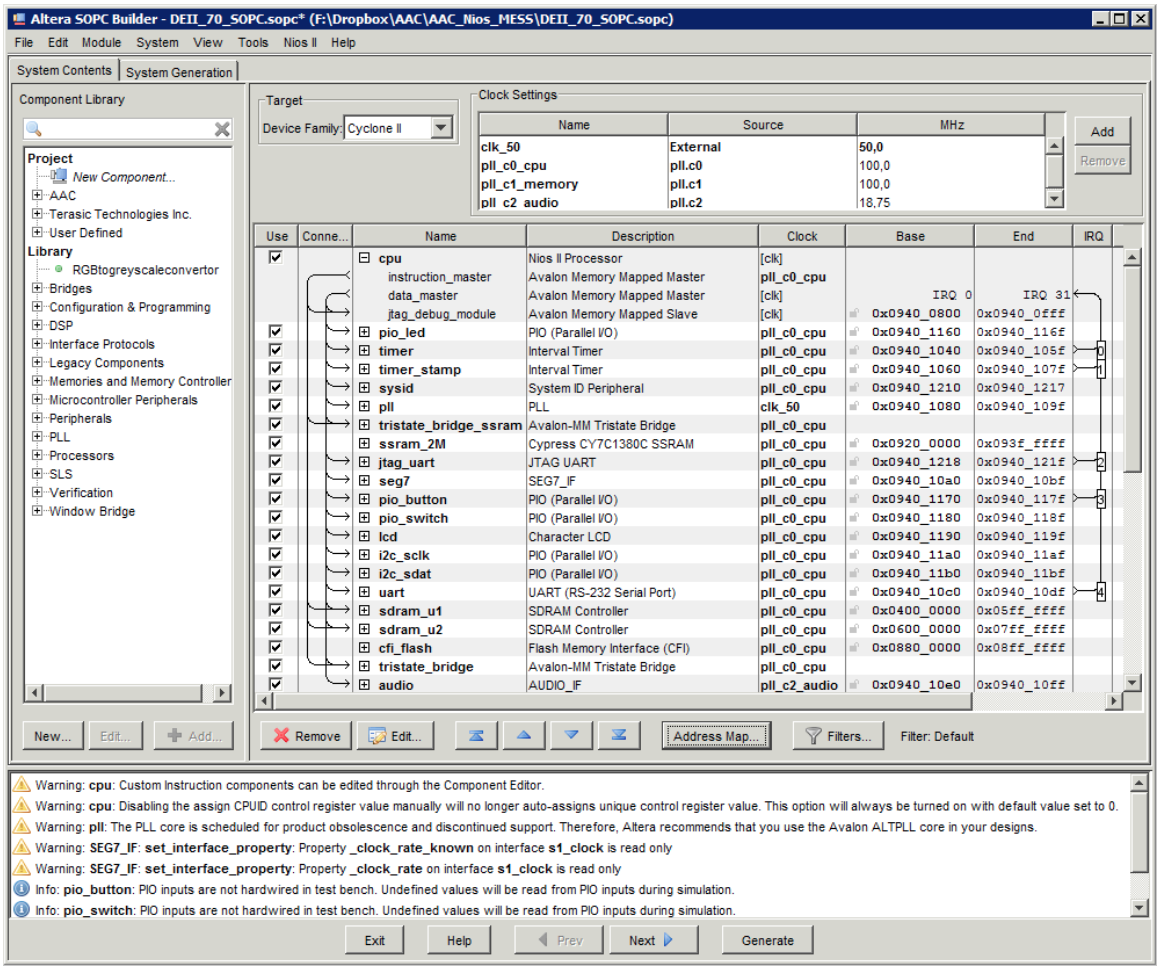

Figura 3.2: Configurações da Arquitetura Inicial no *SOPC Builder*

e a escrita do áudio decodificado. No caso do áudio de entrada, o *bitstream* original em formato RAW foi gravado na memória Flash. Após os primeiros testes nos quais o decodificador apresentou desempenho muito inferior ao tempo real, também direcionamos a saída com o áudio decodificado para a memória SDRAM de 32MB, uma vez que a memória principal SSRAM está limitada a 2MB. Neste caso, seria possível tocar a música sem pausas após a mesma ter sido decodificada, permitindo uma análise da qualidade do som já decodificado.

A Tabela 3.4 apresenta a utilização de hardware desta arquitetura inicial.

Tabela 3.4: Utilização do FPGA na Arquitetura Inicial

Módulo	**Elementos Lógicos**	**Células Lógicas (Comb.)**	**Células Lógicas (Regs.)**	**Memória (Bits)**	**Multiplicadores Dedicados (9x9-bit)**
Processador + *Floating Point Unit* + Barramento	13301	11283	7562	175676	11

3.3 *Profiling* do código

Após ter o código funcionando na plataforma almejada, segue-se para a etapa de perfilamento onde se identificam as partes mais críticas do algoritmo em termos de processamento. Para compreender a complexidade do algoritmo do AAC foi necessário fazer uma análise de cada etapa de decodificação.

O processo de decodificação de áudio MPEG-4 se inicia a partir da leitura do *bitstream* seja ele em formato de arquivo MP4 ou no formato de *streaming*, que utiliza o protocolo de *Transport Stream (LATM – Low-overhead MPEG-4 Audio Transport Multiplex / LOAS - Low Overhead Audio Stream)*, típico das transmissões de TV Digital e internet. No formato de arquivo MP4 (19), as informações gerais de decodificação são armazenadas em um cabeçalho a os dados de áudio são armazenados em um espaço contínuo. No caso do *Transport Stream* (LATM/LOAS) os dados de áudio são transmitidos em pacotes pequenos, cada um deles contendo informações sobre sincronização e parâmetros gerais de decodificação além dos blocos de áudio codificados. Em ambos os casos, os dados de áudio são armazenados em *raw data blocks* contendo uma sequência de canais de áudio, cada um deles com seus próprios parâmetros de decodificação seguidos do áudio codificado em si.

Para cada canal, uma tarefa de *parsing* é utilizada para ler e separar os parâmetros de decodificação e em seguida uma etapa de decodificação sem perdas é utilizada para extrair os coeficientes espectrais de cada bloco de áudio. Esta etapa sem perdas consiste da decodificação Huffman, de uma quantização inversa e de um re-escalonador. O resultado de cada canal é um conjunto de 1024 coeficientes espectrais. A partir disto, uma série de etapas de processamento espectral são executadas, operando sobre os coeficientes espectrais de acordo com os parâmetros de decodificação extraídos inicialmente. A última etapa é transformar os dados de áudio que estão ainda no domínio da frequência em amostras de áudio no domínio do tempo, o que é realizado por meio do banco de filtros inverso.

Para medir a performance utilizou-se a plataforma apresentada na seção anterior. Os testes foram realizados a partir de 20 arquivos de áudio. Dez deles disponibilizados pela norma ISO/IEC 14496-4 e outros dez com perfis variados extraídos de uma biblioteca de músicas, trechos capturados de transmissão de TV Digital e áudio com predominância de voz (*podcasts*). Os arquivos ISO possuem taxa de amostragem variando de 8 kHz a 48 kHz e os demais foram todos gravados a 48kHz (alta qualidade de amostragem). Em todos os casos, utilizamos tanto o *Encoder* de Referência ISO quanto o FAAC *Encoder*.

O primeiro teste realizado, concentrou-se nos 10 arquivos de áudio a 48 kHz e identificou a performance geral do decodificador para taxas de compressão (*bitrates*) variadas. A Tabela 3.5 mostra os resultados para a média dos primeiros 60 segundos de cada um dos arquivos bem como o tempo máximo para cada block de áudio. Neste caso, é importante observar que cada canal de áudio contém 1024 amostras. Portanto, no caso da taxa de amostragem de 48 kHz, em que cada amostra corresponde a 0,020833 ms (1/48), o conjunto de 1024 amostras deve ser decodificado em no máximo 21,33 ms para tocar em tempo real. Para 2 canais são 2048 amostras e para 6 canais 6144 amostras neste mesmo intervalo de 21,33 ms.

A partir da Tabela 3.5, podemos observar que há uma relação direta entre a taxa de compressão (*bitrate*) e o tempo de decodificação, ou seja, quanto maior o *bitrate*, mais

Tabela 3.5: Performance do Decodificador em Software para *bitrates* variados

	Tempo Médio de Decodificação (60 seg de Áudio Stereo)		**Tempo de Decodificação para cada bloco de dados**	
***Bitrate* de Codificação** (kbps)	**Tempo Total (s)**	**Razão para Tempo Real**	**Tempo Máximo (ms)**	**Razão para Tempo Real**
256	566,89	9,45	226,05	10,85
192	553,29	9,20	216,63	10,40
128	534,34	8,90	202,08	9,70
64	498,62	8,30	183,26	8,80
32	464,59	7,75	172,36	8,25

tempo é necessário para decodificar o áudio. Considerando o maior *bitrate*, em média, o decodificador precisaria ser pelo menos 9,5 vezes mais rápido. Porém, para alcançar o tempo real e tocar o áudio sem travamentos é necessário considerar o tempo máximo de decodificação por bloco. Neste caso, para as amostras de áudio stereo analisadas, seria necessário uma melhoria de performance de 10,9 vezes. Fazendo uma extrapolação linear, um áudio de 6 canais precisaria de 3 vezes mais tempo e, portanto, uma aceleração de aproximadamente 32 vezes.

O segundo teste realizado mediu o tempo gasto em cada uma das etapas de decodificação. Neste caso, foram selecionados os 60 segundos iniciais de 5 arquivos de música de áudio estéreo, dentre os 20 previamente selecionados, codificados a uma taxa de 256 kbps. Os resultados médios de tempo utilizado e da razão para a duração total são apresentadas na Tabela 3.6 onde podem ser observados que as etapas do Banco de Filtros e a Decodificação de Entropia são as tarefas que mais consomem o tempo de processamento.

Tabela 3.6: Performance de cada Etapa do Decodificador em Software

Módulos	**Tempo (s)**	**Razão (%)**
Parser	28,00	5,03 %
Decodificação de Entropia (Huffman + Quantização Inversa + Re-escalonamento)	146,87	13,19 %
Ferramentas Espectrais*	19,02	3,42 %
IS	4,64	0,84 %
MS	4,12	0,74 %
TNS	10,12	1,82 %
PNS	0,16	0,03 %
Banco de Filtros	436,31	78,37 %
Total	630,19	100 %

*A linha de "Ferramentas Espectrais" representa a soma de IS, MS, TNS e PNS.

Observa-se que, pela natureza do decodificador de entropia, seu tempo total de decodificação para cada *raw data block* pode variar e, portanto, procuramos considerar os casos em que houvessem poucos elementos sintáticos FIL onde se preenche o bloco com dados vazios. Para o Banco de Filtros observa-se que o tempo de processamento varia

com o tipo de janela sendo o caso da janela curta aproximadamente 70% do tempo de processamento da janela longa principalmente devido ao menor número de estágios da FFT (6 ao invés de 9). Nos trechos de áudio selecionados pouco mais da metade das janelas são no formato longo.

3.4 Definição do Projeto de Hardware

Considerando as medidas de performance apresentadas, iniciou-se o desenvolvimento dos dois módulos críticos. O desenvolvimento dos mesmos foi realizado em VHDL. Para isto, todo o código do decodificador de referência que estava em ponto-flutuante teve que ser adaptado para ponto-fixo. Neste caso a ferramenta MATLAB foi utilizada para facilitar as comparações de sinais de entrada e saída em cada módulo. Os detalhes sobre o número de bits utilizados como parte inteira e fracionária estão apresentados em cada módulo.

Após ter o código escrito em VHDL, a etapa seguinte foi a de efetuar testes de maneira isolada do sistema com a utilização do software QuestaSim 6.6. *Testbenches* de referência foram criados com o intuito de simular o funcionamento do módulo integrado ao sistema. Para isto, arquivos de dados com as entradas de cada módulo eram gerados pelo decodificador em software e utilizados no *testbench*. Os resultados eram comparados com as saídas geradas pelo decodificador de referência em software. Deste modo foi possível medir o erro em cada caso, até que o mesmo pudesse ser ajustado aos padrões de qualidade almejados.

A segunda etapa de testes foi realizada a partir da integração do módulo ao processador Nios II (4) através do Barramento Avalon (2). Em cada caso, foi necessário criar um *wrapper* com portas de ligação ao barramento e funções de controle adaptadas para que fosse possível controlar e monitorar o funcionamento do módulo a partir do processador. Esta integração permitiu a realização de testes mais aprofundados com arquivos completos de áudio sendo decodificados ao mesmo tempo pelo processador no algoritmo em software e pelo módulo em hardware dedicado ao mesmo tempo, sendo os resultados comparados durante a execução.

3.5 Decodificador de Entropia

O primeiro módulo desenvolvido em hardware foi o Decodificador de Entropia que engloba: o decodificador dos Fatores de Escala, o decodificador de Dados Espectrais, a Quantização Inversa e o Re-escalonamento. Estas 4 etapas de decodificação foram desenvolvidas tendo como base a implementação em VHDL feita por Adriano Renner (27). Neste caso, foram realizadas otimizações na arquitetura e adaptações para que os mesmos módulos pudessem se comunicar com o restante do sistema, tanto com o módulo *Stream Buffer* quanto com o processador via barramento Avalon (2).

3.5.1 Decodificador dos Fatores de Escala - DFE

Conforme apresentado na seção 2.2.3, o Decodificador dos Fatores de Escala (DFE) utiliza o método de decodificação Huffman para extrair os fatores de escala codificados no

bitstream. A operação consiste na realização de dois laços sendo o externo definido pelo número de grupos de janelas g de 0 a *num window groups* e o laço interno regido pelo valor sfb que vai de 0 ao valor máximo de bandas de fatores de escala (*max sfb*). Além destes, outros valores decodificados anteriormente pelo *parser*, o valor do Ganho Global (*Global Gain*) e o vetor com o número do *codebook* a ser utilizado para cada banda de fator de escala (*sfb_cb*) são utilizados. Estes dois laços são controlados por uma máquina de estados que varre o vetor *sfb_cb*. Aqui existem 3 casos possíveis para o valor de *sfb_cb*:

1. ZERO_HCB (0): neste caso o valor do Fator de Escala nesta posição composta de g e sfb;

2. INTENSITY_HCB (14) ou INTENSITY_HCB2 (15): neste caso o valor do Fator de Escala se refere à posição de referência para uso da ferramenta espectal Intensity Stereo. Aqui se decodifica o valor a partir do bitstream utilizando a tabela Huffman.

3. Demais valores: aqui se decodifica o valor do Fator de Escala a partir da Tabela Huffman.

A decodificação efetuada com acesso à tabela Huffman é realizada por uma segunda máquina de estados onde o *bitstream* é lido, 1 bit por vez, e o valor deste bit juntamente com um *offset* que inicia em zero, são utilizados para acessar a tabela. O *Offset* define a linha enquanto o valor do bit (0 ou 1) define a coluna a ser acessada. O valor encontrado na posição acessada é retornado para incrementar o valor do *offset*, um novo acesso à tabela é realizado com a nova posição do *offset* até que se encontre uma posição com valor igual a 0. Neste caso, o valor da coluna 0 da linha correspondente ao *offset* é retornado e utilizado para o cálculo do Fator de Escala.

O valor do fator de escala com largura de 9 bits é então armazenado na memória ICS para ser utilizado mais à frente tanto pelo Re-escalonador quanto pelo processamento espectral.

3.5.2 Decodificador dos Dados Espectrais - DDE

A segunda etapa de decodificação que utiliza o algoritmo Huffman é o Decodificador de Dados Espectrais (DDE). Conforme apresentado em 2.2.3, mais uma vez o bitstream é lido 1 bit por vez, porém, diferentemente da decodificação dos Fatores de Escala, esta etapa utiliza 11 tabelas Huffman para extrair os coeficientes espectrais quantizados.

A implementação do módulo consiste em uma máquina de estados principal onde três laços são executados a fim de varrer toda a janela. O laço mais externo percorre os grupos de janelas de 0 até o valor máximo definido por *num window groups*. O segundo laço percorre as seções de cada grupo definidas pelo vetor *num_sec* correspondente ao grupo. Aqui, o vetor *sect_cb* é consultado e sendo seu valor diferente de ZERO_HCB, NOISE_HCB, INTENSITY_HCB, INTENSITY_HCB2, o laço interno é executado nesta seção. Caso contrário passa-se para a próxima seção.

O laço mais interno varre todas as posições de coeficientes espectrais de cada uma das seções e é regido pelos valores armazenados nos vetores *sect_sfb_offset*, *sect_start* e *sect_end*. Para cada posição, verifica-se qual das 11 tabelas será acessada. Nesta implementação, cada tabela foi implementada como um módulo com máquina de estado independente que faz a leitura do *bitstream* e acessa os valores das linhas e colunas de

acordo com as entradas de cada bit até encontrar um valor válido ou um erro. Neste caso, é retornado um valor *idx* a partir do qual são calculados 2 ou 4 valores de coeficientes espectrais quantizados. Em seguida, se avança para a próxima posição dentro da seção.

3.5.3 Quantização Inversa

A implementação da Quantização inversa foi realizada integrada à saída do decodificador de Dados Espectrais. Com o intuito de obter uma solução de rápido processamento para implementar a equação 2.6 que envolve uma potência complexa a ser calculada em poucos ciclos, observa-se o fato de que, sendo os coeficientes quantizados com largura de 13 bits, temos um total de 8192 possíveis valores. Porém, ao invés de implementar uma tabela com 8191 valores, optou-se por usar uma tabela menor e o recurso de interpolação para reduzir o tamanho do hardware. Para isso, tomou-se como referência os trabalhos de Hee, Sunwoo a Moon (15) e Tsai et al. (33).

Nesta abordagem, os valores de 0 a 8191 são divididos em 3 faixas (0 a 255, 256 a 2057 e 2048 a 8191). Os valores da primeira faixa são obtidos diretamente por meio de uma tabela com 256 valores de 16 bits de largura, representando as entradas de $x^{4/3}$. Neste caso, $x^{4/3} = f(x)$, onde $f(x)$ representa o valor da tabela pré-calculada de $x^{4/3}$ para o índice x entre 0 a 255. As duas faixas seguintes são obtidas por aproximação por meio das Equações 3.2 e 3.3. Para executar os cálculos de aproximação, são usados um multiplicador, dois somadores e operações de *shift* para as divisões.

$$x^{4/3} \approx 2 \cdot \left(f\left(\frac{x}{8} + 1\right) - f\left(\frac{x}{8}\right)\right) \cdot rem\left(\frac{x}{8}\right) + f\left(\frac{x}{8}\right) \cdot 16, \quad \text{para } 256 \leq \text{x} < 2048 \tag{3.2}$$

$$x^{4/3} \approx 2 \cdot \left(f\left(\frac{x}{64} + 1\right) - f\left(\frac{x}{64}\right)\right) \cdot rem\left(\frac{x}{64}\right) + f\left(\frac{x}{64}\right) \cdot 16, \quad \text{para } 2048 \leq \text{x} < 8192 \tag{3.3}$$

A saída da quantização são os coeficientes representados por 24 bits, sendo 1 para sinal, 18 de parte inteira e 6 de parte fracionária.

3.5.4 Re-escalonador

Por fim, a última etapa para se obter os coeficientes espectrais é o Re-escalonador. Aqui são utilizados os valores da saída do Quantizador Inverso em conjunto com os Fatores de Escala para recuperar a magnitude original dos coeficientes espectrais.

Neste caso, a Equação 2.7 pode ser re-escrita como 3.4 onde o Fator de Escala é separado em duas partes e seus dois bits menos significativos são usados para acessar uma tabela com valores pré-calculados de 2^0, $2^{1/4}$, $2^{1/2}$ e $2^{3/4}$. Este resultado é multiplicado pelo valor de saída do quantizador inverso e em seguida, o valor de $scale\_factor[9:2]-25$ é utilizado como índice para realizar um deslocamento em um *barrel shifter* e obter o valor final do coeficiente espectral.

$$ganho = 2^{(scale\_factor-100)/4} = 2^{(scale\_factor[9:2]-25)} \cdot 2^{scale\_factor[1:0]} \tag{3.4}$$

O valor final dos coeficientes espectrais é representado com 32 bits sendo 1 de sinal, 25 de parte inteira e 6 de parte fracionária.

3.5.5 Integração do Decodificador de Entropia

Sendo este o primeiro módulo em Hardware a funcionar na arquitetura do AAC, foi necessário, após os testes no simulador, realizar sua integração com o restante do decodificador funcionando no processador. Para isto, foi criado um *wrapper* para a conexão com o barramento AVALON. Neste módulo, são incluídos o Decodificador de Fatores de Escala e o Decodificador dos Dados Espectrais já integrado à Quantização Inversa e ao Re-escalonador. Neste caso, há uma máquina de estados que controla a execução dos mesmos a partir das solicitações do processador. Além da conexão com o barramento, temos a conexão direta com o módulo *Stream Buffer* descrito a seguir na seção 3.6.

A Figura 3.3 apresenta o diagrama da integração dos módulos bem como suas portas de comunicação. Nela podemos observar a presença dos módulos de decodificação e de memórias temporárias utilizadas como *buffer* dos dados a serem retornados ao processador. Aqui estão presentes as memórias para os Fatores de Escala, também utilizada pelo Re-escalonador, a de Coeficientes Espectrais decodificados bem como uma área de memória ICS para os parâmetros usados por estes módulos. Há também a existência de um multiplexador que coordena o acesso tanto do DFE quanto do DDE ao *Stream Buffer*. São três portas de conexão com o *Stream Buffer*, cada uma com largura de 1 bit.

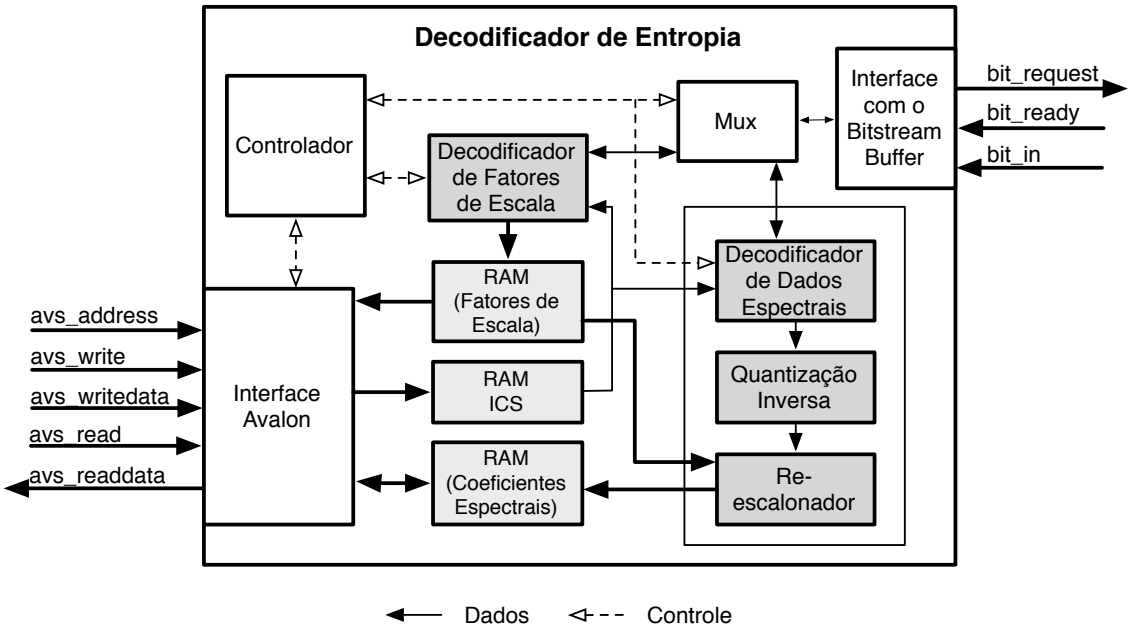

Figura 3.3: Diagrama da arquitetura do Decodificador de Entropia

A conexão com o barramento Avalon é realizada no modo escravo (*Slave*) a partir de um multiplexador que aciona o controlador a partir dos sinais de leitura e escrita do barramento. Todo o controle é feito pelo processador através do barramento e o módulo somente responde às requisições. Neste caso, implementamos somente as portas mais básicas que são:

1. AVS_ADDRESS: Endereço usado para acionar a função desejada no multiplexador (Neste caso com 3 bits de largura);

2. AVS_WRITE: Sinal de 1 bit indicando a solicitação de escrita vinda do barramento;

3. AVS_WRITEDATA: Sinal com largura de 32 bits contendo os dados vindos do processador;

4. AVS_READ: Sinal de 1 bit indicando a solicitação de leitura do barramento;

5. AVS_READDATA: Sinal com largura de 32 bits a ser preenchido pelo módulo a fim de ser lido pelo barramento no próximo ciclo re *clock*.

A Tabela 3.7 apresenta os resultados da síntese em FPGA com o uso de hardware do Módulo.

Tabela 3.7: Utilização de Hardware do Decodificador de Entropia

Módulo	**Elementos Lógicos**	**Memória (bits)**	**Multiplicadores Dedicados (9x9-bit)**
Decodificador de Entropia AVS *wrapper*	94	0	0
Memória ICS	-	16576	-
Decodificador de Fatores de Escala	380		0
Memória de Fatores de Escala	-	4608	-
Decodificador de Dados Espectrais	4318	0	8
Quantização Inversa	1625	0	4
Re-escalonador	394	0	3
Memória de Dados Espectrais	-	32768	-
Total	**6417**	**53952**	**15**

3.6 Stream Buffer

A leitura do *bitstream* é realizada pelo decodificador 1 bit por vez no caso do decodificador de entropia e de 1 a 16 bits por vez para o caso do *parser*. Considerando que a entrada de dados pode vir da leitura de um arquivo em disco, em memória ou de uma transmissão direta, estes dados normalmente virão agrupados em um ou mais bytes tornando necessário um mecanismo para leitura com maior granularidade.

A primeira versão do *Stream Buffer* foi feita direcionada especificamente para o Decodificador de Entropia permitindo apenas a leitura de 1 bit por vez diretamente em hardware. Depois, observamos que o *parser* sendo executado pelo processador também poderia se beneficiar deste módulo já que o processador gasta alguns ciclos para poder realizar as operações de *shift* e concatenação e mascaramento para obter a quantidade requerida de bits.

A versão mais simples de 1 bit foi criada baseada em um *buffer* circular contendo ao todo 64 bits separados em dois registradores de 32 bits. Neste caso, optou-se pelos 32

bits para aproveitar a largura de palavras do barramento Avalon que transmite em uma só comunicação o máximo de bits possível. O módulo é controlado por duas máquinas de estado, uma para atender às solicitações de leitura dos bits e realizar a operação de deslocamento e a outra para realizar a recarga dos registradores. A existência dos dois registradores garante que a operação de leitura não será interrompida já que a recarga é realizada em paralelo.

A segunda versão inclui a possibilidade de leitura de qualquer valor entre 1 e 16 bits e foi implementada com o mesmo princípio dos dois registradores de 32 bits, porém com adaptações. Neste caso três máquinas de estado atuam em conjunto. A primeira controla as requisições de leitura do *buffer*, a segunda controla o procedimento de recarga do *buffer* enquanto uma terceira atua como intermediária para sincronizar os sinais de *buffer vazio* e *buffer recarregado*.

A Figura 3.4 apresenta a arquitetura do *Stream Buffer* enquanto as Figuras 3.5 e 3.6 apresentam os diagramas com o funcionamento das duas principais máquinas de estado. Note que os 16 bits iniciais do registrador 1 são replicados ao final para que a leitura de bits acima de 48 possam ser realizadas com até 16 bits de largura já contando com a próxima recarga do registrador 1.

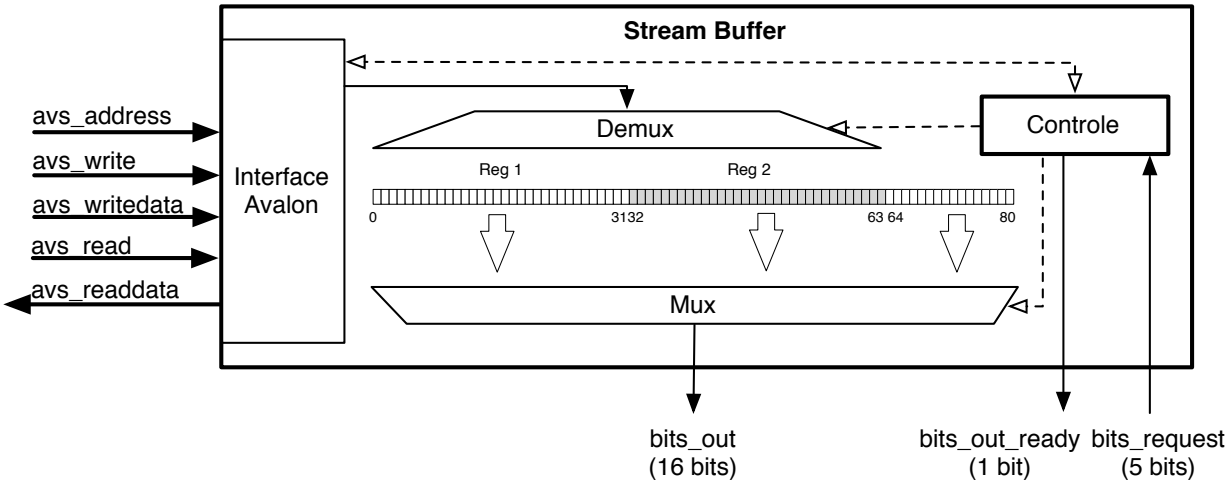

Figura 3.4: Diagrama da arquitetura do *Stream Buffer*

A Máquina de Estados 1 possui 5 estados sendo o primeiro, $s0$, o estado inicial no qual se aguarda a recarga caso os registradores estejam vazios. O controle de consumo dos bits é feito através de um ponteiro que é incrementado com o número de bits lidos a cada operação de leitura. Conforme pode ser observado na Figura 3.5, são 4 os casos em que a máquina permanece em $s0$ aguardando a recarda do *buffer*. Caso o *buffer* 1 esteja vazio, não se pode ler bits nas posições abaixo de 16, obviamente, e também não é possível acima de 47, pois caso a requisição seja de 16 bits para leitura, estes não estarão atualizados. Da mesma maneira, caso o *buffer* 2 não tenha sido carregado, não se podem ler bits entre 32 e 63 ou acima de 15 pelo mesmo motivo anterior. Portanto, o estado $s0$ verifica se o *buffer* está pronto para ser lido de acordo com o estado de recarga dos registradores e a posição do ponteiro.

A operação de leitura é realizada normalmente pelos estados $s1$ e $s2$. Em $s1$, o sinal de saída *bit_ready* é colocado em '1' e aguarda-se a requisição pelo sinal *bit_request* que

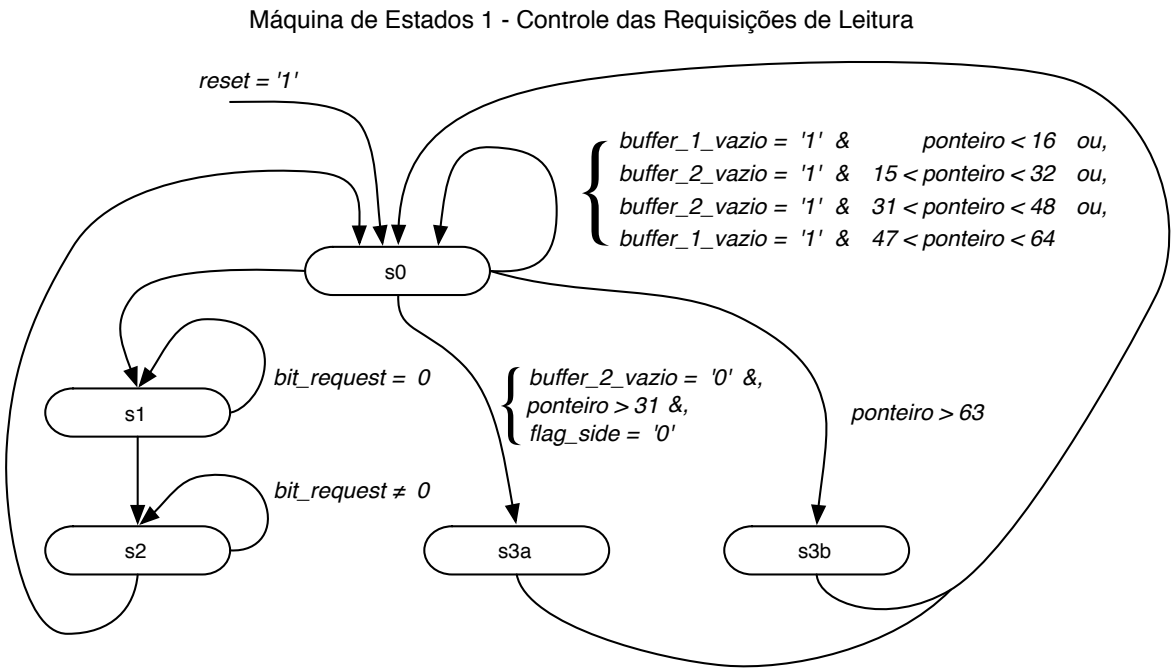

Figura 3.5: Diagrama da Máquina de Estados 1 do *Stream Buffer*

indica inclusive quantos bits serão lidos. Em seguida passa-se para o estado $s2$ que aguarda o sinal de leitura, ou seja, $bit\_request = 0$. Neste momento o ponteiro é incrementado e se retorna ao estado $s0$.

Os estados $s3a$ e $s3b$ são ativados quando o ponteiro atinge o final de um registrador. Um *flag* de recarga do mesmo é acionado.

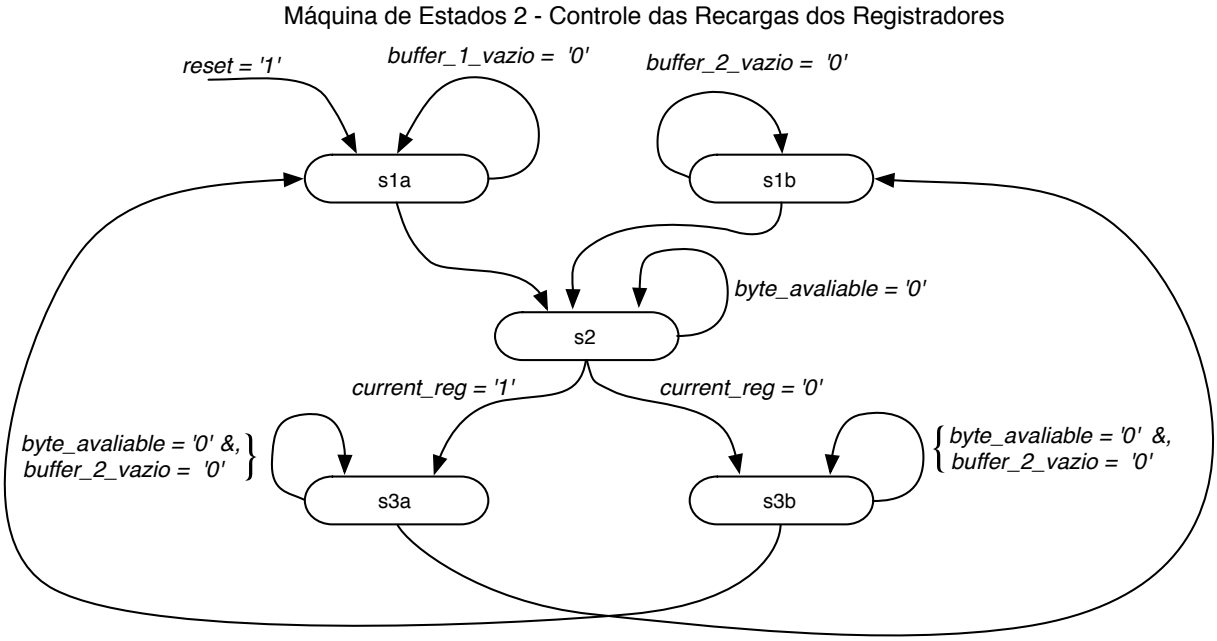

Figura 3.6: Diagrama da Máquina de Estados 2 do *Stream Buffer*

A Máquina de Estados 2 opera em paralelo verificando os *flags* de *buffer* vazio de cada registrador. O processo se inicia pelo estado $s1a$ em que se aguarda a solicitação de recarga do registrador 1. Em seguida, entra-se no estado $s2$ no qual o pedido de recarga é feito e aguarda-se o retorno com os novos 32 bits do *bitstream*. De $s2$ se vai para $s3b$ no caso do registrador 1 ter sido recarregado. O processo então se repete para o ciclo do registrador 2 em $s1b$, $s2$ e $s3a$.

A recarga dos registradores nesta implementação é feita pelo processador através do barramento uma vez que o arquivo de áudio está sendo lido da memória flash controlada pelo mesmo. Porém, a arquitetura está pronta caso outro mecanismo específico em hardware, por exemplo o Demux na arquitetura da TV Digital esteja pronto para fornecer o *bitstream*.

O controle de sincronia entre as Máquinas 1 e 2 é realizado por duas outras máquinas simples com 2 estados cada. O papel destas é acionar os *flags* *buffer_1_vazio* e *buffer_2_vazio* logo após a passagem da Máquina 1 pelos estados $s3a$ ou $s3b$ respectivamente. Em seguida, ao receber o sinal de *buffer_recarregado* 1 ou 2, o *flag* do registrador respectivo é colocado em 0.

A Tabela 3.8 apresenta os resultados da síntese em FPGA com o uso de hardware do Módulo.

Tabela 3.8: Utilização de Hardware do Stream Buffer

Versão	**Elementos Lógicos**	**Memória (bits)**	**Multiplicadores Dedicados (9x9-bit)**
1 bit	112	0	-
1 a 16 bits	509	0	-

3.7 Banco de Filtros Inverso

O Banco de Filtros Inverso foi o segundo módulo em hardware a ser desenvolvido nesta arquitetura. Conforme apresentado na seção 2.2.7, o Banco de Filtros Inverso é composto pela IMDCT, seguida da aplicação das funções de Janelamento e da sobreposição e adição com parte da janela anterior (*Overlap and Add*). Este é o módulo que contém as funções mais críticas de todo o AAC em termos de utilização de memória e processamento. Conforme visto no procedimento de *Profiling* na seção 3.3, seu processamento é responsável por aproximadamente 78% do uso do processador.

Conforme determinado pela norma ABNT NBR 15602-2 (1), o AAC deve decodificar tanto janelas curtas com N=256 amostras quanto janelas longas com N=2048 amostras. No caso do Banco de Filtros Inverso, o processamento é feito individualmente para cada canal onde a entrada é fixa com 1024 coeficientes espectrais e um *flag* indicando o tamanho da janela informa se é uma janela longa onde os N/2=1024 coeficientes são decodificados de maneira contínua ou se temos oito janelas curtas, contendo 8 grupos de N/2=128 coeficientes espectrais. Além desta informação, recebem-se também os parâmetros *window_shape* informando o tipo de função utilizada no Janelamento e *window_sequence* que indica a forma da sequência a ser usada para o Janelamento.

Conforme visto na Revisão de Literatura e discutido por Li (22) e Lai (21), existem diversas maneiras de implementar o algoritmo da IMDCT. Existem métodos que utilizam uma arquitetura recursiva e que são propícios para situações nas quais se têm poucos recursos de hardware e em que o requisito de velocidade de processamento é baixo, conforme pode ser visto em (25, 8). Por outro lado, há métodos de implementação baseados em um núcleo de FFT que oferece não só um desempenho computacional rápido quanto maior precisão, conforme pode ser visto em (13, 11, 24). Seguindo nossas necessidades de decodificação em tempo real, optamos pela implementação com o uso da FFT que, no caso é baseada no algoritmo Radix-2.

A implementação do Banco de Filtros é composta por quatro grandes módulos. Os primeiros três são a implementação da IMDCT que é dividida na primeira etapa de Pré-processamento, na segunda etapa onde se utiliza uma iFFT ou uma FFT inversa e na terceira etapa de Pós-processamento. O último módulo implementa em conjunto as funções de Janelamento, Sobreposição e Adição (*Windowing, Overlap and Add*).

A seguir descrevemos a implementação de cada um destes módulos.

3.7.1 Pré-Processamento da IMDCT

O Pré-processamento é a etapa inicial da IMDCT que irá preparar os dados a serem calculados pela iFFT. O procedimento de preparação dos dados consiste em transformar os N coeficientes espectrais (N/2=1024 para janelas longas ou N/2=128 para janelas curtas) em dois vetores de N/4 elementos, sendo um vetor da parte real (Re) e outro da parte imaginária (Im).

Conforme documentado por Du em (10), as expressões que descrevem o pré-processamento da IMDCT são apresentadas pelas equações 3.5 e 3.6 onde X é o vetor de entrada, Y é o vetor de números imaginários e Y_1 a saída.

$$Y(k) = \left(-X(2k) + jX\left(\frac{N}{2} - 1 - 2k\right)\right), \quad \text{para } 0 \leq k < \frac{N}{4} \tag{3.5}$$

$$Y_1(k) = Y(k) \cdot e^{j\left(\frac{2\pi k}{N} + \frac{\pi}{4N}\right)}, \quad \text{para } 0 \leq k < \frac{N}{4} \tag{3.6}$$

O algoritmo 3.1 em C que descreve o laço de implementação do pré-processamento da IMDCT é apresentado abaixo onde podemos perceber que são realizadas 4 multiplicações pelos chamados fatores *Twiddle* de $e^{j\left(\frac{2\pi k}{N} + \frac{\pi}{4N}\right)}$, uma soma e uma subtração para obter os vetores Real e Imaginário. No algoritmo n2 e n4 são os valores da entrada N = 2048 ou 256 dividido por 2 e 4 respectivamente, X é o vetor de entrada, Re e Im são a os vetores Real e Imaginário de saída. Os fatores *Twiddle* estão armazenados em 2 tabelas de seno e cosseno.

Algoritmo 3.1: Algoritmo do Pré-processamento da IMDCT

```
for (k = 0; k < n4; k++) {
     Im[k] = (X[2k] * twiddle[0][k]) + (X[n2-1-2k] * twiddle[1][k]);
     Re[k] = (X[n2 - 1 - 2k] * twiddle[0][k]) - (X[2k] * twiddle[1][k]);
 }
```

A Figura 3.7 ilustra a sequência de operações deste pré-processamento.

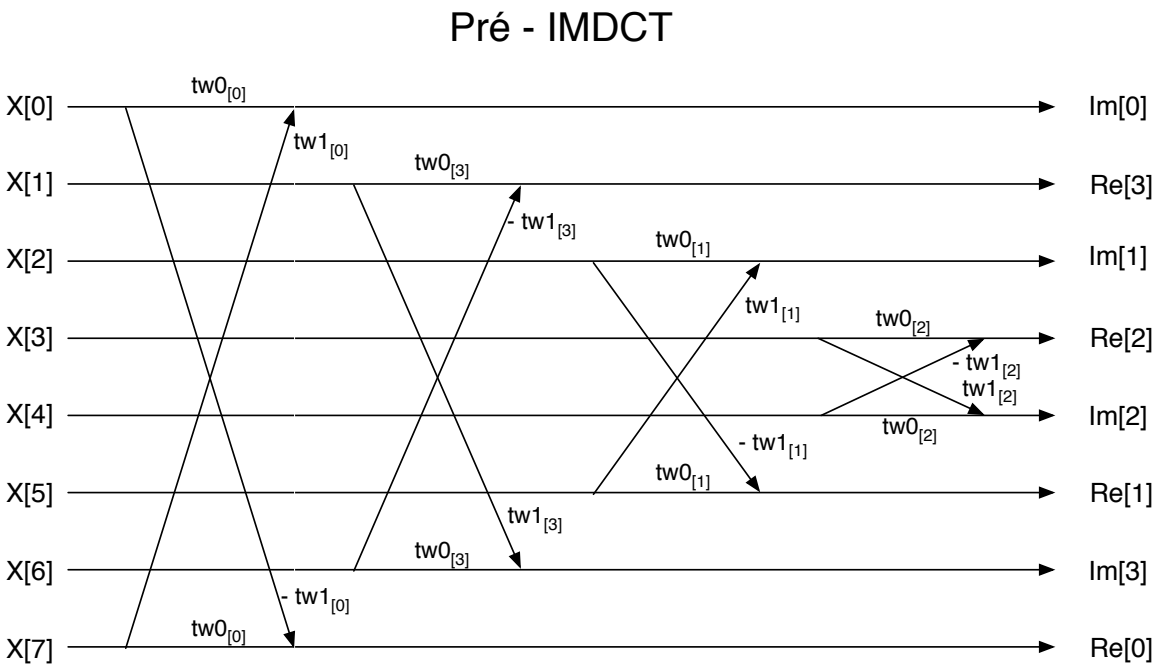

Figura 3.7: Diagrama mostrando exemplo do processamento da Pré-IMDCT simplificada com N/2 = 8 onde podem ser observadas a execução de 4 laços.

Para a implementação em hardware foi necessário observar as limitações de entrada e saída das memórias. Neste caso, a memória de entrada, onde estão armazenados os coeficientes espectrais, foi definida como *single channel* com o intuito de manter a arquitetura uma arquitetura simples para uma futura implementação em silício. Portanto, há acesso a um único valor por ciclo de *clock*. Para cada execução do laço, é necessário acessar duas posições desta memória de entrada $X(2k)$ e $X(n2-1-2k)$. Para a saída, temos duas áreas *single channel* separadas de memória onde é necessário acesso a apenas uma posição $Re(k)$ e $Im(k)$ a cada ciclo do laço. Quanto aos fatores *Twiddle*, os mesmos estão armazenados em 4 áreas de memória ROM *single channel*, duas para os fatores das janelas longas com 1024 fatores de 32 bits cada e duas para as janelas curtas com 128 fatores cada. Os fatores *Twiddle* também têm um único acesso por ciclo.

Visando alcançar o maior desempenho possível com as limitações das memórias, nossa implementação foi realizada utilizando 4 multiplicadores com entradas de 32 bits e dois somadores. O controle é realizado por uma máquina de estados em que apenas dois estados são utilizados para a realização do cálculo, um terceiro estado auxilia na inicialização da entrada e outro indica o fim da operação. O diagrama de estados é apresentado na Figura 3.8.

Apesar da simplicidade da máquina de estados, a operação de pré-processamento requer uma complexa sincronia entre os valores de todos os sinais envolvidos no cálculo. O estado inicial *start* é utilizado para que se possa receber o valor de $x(0)$ e definir o valor de x_addr para 1023, no caso da janela longa, para que em $s0$ já tenhamos ambos os valores $X(2k)$ e $X(n2-1-2k)$ bem como os valores dos fatores *Twiddle* e assim poder executar as quatro multiplicações em paralelo. Em seguida, no estado $s1$ já possuindo os resultados

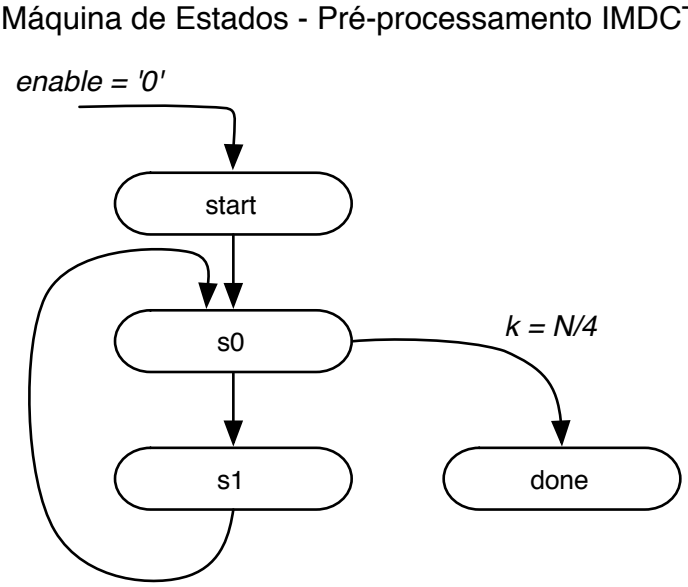

Figura 3.8: Diagrama da Máquina de Estados do Pré-processamento da IMDCT

das multiplicações, podemos executar a soma e a subtração. Enquanto isso, o valor de k é incrementado em $s0$ e utilizado para recuperar em $s1$ o próximo valor de $X(2k)$ para que em $s0$ tenhamos o próximo valor de $X(n2-1-2k)$. Finalmente, no próximo ciclo, em $s0$, podemos armazenar na memória os resultados da soma e da subtração em Re_z e Im_z na posição (k-1) que é armazenada no sinal k_aux. Além disso, já temos os novos valores dos *Twiddles* e podemos executar as próximas quatro multiplicações.

O procedimento pode ser melhor compreendido pelo exemplo apresentado na Tabela 3.9 para o caso de uma janela longa. Na Tabela são apresentados os valores de cada sinal tanto em seus estados inicial e final quanto nos estados intermediários onde temos o exemplo para dois ciclos completos da máquina. A Tabela 3.10 apresenta em VHDL as operações implementadas tanto para controle dos endereços de acesso quanto para a manipulação dos dados.

Com esta implementação conseguimos alcançar o máximo desempenho possível sendo limitado pela memória de entrada. Todo os processamento é realizado em N/2+2 ciclos sendo uma para início, N/2 de processamento e 1 para o término. A utilização de uma memória *dual channel* na entrada resultaria em uma execução com a metade dos ciclos. No FPGA isto não teria grandes consequências no uso dos recursos pois as memórias embarcadas já suportam esta arquitetura. Porém, em silício, testes realizados com a síntese de tais memórias apontaram o uso de de uma área 2,19 vezes maior e consumindo 2,09 vezes mais energia. Portanto, optou-se pelo uso de memórias *single channel*.

Finalmente, observamos que, conforme estudo realizado no MATLAB, as operações em ponto-fixo foram realizadas conforme a Tabela 3.11.

3.7.2 iFFT - Inverse Fast Fourier Transform

A segunda etapa da IMDCT é a da iFFT ou *Inverse Fast Fourier Transform*. A FFT é um algoritmo que implementa de maneira eficiente a Transformada Discreta de Fourier

Tabela 3.9: Exemplo dos valores dos sinais do Pré-processamento para janelas longas

	Estados							
Sinais de controle	Valor Inicial	start	s0	s1	s0	s1	...	done
k_aux	0	0	0	0	0	1	...	511
k	0	0	1	1	2	2	...	511
x_addr	0	1023 (x2_addr)	2 (x1_addr)	1021 (x2_addr)	4	1019	...	
x1_addr	0	0	2	2	4	4	...	1022
x2_addr	1023	1023	1021	1021	1019	1019	...	1
twd_addr	0	0	1	1	2	2	...	511
Re_Im_addr	0	0	1	1	2	2	...	511
Entradas								
x		x(0) (x(2k))	x(1023) (x(n2-1-2k))	x(1) (x(2k))	x(1021)	x(2)	...	
x_tmp			x(0) (x(2k))	x(0) (x(2k))	x(1)	x(1)	...	
twd_0		twd_0(k)	twd_0(k)	twd_0(k)	...	...	...	
twd_1		twd_1(k)	twd_1(k)	twd_1(k)	...	...	...	
Saidas								
Re		-	-	-	Re(0)	-	...	Re(511)
Im		-	-	-	Im(0)	-	...	Im(511)

Tabela 3.10: Descrição das operações do Pré-processamento

	s0	s1
Operações de Controle	k_aux <= k k<=k+1 twd_addr<=k+1 x1_addr<=(k+1)&'0' x2_addr<=not(x1_addr)&'1'	
Operações com Dados	m1<=x·twd_0 m2<=x·twd_1 m3<=x_tmp·twd_0 m4<=x_tmp·twd_1	Im <= m1 + m4 Re <= m3 - m2

Tabela 3.11: Tabela com o número de bits para operações de ponto-fixo do Pré-processamento da IMDCT

Dado	**Sinal**	Parte Inteira	Parte Fracionária
Entradas (Coef_Spec_in)	1	25	6
Twiddles	-	1	31
Saídas (Re e Im)	1	15	16

(DFT), sendo largamente utilizado em processamento de sinais. O algoritmo usado foi o Radix-2 que é uma implementação do algoritmo Cooley–Tukey que divide recursivamente o conjunto de N valores em N/2 até finalizar os cálculos. Neste caso, N deve ser uma

potência de 2. A Equação 3.7 representa o cálculo da FFT onde, dada uma sequência $x(n)$, calcula-se $X(k)$. As equações 3.8 a 3.9 apresentam a fatoração de 3.7 mostrando o princípio do algoritmo Radix-2.

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-\left(i \cdot \frac{2\pi nk}{N}\right)} \tag{3.7}$$

$$X(k) = \sum_{n=0}^{N/2-1} x(2n) \cdot e^{-\left(i \cdot \frac{2\pi(2n)k}{N}\right)} + \sum_{n=0}^{N/2-1} x(2n+1) \cdot e^{-\left(i \cdot \frac{2\pi(2n+1)k}{N}\right)} \tag{3.8}$$

$$X(k) = \sum_{n=0}^{N/2-1} x(2n) \cdot e^{-\left(i \cdot \frac{2\pi nk}{N/2}\right)} + e^{-\left(i \cdot \frac{2\pi nk}{N}\right)} \cdot \sum_{n=0}^{N/2-1} x(2n+1) \cdot e^{-\left(i \cdot \frac{2\pi nk}{N/2}\right)} \tag{3.9}$$

$$X(k) = DFT_{\frac{N}{2}}\left[[x(0), x(2), ..., x(N-2)]\right] + W_N^k \cdot DFT_{\frac{N}{2}}\left[[x(1), x(3), ..., x(N-1)]\right] \tag{3.10}$$

Nestas Equações é possível observar a separação dos coeficientes ímpares (lado direito das somas) dos pares (lado esquerdo das somas), transformando as N entradas em dois grupos de N/2 entradas, operação esta que é aplicada recursivamente até atingir a unidade mínima de cálculo que é uma Borboleta (*Butterfly*). A Borboleta que consiste em 2 multiplicações e duas somas de números complexos conforme a Figura 3.9. Nela, são utilizados os fatores *Twiddle* que são representados na Equação 3.9 por $e^{-\left(i \cdot \frac{2\pi nk}{N}\right)}$ e 3.10 por W_N^k. As Equações 3.11 e 3.12 apresentam a operação da borboleta.

$$X_0 = x_0 + W_N^k \cdot x_1 \tag{3.11}$$

$$X_1 = x_0 - W_N^k \cdot x_1 \tag{3.12}$$

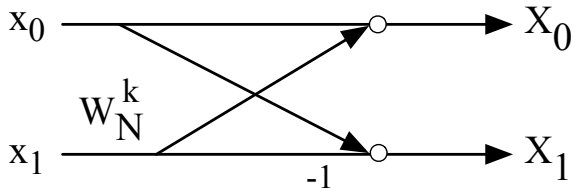

Figura 3.9: Diagrama da Borboleta da FFT Radix-2

O algoritmo de cálculo é executado em $log_2(N/2)$ ciclos onde todos os N fatores passam pela Borboleta a cada ciclo conforme mostrado no exemplo de uma FFT de 8 pontos da Figura 3.10. A implementação da iFFT é baseada na FFT. A única diferença é a necessidade de invertemos as entradas dos vetores da parte Real e Imaginária.

O processo de implementação em hardware partiu de um algoritmo em C descrito em ponto flutuante em (12) que foi adaptado para ponto-fixo. Diferentemente dos demais módulos, cuja implementação em ponto-fixo foi testada no MATLAB, para a FFT foi criado um *testbench* utilizando o pacote SC_Verify da ferramenta Catapult, que é uma ferramenta de síntese de alto nível, onde o código C pode ser transformado em descrição

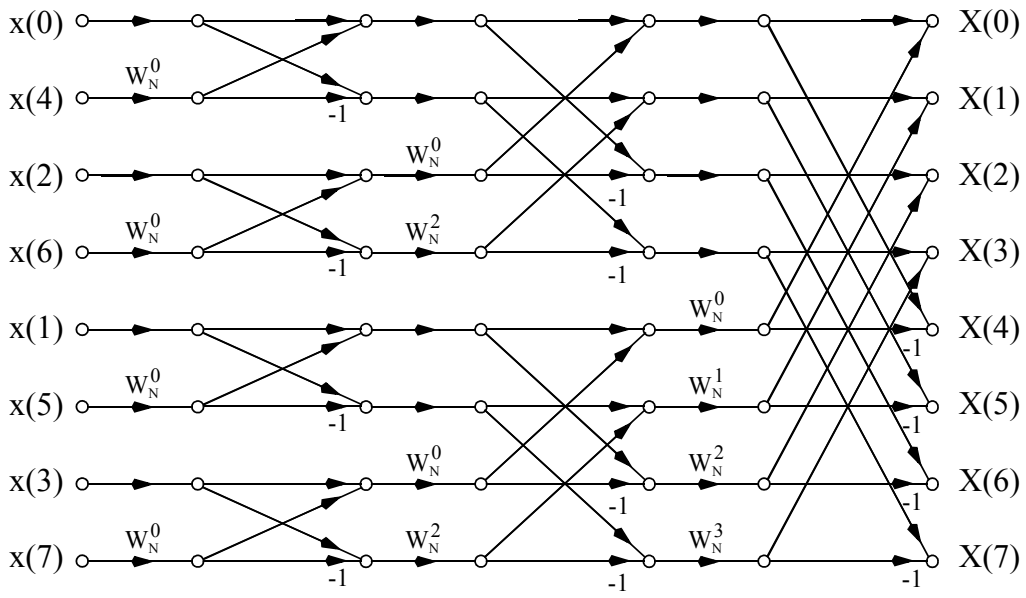

Figura 3.10: Diagrama mostrando a operação da FFT Radix-2 para 8 entradas.

VHDL ou Verilog. Neste caso, foram introduzidos vetores de entrada tanto do algoritmo original em C em ponto-flutuante quanto no algoritmo em ponto-fixo e as saídas foram comparadas em C. As diferenças encontradas foram um erro máximo de 0,0056% e um erro médio de 0,000091%.

Foram feitas algumas rodadas de implementação automática a partir do código C, explorando diferentes arquiteturas e configurações de laços com resultados razoáveis conforme mostrado na Tabela 3.12. Porém, optou-se por seguir o desenvolvimento manual para fazer, inclusive, uma comparação de eficiência do método de implementação. Como pode ser observado na tabela e conforme descrição a seguir, a implementação manual permite manipulações de bits individuais dos sinais de tal modo que sua descrição em C necessitaria de operações de *shift*, mascaramento com *and* e *or* que seriam sintetizadas em vários ciclos.

O desenvolvimento manual em Hardware foi realizado, neste caso, em Verilog, e utilizou-se o mesmo algoritmo em ponto-fixo como base para implementar as duas máquinas de estado, uma controlando o Laço principal que define o endereço de acesso às memórias em cada iteração e outra controlando a borboleta (multiplicações e somas). O objetivo foi utilizar o menor hardware possível com a maior eficiência alcançável considerando as limitações de memória do algoritmo *in-place*. Neste caso, são utilizadas as duas áreas de memória RAM de acesso simples (*single channel*) para os dados de entrada e saída (Re e Im) e uma memória ROM de acesso simples para os fatores de multiplicação ($Twiddles$).

A operação da borboleta consiste em ler da memória dois valores do vetor Re e dois valores do vetor Im, executando quatro multiplicações dos coeficientes reais e imaginários pelos fatores $Twiddle$, uma soma e uma subtração, conforme pode ser visto no algoritmo 3.2. Em seguida, mais duas somas e duas subtrações são realizadas para obter os valores de saída.

Tabela 3.12: Versões da FFT de 64 entradas implementadas no Catapult e em Verilog manualmente

Uso de Recursos FPGA	**Ver.1**	**Ver.2**	**Ver.3**	**Ver.4**	**RTL Verilog**
LC Combinational	3820	1948	1958	2126	1306
LC Registers	727	765	760	592	385
9-bit multipliers	8	32	32	64	32
Memory Bits	1536	1984	1984	1984	2048
Tempo total (50 MHz)	29,44 us	26 us	22,3 us	13,5 us	16,6 us
Fmax	80,8 MHz	76,9 MHz	76,9 MHz	76,4 MHz	167,87 MHz
Configuração	3 Laços de tamanho variável Memória Single Ch.	2 Laços de tamanho fixo Memória Single Ch.	2 Laços de tamanho fixo Memória Single Ch. Pipeline de 5 ciclos	2 Laços de tamanho fixo Memória Dual Ch. Pipeline de 3 ciclos	1 Laço de tamanho variável Memória Single Channel
Tempo de Implementação	3 dias				16 dias

Algoritmo 3.2: Algoritmo da Borboleta da FFT

```
tempr = c * Re[j] - s * Im[j];
tempi = c * Im[j] + s * Re[j];
temprr = Re[i];
tempii = Im[i];
Re[j] = temprr - tempr;
Im[j] = tempii - tempi;
Re[i] = temprr + tempr;
Im[i] = tempii + tempi;
```

Com o intuito de realizar o menor número de ciclos possível, observamos que temos duas áreas de memória, onde se pode ler ou gravar 2 dados por ciclo de *clock*. Como é necessário ler 4 dados e escrever outros 4 dados, o tempo mínimo neste algoritmo *inplace* para a execução da borboleta é de 4 ciclos.

Como as multiplicações são realizadas somente para os valores $Re(j)$ e $Im(j)$, podem ser executadas já no estado $s0$, em seguida, realizamos a soma e subtração dos resultados da multiplicação em $s1$ e recuperamos os valores de $Re(i)$ e $Im(i)$ da memória. Em $s2$ realizamos a somas e subtrações finais gerando as 4 saídas que são então gravadas novamente na memória em $s3$ e $s0$.

Além da Borboleta, outras duas máquinas de estado são utilizadas para controlar os laços de *bit reverse* e o laço principal da FFT. Na operação de *bit reverse*, as posições dos vetores de entrada são reordenadas invertendo suas posições na memória pelo endereço de bits invertido conforme pode ser visto na Figura 3.11. Neste caso, por exemplo, o endereço "0001" vira "1000" enquanto "0110" se mantêm.

O laço principal é implementado em uma máquina com 4 estados que operam sincronicamente e em paralelo à borboleta. Neste caso, os índices de cada iteração da fft são calculados e incrementados. No total, são realizadas $log_2(N) \cdot N/2$ operações de borboleta,

sendo N o número de entradas da FFT que no caso de janelas curtas é 64 e de janelas longas é 512.

Os coeficientes *Twiddle* foram armazenados em memória ROM com largura de 32 bits. Como o algoritmo só exige que os valores dos *Twiddles* sejam atualizados para cada borboleta, a tabela completa foi implementada em uma única área de memória com 384 dados (capaz de processar FFTs de 64 a 512 pontos).

O diagrama das três máquinas de estado é apresentado na Figura 3.11.

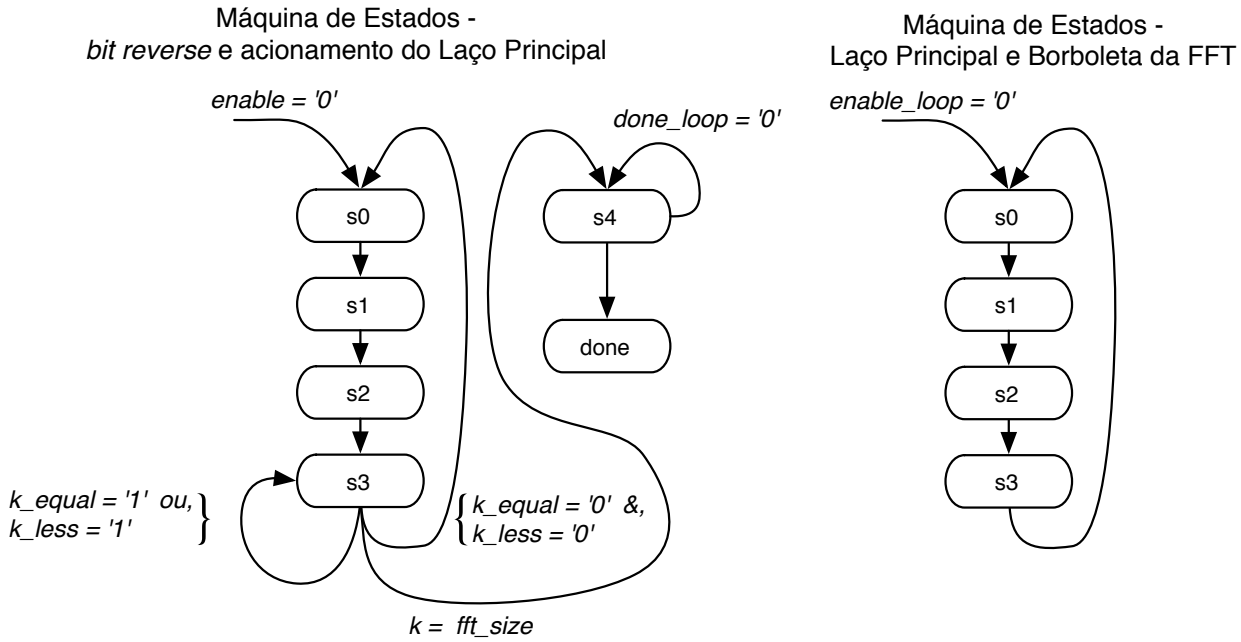

Figura 3.11: Diagrama das Máquinas de Estado da FFT

A implementação contempla tanto entradas de 512 quanto de 64 valores, sendo necessário somente definir o tamanho da janela como parâmetro de entrada do módulo. As operações em ponto-fixo foram realizadas conforme a Tabela 3.13.

Tabela 3.13: Tabela de número de bits para operações de ponto-fixo da FFT

Dado	**Sinal**	Parte Inteira	Parte Fracionária
Entradas e Saidas (Re e Im)	1	15	16
Twiddles	-	1	31

3.7.3 Pós-Processamento da IMDCT

O Pós-processamento é a etapa final da IMDCT onde os vetores Re e Im serão novamente transformados nos 2048 coeficientes espectrais. Este processamento é realizado em duas etapas. A primeira delas tem processamento semelhante ao da primeira etapa da IMDCT, o Pré-processamento. Neste caso, o cálculo é representado pela Equação 3.13, onde Y é a saída intermediária e X é a entrada dos vetores Real e Imaginário.

$$Y(k) = X(k) \cdot e^{j\left(\frac{2\pi k}{N} + \frac{\pi}{4N}\right)}, \quad \text{para } 0 \leq k < \frac{N}{4} \tag{3.13}$$

O algoritmo 3.3 em C, apresenta a implementação da Equação 3.13 onde temos a saída da iFFT armazenada nas memórias Re e Im cujos valores serão, mais uma vez, multiplicados pelos mesmo fatores $Twiddle$ utilizados na etapa de pré-processamento. Em seguida, é realizada uma soma e uma subtração e os resultados são armazenados em memória.

Algoritmo 3.3: Primeira Etapa do Pós-processamento da IMDCT

```
for (k = 0; k < n4; k++) {
    Re_tmp[k] = Re[k];
    Im_tmp[k] = Im[k];
    Im[k] = (Im_tmp[k] * twiddle[0][k]) + (Re_tmp[k] * twiddle[1][k]);
    Re[k] = (Re_tmp[k] * twiddle[0][k]) - (Im_tmp[k] * twiddle[1][k]);
}
```

A implementação em hardware neste caso utiliza um algoritmo *in-place*, mesma técnica utilizada na iFFT em que os dados da memória são lidos e armazenados nela mesma. O processamento desta etapa utiliza quatro multiplicadores e dois somadores para as etapas de soma e subtração e é controlado pela Máquina de Estados apresentada na Figura 3.12. Neste caso, dois estados realizam o processamento enquanto um terceiro indica o fim da operação.

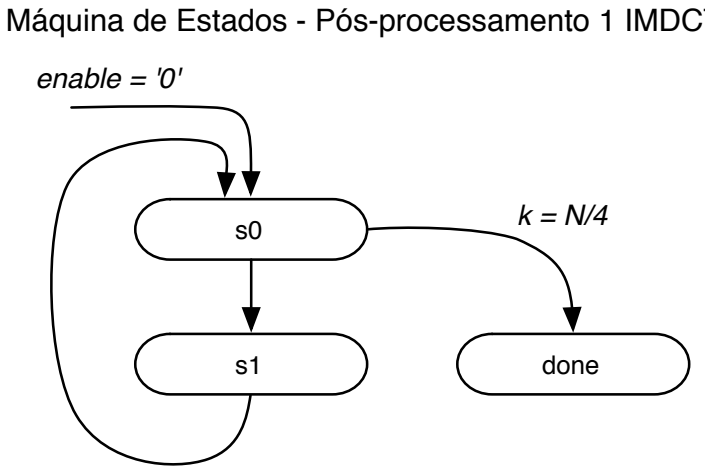

Figura 3.12: Diagrama da Máquina de Estados da primeira etapa do Pós-processamento da IMDCT

De maneira semelhante ao pré-processamento, as quatro multiplicações são realizadas no estado $s0$ enquanto a soma e a subtração são realizadas no estado $s1$. Os resultados são armazenados na memória no estado $s0$ cujo endereço é armazenado em uma variável $k\_auxiliar$ enquanto o valor de k é incrementado em $s0$ para acessar os valores das tabelas de $Twiddles$ bem como o próximo valor de Re e Im a serem utilizados nas multiplicações. Este processamento é realizado em N/2+1 ciclos e utiliza a abordagem *in-place* onde cada sequência Re e Im está armazenada e uma área de memória *single channel* própria de onde os dados são lidos e armazenados.

A segunda etapa do Pós-Processamento consiste em preparar os dados de saída para a etapa de Janelamento. Neste caso, os 512 ou 64 valores Reais e Imaginários serão transformados novamente em 2048 ou 256 amostras de áudio. A ordenação dos dados

é realizada conforme apresentado nas Equações 3.14 a 3.21, onde X é o vetor composto pelos valores Reais e Imaginários (Re e Im) e Y é a saída.

$$Y(2k) = Im\left[X\left(\frac{N}{8}+k\right)\right], \quad \text{para } 0 \leq k < \frac{N}{8} \tag{3.14}$$

$$Y(2k) = Re\left[X\left(\frac{N}{8}-k-1\right)\right], \quad \text{para } 0 \leq k < \frac{N}{8} \tag{3.15}$$

$$Y\left(\frac{N}{4}+2k\right) = Re\left[X(k)\right], \quad \text{para } 0 \leq k < \frac{N}{8} \tag{3.16}$$

$$Y\left(\frac{N}{4}+2k+1\right) = -Im\left[X\left(\frac{N}{4}-k-1\right)\right], \quad \text{para } 0 \leq k < \frac{N}{8} \tag{3.17}$$

$$Y\left(\frac{N}{2}+2k\right) = Re\left[X\left(\frac{N}{8}+k\right)\right], \quad \text{para } 0 \leq k < \frac{N}{8} \tag{3.18}$$

$$Y\left(\frac{N}{2}+2k+1\right) = -Im\left[X\left(\frac{N}{8}-k-1\right)\right], \quad \text{para } 0 \leq k < \frac{N}{8} \tag{3.19}$$

$$Y\left(\frac{3N}{4}+2k\right) = -Im\left[X(k)\right], \quad \text{para } 0 \leq k < \frac{N}{8} \tag{3.20}$$

$$Y\left(\frac{3N}{4}+2k+1\right) = Re\left[X\left(\frac{N}{4}-k-1\right)\right], \quad \text{para } 0 \leq k < \frac{N}{8} \tag{3.21}$$

A implementação em hardware é realizada por uma Máquina de Estados que acessa as memórias Re e Im de acordo com as Equações descritas. Seu diagrama é apresentado na Figura 3.13. Nele é possível observar que a cada dois estados, existe um estado de espera (*wait*) que foi introduzido para realizar a sincronia com as operações de janelamento e sobreposição apresentadas na seção 3.7.5.

As operações em ponto-fixo foram realizadas conforme a Tabela 3.14.

Tabela 3.14: Tabela de número de bits para operações de ponto-fixo do Pós-processamento da IMDCT

Dado	**Sinal**	Parte Inteira	Parte Fracionária
Entradas (Re e Im)	1	15	16
Twiddles	-	1	31
Armazenamento intermediário (Re e Im)	1	15	16
Saídas (Coef_Spec_out)	1	25	6

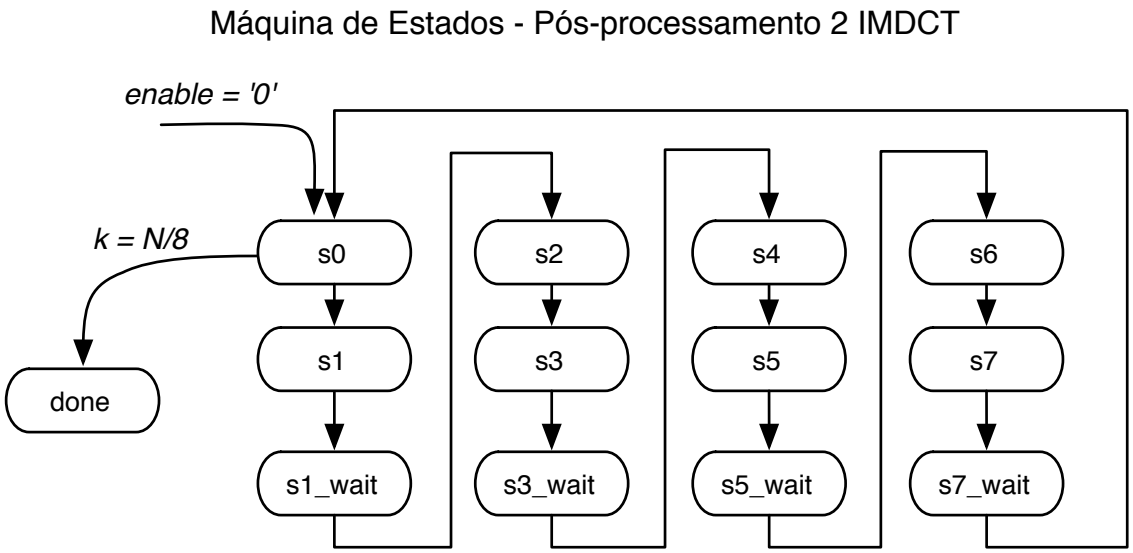

Figura 3.13: Diagrama da Máquina de Estados da segunda etapa do Pós-processamento da IMDCT

3.7.4 Integração da IMDCT

Após terem sido desenvolvidos e testados os 3 módulos da IMDCT separadamente, a etapa seguinte foi a de integração. O módulo integrador foi implementado com uma única máquina de estados que recebe como entrada o tamanho da janela e o sinal de *enable* e controla a execução sequenciada dos três processamentos até atingir o final do pós-processamento onde retorna um sinal de término (*done*). Para os três módulos desenvolvidos, as memórias Re e Im já haviam sido implementadas de modo externo e portanto foi necessário somente acrescentar os multiplexadores para controle de acesso às mesmas.

Porém, a integração também permite o compartilhamento de outros recursos como as memórias ROM dos *Twiddles* tanto das etapas de Pré quanto do Pós-Processamento, bem como os multiplicadores já que a execução dos três módulos não é paralelizável devido à dependência de dados entre eles. Neste caso, outros multiplexadores são adicionados para controle do endereçamento da ROM de *Twiddles* e controle das entradas dos multiplicadores. Além disso, a operação de *bit reverse* da iFFT, necessária pela restrição da memória (*single channel*), pôde ser suprimida modificando o endereçamento de saída do Pré-processamento para que já escreva nas memórias Re e Im nos endereços com bits invertidos, poupando área do módulo e tempo de processamento.

O resultado de integração é apresentado na Figura 3.14 onde a arquitetura de integração inicial é apresentada em (a) e a arquitetura otimizada em (b).

3.7.5 Janelamento, Sobreposição e Adição (*Windowing, Overlap and Add*)

Conforme descrito na seção 2.2.7, após a realização da IMDCT, a etapa de Janelamento (*Windowing*) aplica as funções Seno ou KBD às amostras de áudio conforme um dos quatro formatos de janela (*Only Long Sequence, Eight Short Sequence, Long Start Sequence e Long Stop Sequence*).

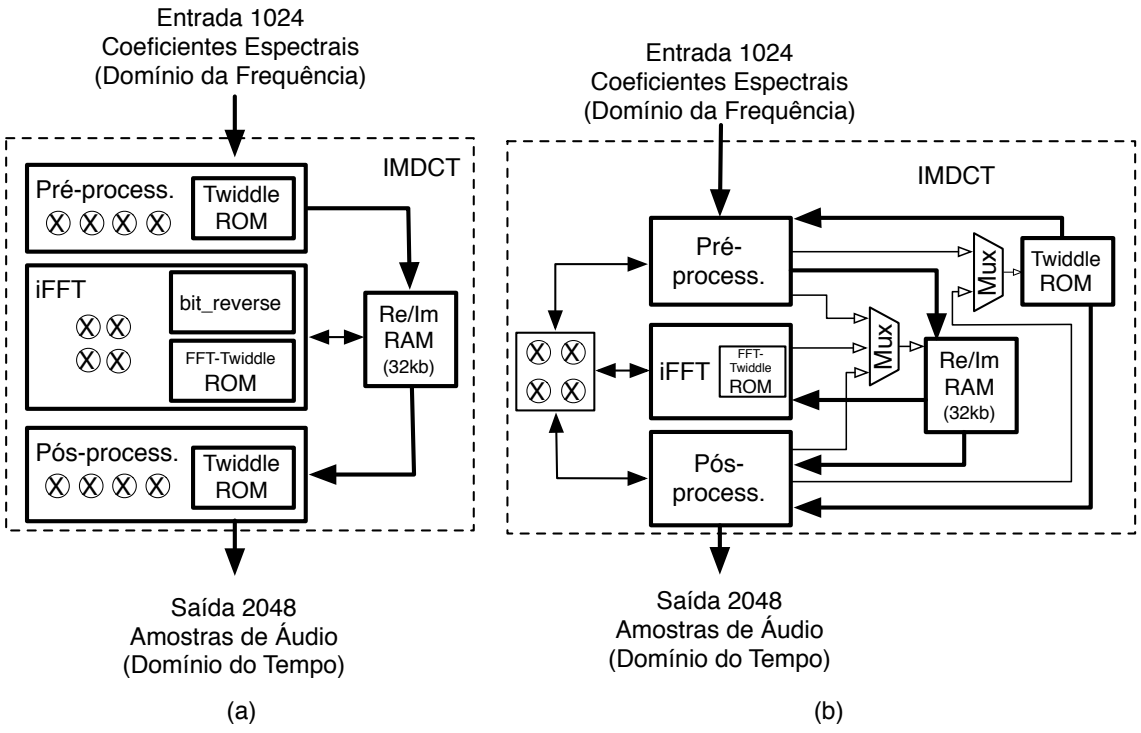

Figura 3.14: Diagrama da IMDCT completa

A implementação em hardware desta etapa foi realizada de modo que uma única Máquina de Estados controla o processamento da IMDCT, janelamento e sobreposição. Dependendo do formato da janela, especificado em *window_sequence*, a IMDCT é executada uma única vez no caso das janelas longas ou oito vezes em sequência no caso de janela curta.

Conforme pode ser visto no diagrama da máquina de estados que controla este processamento apresentado na Figura 3.15, o funcionamento do Banco de Filtros é iniciado pelo sinal de *enable* que coloca a máquina no estado $s0$ em que todos os sinais de controle e endereços são zerados. A máquina fica aguardando pelo sinal de *buffer_cheio* que indica quando o tocador está pronto para receber novos dados. Caso ainda esteja tocando as amostras da janela anterior, o processo fica parado conforme será apresentado na seção 3.11. Em seguida, de acordo com a variável *window_sequence* define-se o caminho a ser seguido pela máquina. Neste caso, os estados relativos aos três tipos de janela longa são executados em um fluxo integrado iniciado em $Long\_S1$ enquanto a sequência de estados da janela curta se inicia em $Short\_Init\_0$.

Para que a implementação fosse a mais rápida possível com poucos recursos de hardware, optou-se por realizar todo o processamento de Janelamento e Sobreposição em paralelo com a segunda máquina de estados do Pós-Processamento. Isso é possível e oportuno já que a segunda etapa do pós-processamento consiste puramente de reordenação dos dados na memória, e podemos então compartilhar os mesmos multiplicadores, reduzindo o requisito de área da arquitetura. Para realizar este processamento é necessário, porém, que a arquitetura esteja pronta para processar duas amostras com endereços distintos ao mesmo tempo já que a saída do pós-processamento fornece 2048 amostras, duas a cada 2 ciclos de *clock*.

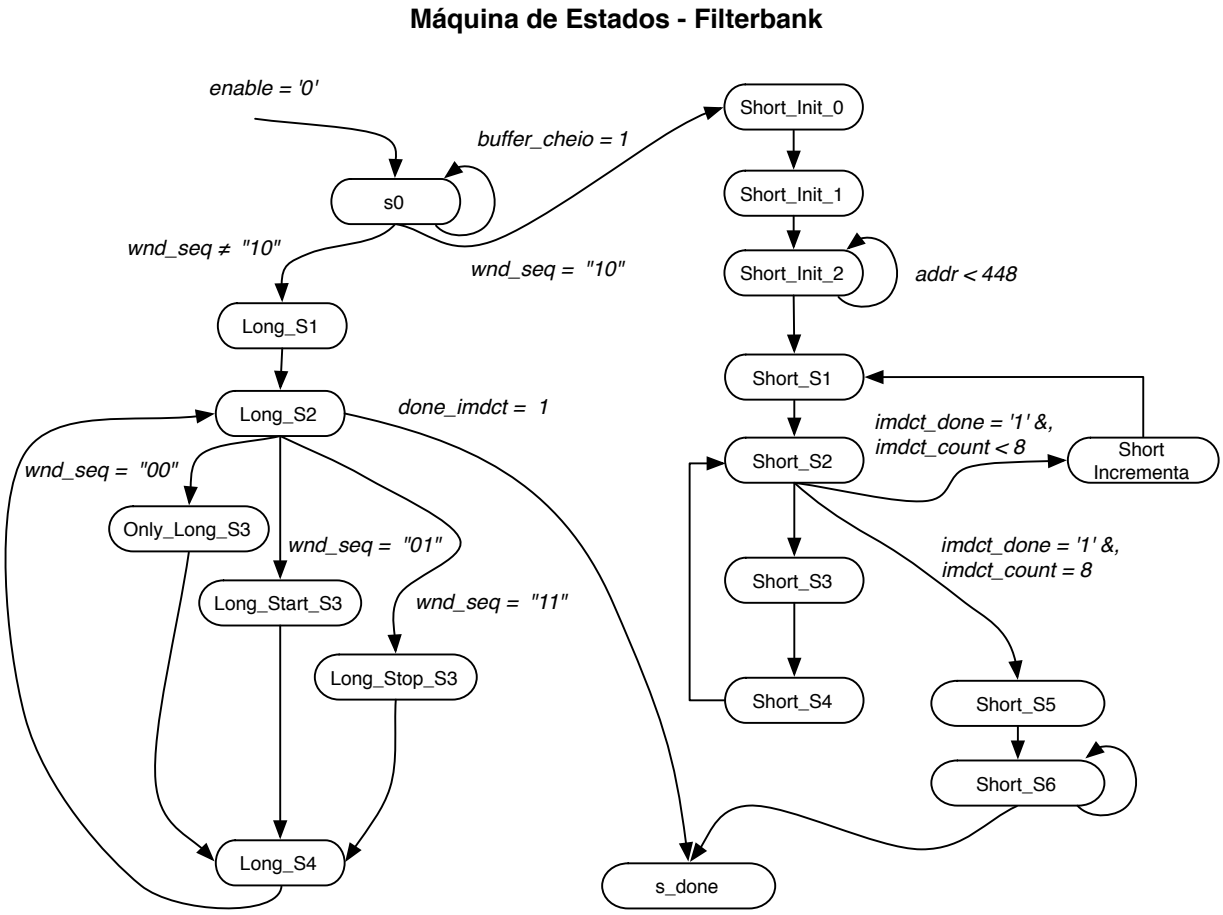

Figura 3.15: Diagrama de Blocos do Banco de Filtros Inverso

No caso das janelas Longas, o processamento é executado na sequência de 4 estados:

- $Long\_S1$ aciona a IMDCT;
- $Long\_S2$ aguarda o sinal de $write\_enable$ vindo da segunda máquina de estados do Pós-processamento, informando que as duas primeiras amostras estão prontas, juntamente com seus respectivos endereços que são utilizados para acessar a tabela de fatores do Janelamento. Caso receba o sinal de fim da IMDCT ($imdct\_done$), avança para o estado final s_done;
- $Long\_S3$ executa as multiplicações das amostras pelos fatores do janelamento, de acordo com a sequência definida ($Only\_Long\_Sequence$, $Long\_Start\_Sequence$ ou $Long\_Stop\_Sequence$);
- $Long\_S4$ realiza a soma da sobreposição com as amostras da janela anterior, armazenadas no $Overlap\_Buffer$.

Como a saída do pós-processamento não é sequencial, o Janelamento é auxiliado por um multiplexador que define os valores a serem utilizados como entradas dos multiplicadores dependendo do endereço transmitido na saída da IMDCT. A Figura 3.16 detalha a configuração dos multiplexadores no caso de Janelas Longas, convergindo para a operação de multiplicação do Janelamento, seguida da soma da sobreposição.

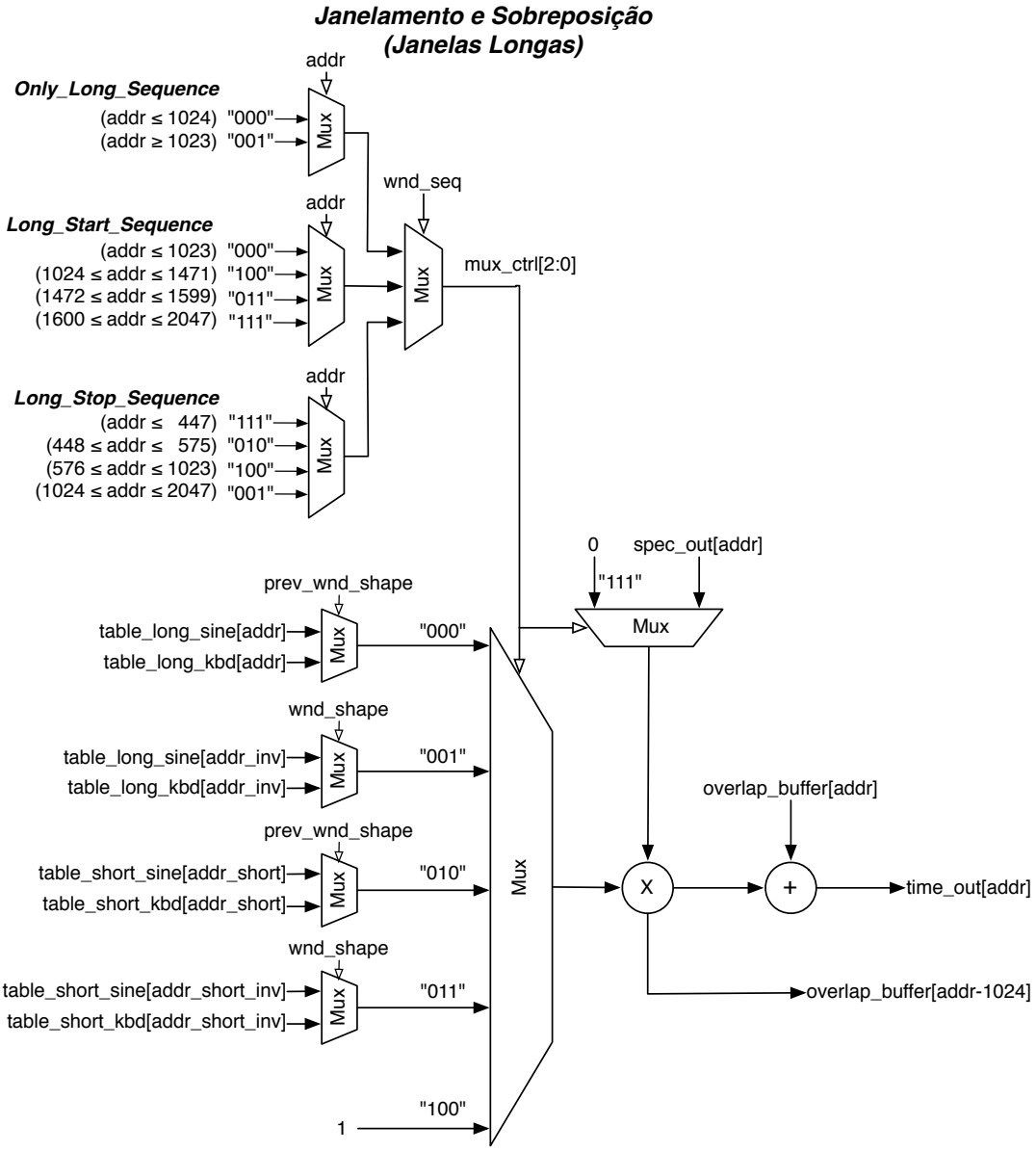

Figura 3.16: Diagrama dos Multiplexadores para Janelas Longas

No caso da janela curta, o processamento é semelhante ao das janelas longas, porém com variações conforme a listagem a seguir:

- Os primeiros dois estados $Short\_Init\_0$ e $Short\_Init\_1$ configuram os endereços;
- Em $Short\_Init\_2$ é realizada a sobreposição copiando o $Overlap\_Buffer$ dos endereços 0 a 447 para a saída já que a saída do Janelamento é igual a 0 neste caso;
- O estado $Short\_S1$ habilita a IMDCT de janela curta;
- Os estados $Short\_S2$, $Short\_S3$ e $Short\_S4$ realizam a operação de Janelamento e sobreposição sendo que em $Short_S1$ o contador é monitorado para indicar o fim da IMDCT e o estado $Short\_Incrementa$ é acionado para incrementar o contador do número de janelas curtas já realizados. Este mesmo contador de 3 bits é utili-

zado para complementar o endereçamento das memórias de *Overlap* e de saída das Amostras de áudio. Neste caso, os 3 bits são concatenados com o endereço de 7 bits de saída do pós-processamento formando, assim, um sinal de 10 bits capaz de endereçar toda a área de memória;

- *Short_S5* é usado para configurar o endereço do *Overlap_Buffer* em 576;
- *Short_S6* é usado para zerar os valores do Overlap Buffer do endereço 576 ao 1023.

A Figura 3.17 detalha os multiplexadores do Janelamento no caso das Janelas Curtas.

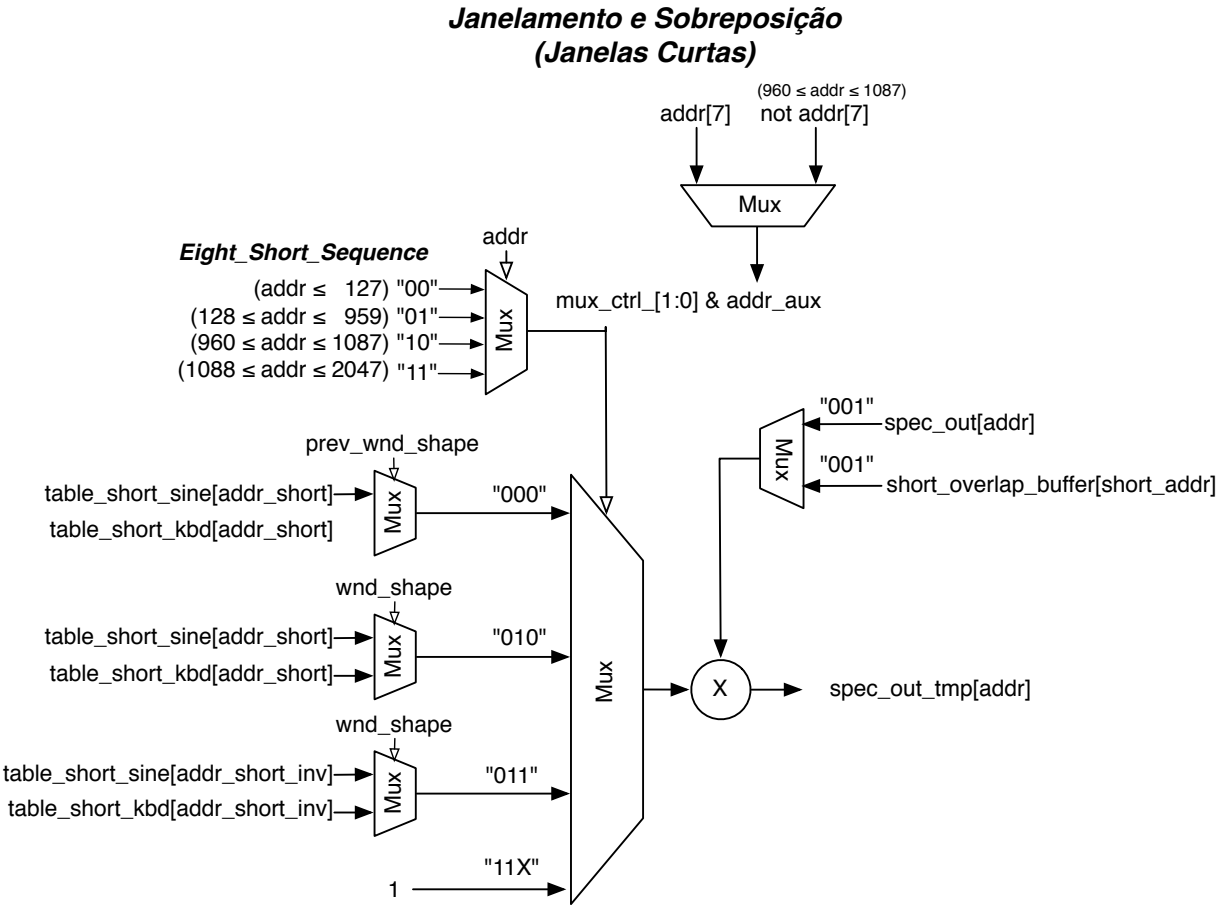

Figura 3.17: Diagrama dos Multiplexadores para Janelas Curtas

Para Janelas Curtas, os multiplexadores são diferentes pois o Janelamento é feito para cada bloco de 256 amostras. Neste caso, além da sobreposição de 50% da janela do bloco anterior (1024 amostras de áudio) temos a sobreposição entre cada janela curta de 256 amostras. Portanto, foi necessária a criação de uma área de memória para guardar as 128 amostras e efetuar a sobreposição dentro do bloco.

A arquitetura deste módulo é apresentada na Figura 3.18 e consiste em: uma máquina de estados que controla o processo; um módulo com multiplicadores compartilhados; a IMDCT; um módulo de Memória com os coeficientes do Janelamento (2 tabelas com 1024 coeficientes para janelas longas e 2 tabelas com 128 coeficientes para janelas curtas); uma área de memória para a Sobreposição de Janelas Longas e outra área para Janelas curtas).

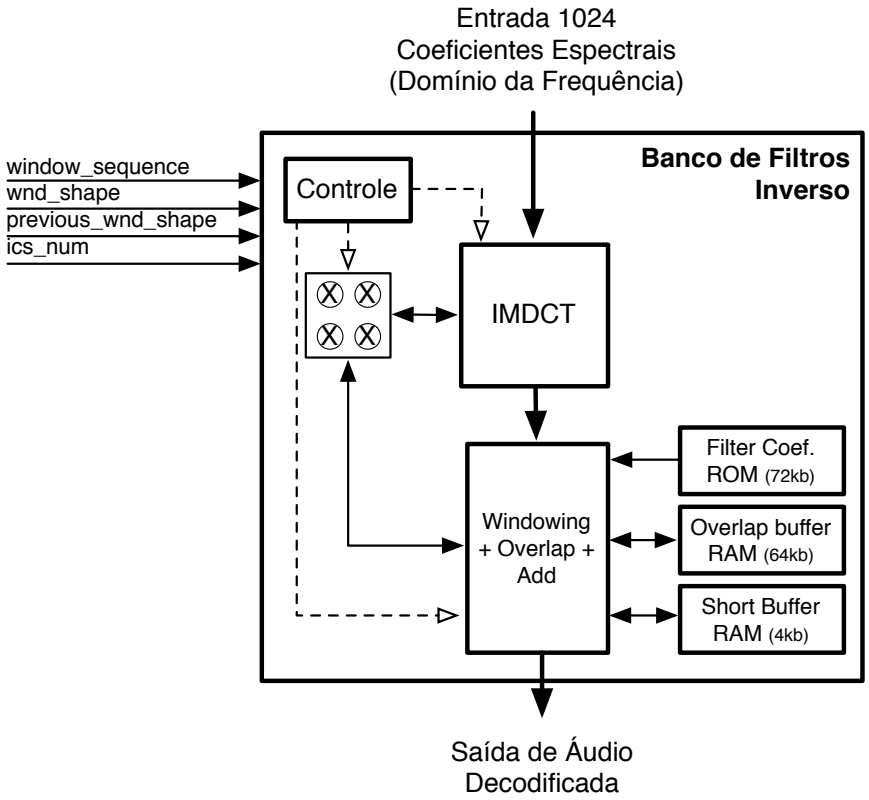

Figura 3.18: Diagrama de Blocos do Banco de Filtros Inverso

Finalmente, foi criado o *wrapper* para integrar o Banco de Filtros ao barramento Avalon. Semelhante ao Decodificador de Entropia, a conexão com o barramento Avalon é realizada no modo escravo (*Slave*) a partir de um multiplexador que aciona o controlador seguindo os comandos de leitura e escrita vindos do processador. Neste caso, os sinais de comunicação são:

1. *AVS_ADDRESS*: Endereço usado para acionar a função desejada no multiplexador (Neste caso com 3 bits de largura);
2. *AVS_WRITE*: Sinal de 1 bit indicando a solicitação de escrita vinda do barramento;
3. *AVS_WRITEDATA*: Sinal com largura de 32 bits contendo os dados vindos do processador;
4. *AVS_READ*: Sinal de 1 bit indicando a solicitação de leitura do barramento;
5. *AVS_READDATA*: Sinal com largura de 32 bits a ser preenchido pelo módulo a fim de ser lido pelo barramento no próximo ciclo de *clock*.

As operações em ponto-fixo foram realizadas conforme a Tabela 3.15 enquanto a Tabela 3.16 apresenta os resultados da síntese em FPGA com o uso de hardware do Módulo.

Tabela 3.15: Tabela de número de bits para operações de ponto-fixo do Janelamento e Sobreposição

Dado	**Sinal**	Parte Inteira	Parte Fracionária
Entradas Coef_Spec_out)	1	25	6
Fatores de Janelamento	-	1	31
Overlap_buffers (*Long* e *Short*)	1	15	-
Saídas	1	15	-

Tabela 3.16: Utilização de Hardware do Banco de Filtros Inverso

Módulo	**Elementos Lógicos**	**Memória (bits)**	**Multiplicadores Dedicados (9x9-bit)**
Filterbank_TOP (Windowing, Overlap and Add)	1192	0	-
IMDCT_TOP	400	0	
Pré-processamento	91	0	-
iFFT	503	0	-
Pós-processamento	427	0	-
Memória ROM Twidles Pré/Pós	-	32768	-
Memória ROM Twiddles iFFT	-	16384	-
Memória RAM - Real/Imag	-	32768	-
Memória ROM - Coefs Jan. Long	-	126976	-
Memória ROM - Coefs Jan. Short	-	15872	-
Memória RAM - Overlap Buffer	-	32768	-
Memória RAM - Overlap Buffer Short	-	2048	-
4 Multiplicadores Compartilhados	368	-	32
Total	**2981**	**264216**	**32**

3.8 Avaliação da solução de coprojeto

Tendo o Decodificador de Entropia, o Banco de Filtros e uma versão do *Stream Buffer* dedicada ao Decodificador de Entropia implementados em hardware, uma nova avaliação de desempenho foi realizada. Neste caso, utilizou-se a frequência de 50 MHz para os módulos em Hardware, uma vez que esta era a frequência máxima alcançada pelo Decodificador de Entropia, 100 MHz para o processador Nios II. Os resultados são apresentados na Tabela 3.17 onde se nota que a decodificação em tempo real ainda não é possível para dois canais. A nova performance é de 1,08 vezes o tempo real para o som estéreo.

Com estes resultados, podemos observar a necessidade de ainda diminuir em 7,4% o tempo total de processamento para canais estéreo e 69% para 6 canais. Os módulos que agora mais consomem tempo de processamento são o *parser* e as Ferramentas Espectrais.

Observando a arquitetura com mais detalhes, podemos identificar alguns pontos de gargalo.

1. O primeiro deles é o gargalo no barramento decorrente do fato do processamento ser feito em parte pelo processador com sua memória dedicada e outra parte em

Tabela 3.17: Performance de cada Etapa do Decodificador com o Decodificador de Entropia e o Banco de Filtros Inverso em hardware

Módulos	**Tempo (s)**	**Razão (%)**
Parser (*Bitstream*)	27,99	43,15 %
Decodificação de Entropia (Huffman + Quantização Inversa + Re-escalonamento)	7,08	10,91 %
Ferramentas Espectrais*	19,02	29,31 %
IS	4,64	7,14 %
MS	4,12	6,34 %
TNS	10,12	15,59 %
PNS	0,16	0,24 %
Banco de Filtros	10,79	16,63 %
Total	**64,89**	**100 %**

*A linha de "Ferramentas Espectrais" representa a soma de IS, MS, TNS e PNS.

Hardware dedicado, em que é necessário que os dados sejam enviados e retornados a cada etapa de processamento. Além disso, o processador ao trabalhar em ponto-flutuante enquanto o hardware trabalha em ponto-fixo faz com que um processo de conversão seja necessário. No caso do Decodificador de Entropia, os parâmetros de decodificação recuperados pelo parser são enviados ao módulo para processamento local e o mesmo terá que retornar um vetor de Fatores de Escala mais os Coeficientes Espectrais para utilização das Ferramentas Espectrais. Para o processamento do Banco de Filtros Inverso, são novamente transmitidos os Coeficientes Espectrais e finalmente retornadas as Amostras de Áudio para serem normatizadas e executadas por um tocador embutido no módulo do parser no processador onde são transmitidos novamente ao Tocador de Áudio da placa em hardware.

2. O segundo ponto de atraso é a existência de uma memória única no sistema que inviabiliza que tarefas de decodificação sejam executadas em paralelo por módulos distintos.

3. O terceiro ponto é a melhoria na eficiência de leitura do *bitstream* pelo parser, anteriormente comentada na seção 3.6, já que o mesmo foi implementado a princípio somente para o Decodificador de Entropia com saída para um único bit por leitura.

Tendo em vista estes problemas identificados, a proposta de solução descrita a seguir inclui:

1. A implementação em hardware das Ferramentas Espectrais, o que evitará que a saída do Decodificador de Entropia tenha de retornar pelo barramento e que a saída das próprias Ferramentas Espectrais possar ir diretamente para o Banco de Filtros Inverso;

2. A implementação de uma área de memória dedicada aos coeficientes espectrais para viabilizar a comunicação descrita no item anterior;

3. A implementação de área de memória dedicada para os parâmetros de decodificação que são largamente utilizados pelo Decodificador de Entropia e pelas Ferramentas Espectrais;

4. A modificação do *Stream Buffer* para atender às requisições de leitura de 1 a 16 bits do processador e;

5. Um *buffer* e controlador dedicado em Hardware para acionar o tocador de áudio da placa evitando que a saída do Banco de Filtros retorne pelo barramento bem como permita que o processador possa se dedicar a outra tarefa enquanto o som é tocado em paralelo.

O desenvolvimento das Ferramentas Espectrais partiu da análise do software de referência em C em que foram analisadas suas funções e fluxo de dados entre elas.

Conforme apresentado na seção 2.2.6 e apresentado na Figura 2.9, a sequência do processamento espectral ocorre após o término do Decodificador de Entropia depois de os Coeficientes Espectrais terem sido recuperados pela quantização inversa e pelo reescalonador. Em seguida são executadas as ferramentas *Perceptual Noise Substitution*, o *Mid/Side Stereo*, o *Intensity Stereo* e por fim o *Temporal Noise Shaping*. Todas estas ferramentas utilizam os parâmetros de decodificação extraídos pelo *parser* e armazenado na memória ICS. Finalmente, os Coeficientes Espectrais processados são enviados ao Banco de Filtros Inverso para serem convertidos em amostras de áudio e tocadas.

Para canais mono, o processamento espectral ocorre logo após uma única execução do Decodificador de Entropia. Para canais estéreo, há a necessidade de aguardar a recuperação dos Coeficientes Espectrais de ambos os canais para que o processamento espectral seja executado. No modo multicanal, a decodificação ocorre ao modo de grupos de canais mono, stereo ou uma mistura dos mesmo, sendo necessário aguardar no máximo a decodificação de dois canais para início do processamento.

Considerando o fluxo dos dados entre os módulos, fez-se necessário primeiro definir a arquitetura da memória compartilhada para depois implementar cada um dos módulos e integrá-los ao restante do decodificador.

3.9 Memória Compartilhada

Devido à dinâmica sequencial de processamento cuja ordem é PNS, MS, IS e TNS, e observando que cada Ferramenta Espectral deve ter acesso à leitura e escrita dos coeficientes espectrais e à leitura dos parâmetros de decodificação armazenados na memória ICS, propusemos a arquitetura de memória ilustrada na Figura 3.19. De acordo com a dinâmica de decodificação, é necessário espaço para armazenar dois canais ICS e dois canais de coeficientes espectrais.

A implementação das duas áreas de memória em hardware foi encapsulada em um módulo que controla o acesso chaveando as entradas e saídas de acordo com o módulo ativo no decodificador. O detalhe do encapsulamento desta memória é apresentado na Figura 3.20 onde se observa a interface com o *Parser* através do barramento Avalon e as interfaces diretas com os demais módulos do decodificador em hardware através dos multiplexadores.

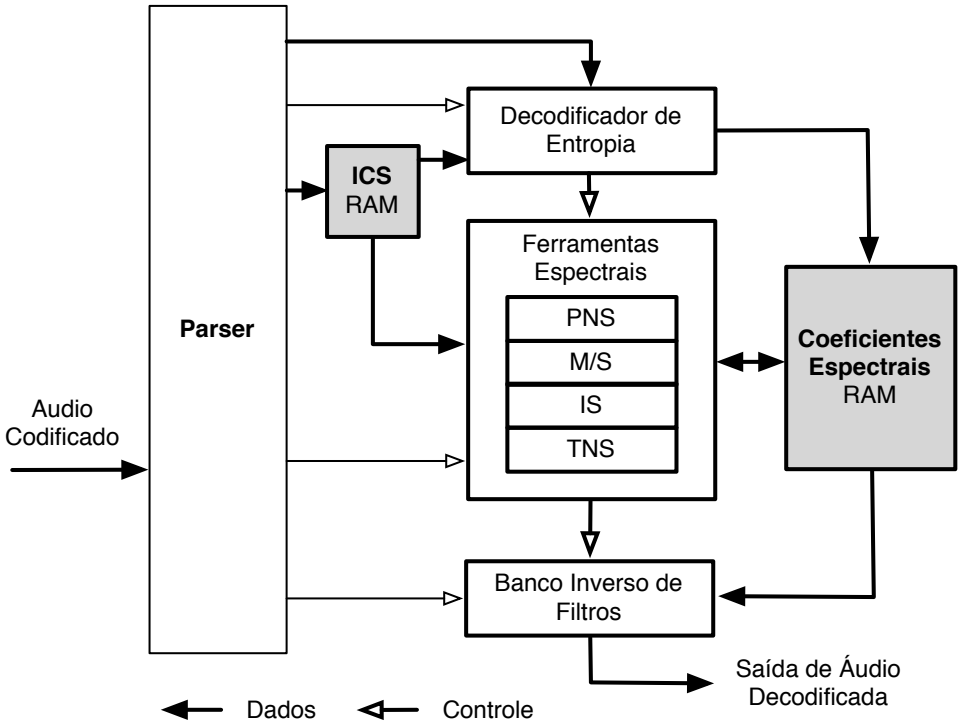

Figura 3.19: Arquitetura com as memórias compartilhadas

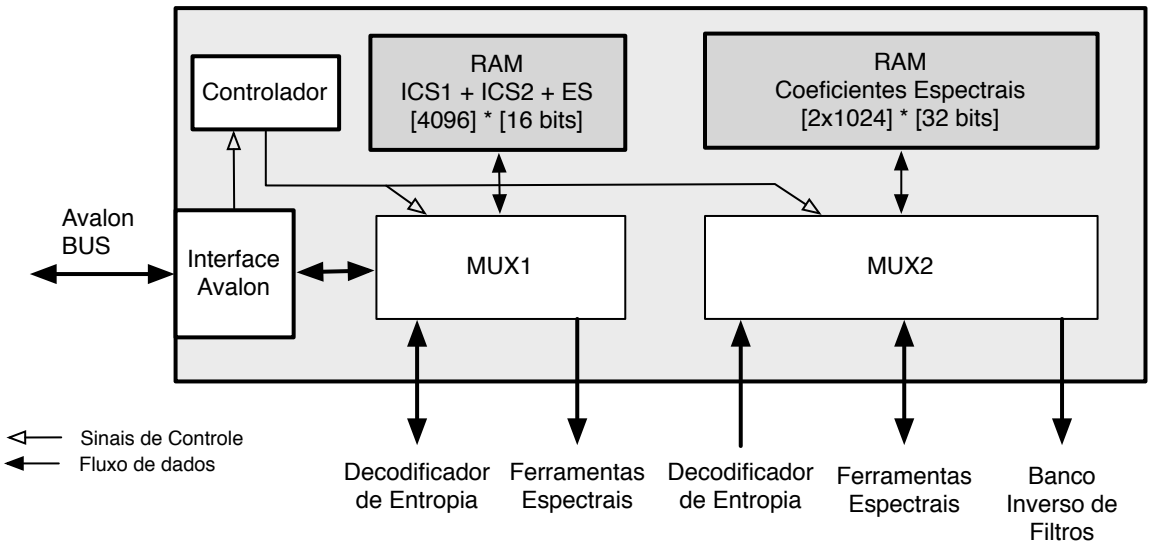

Figura 3.20: Arquitetura com as memórias compartilhadas

3.9.1 ICS RAM - *Individual Channel Stream*

Os parâmetros de decodificação são armazenados em uma estrutura denominada ICS (*Individual Channel Stream*) e são utilizados tanto pelo Decodificador de Entropia quanto pelas quatro Ferramentas Espectrais. Nessa estrutura existem variáveis que são compartilhadas entre os módulos e variáveis específicas para cada módulo do decodificador. Com o intuito de otimizar o espaço requerido para o armazenamento, estudou-se a estrutura de dados e a maneira como cada variável é acessada, mantendo assim, uma área compacta

de memória e um modo ágil de acesso.

De acordo com a Tabela 3.18, temos ao todo 40 variáveis na estrutura ICS num tamanho total de 53.832 bits ou 6.729 bytes. Como a implementação em hardware permite a manipulação de bits de maneira mais direta do que o C, foi feita uma verificação da largura exata em número de bits de cada uma das variáveis da estrutura. Neste caso, identificou-se que a largura de dados variou entre 1 e 14 bits sendo que 21,5% dos dados têm largura acima de 10 bits e 56,3% dos dados têm largura igual ou menor que 4 bits.

Tabela 3.18: Listagem das Variáveis da Estrutura ICS

	Tipo de Variável	Quantidade	Número total de Bits
Simples	unsigned short int	2	32
	unsigned char	21	168
Vetor Unidimensional	int [8]	2	512
	unsigned char [40]	1	320
	unsigned char [8]	2	128
	unsigned short int [52]	1	832
Vetor Bidimensional	int [8][4]	4	4096
	signed short[8][51]	1	6528
	unsigned short int [8][52]	1	6656
	unsigned char[8][128]	1	8192
	unsigned char[8][52]	3	9984
Vetor Tridimensional	short int [8][4][32]	1	16384
Total		40	53832

Analisando os blocos de memória no FPGA (Cyclone II), observou-se a possibilidade de implementar áreas de memória com largura de 1, 2, 4, 8, 9, 16, 18, 32 ou 36 bits. Para uniformizar o acesso de cada módulo à memória, escolhemos uma só largura de palavra que, no caso, foi a de 16 bits. Desse modo, foi possível acomodar todos os dados na mesma área. Porém, para otimizar o uso do espaço e a velocidade de acesso foram empregadas três otimizações.

1. Agrupamento de variáveis em um único endereço: as variáveis com larguras menores que 16 bits e normalmente acessadas pelo mesmo módulo foram agrupadas em um mesmo endereço de memória. Ex: *window sequence* (2 bits), *window_shape* (1 bit), *max_sfb* (6 bits) e *scale_factor_grouping* (7 bits) foram todas armazenadas no endereço 53 da área de memória. Ao todo são 16 bits de largura preenchendo todo o espaço disponível no endereço.

2. Agrupamento de vetores em um mesmo endereço de memória: vetores cuja largura das variáveis era menor que 8 bits foram agrupados armazenando mais de 1 variável por endereço ao modo do item 1. Ex: *tns_direction*[8][4] possui dado com 1 bit de largura e ao todo são 32 bits (8 x 4) de dados que foram armazenados nos endereços 126 e 127.

3. Inserção de variáveis de endereçamento simples em meio ao endereçamento de vetores longos: neste caso, aproveitando o fato de o endereçamento da variável vetorial bidimensional *sect_sfb_offset* ser de [8][52], observou-se que o acesso aos 52 itens,

endereçado por 6 bits (64 endereços) geraria uma área não aproveitada de 12 endereços não utilizados. Portanto, foram inseridas variáveis neste intervalo, aproveitando o espaço total da memória

A Figura 3.21 apresenta um trecho de exemplo da configuração final da memória ICS. Observa-se que todos os vetores bidimensionais e tridimensionais foram armazenados utilizando toda a largura de bits de cada endereço da memória.

Nome da Variável	Formato do dado na memória	Número de dados	largura do dado (bits)	End. Inicial	End. Final	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sect_sfb_offset[0][52]	(10 downto 0)	52	11	0	51																
element_instance_tag	(5 downto 1)	1	4	52	52																
common_window	(0)	1	1	52	52																
window_sequence	(1 downto 0)	1	2	53	53																
window_shape	(2)	1	1	53	53																
max_sfb	(8 downto 3)	1	6	53	53																
scale_factor_grouping	(15 downto 9)	1	7	53	53																
predictor_data_present	(0)	1	1	54	54																
predictor_reset	(1)	1	1	54	54																
predictor_reset_group_number	(6 downto 2)	1	5	54	54																
prediction_used[40]	[(39) .. (32)] [(31) .. (24)] [(23) .. (16)] [(15) .. (8)] [(7) .. (0)]	40	1	55	59																
ltp_data_present	(0)	1	1	60	60																
num_windows	(3 downto 0)	1	4	61	61																
num_window_groups	(7 downto 4)	1	4	61	61																
window_group_length[8]	(15 downto 12) (11 downto 8) (7 downto 4) (3 downto 0)	8	4	62	63																
sect_sfb_offset[1][52]	(10 downto 0)	52	11	64	115																
num_swb	(5 downto 0)	1	6	116	116																
swb_offset_max	(10 downto 0)	1	11	117	117																
global_gain	(7 downto 0)	1	8	118	118																
num_sec[8]	(5 downto 0)	8	6	119	122																
pulse_data_present	(0)	1	1	123	123																
tns_data_present	(1)	1	1	123	123																
gain_control_data_present	(2)	1	1	123	123																
sr_index	(3)	1	4	123	123																
tns.n_filt[8]	(1 downto 0)	8	2	124	124																
tns.coef_res[8]	(0)	8	1	125	125																
tns.direction[8][4]		32	1	126	127																
sect_sfb_offset[2][52]	(10 downto 0)	52	11	128	179																
tns.coef_compress[8][4]		32	1	180	181																
length_of_reordered_spectral_data		1	14	182	182																

Figura 3.21: Trecho do Mapa de Memória ICS

3.9.2 Memória de Coeficientes Espectrais

Em cada bloco de áudio (*raw data block*) temos 1024 amostras de coeficientes espectrais por canal. As Ferramentas Espectrais PNS e TNS são aplicadas a cada canal individualmente, enquanto o IS e MS, que usam o *Joint Stereo* para recuperar os dados, necessitam dos coeficientes de ambos os canais estéreo. Portanto, a estrutura da memória de coeficientes espectrais exige o armazenamento total de 2048 amostras de 32 bits. Além disso, esta memória deve permitir acesso à escrita para o Decodificador de Entropia, acesso à leitura/escrita para as 4 Ferramentas Espectrais e acesso à leitura para o Banco Inverso de Filtros, etapa final do decodificador.

3.10 Ferramentas Espectrais

Após a definição da área compartilhada de memória, o desenvolvimento de cada uma das ferramentas espectrais seguiu as seguintes etapas:

1. Estudo do algoritmo em C em ponto flutuante;
2. Modularização de cada função C apresentada no algoritmo e conversão do algoritmo para ponto-fixo;
3. Implementação de cada Módulo da Ferramenta Spectral em VHDL; Integração dos módulos VHDL;
4. Desenvolvimento de *test bench* para a validação do Modelo construído em VHDL;
5. Aplicação do *test bench* comparando os resultados das entradas e saídas do algoritmo em C com as entradas e saídas dos módulos em VHDL.

Em todos os 4 casos, a implementação em VHDL seguiu o padrão de interface externa de Leitura da Memória ICS (Vetor que guarda os parâmetros de decodificação de cada janela de áudio a ser decodificado), Leitura/Escrita na memória de Coeficientes Espectrais, sinal de *enable* e *done* para início e fim do cálculo. Além disso, as ferramentas seguiram o padrão de possuir 2 máquinas de estado em seu módulo *top*, uma controlando o acesso à memória ICS para a recuperação dos parâmetros de decodificação e outra para controlar as funções de cálculo.

Todas as simulações foram realizadas utilizando a ferramenta QuestaSim 6.6. Em todos os casos foram utilizados dados de 10 diferentes janelas para validar os resultados e sua comparação até que os mesmos atingissem um erro máximo de 0,01% em comparação aos resultados obtidos nos cálculos gerados pelo algoritmo em ponto flutuante em C.

3.10.1 IS - *Intensity Stereo*

A primeira ferramenta espectral desenvolvida foi o IS - *Intensity Stereo* que se baseia no princípio da localização do som e no fato do ouvido humano ter maior sensibilidade para distinguir o senso de direção para frequências mais baixas. Conforme apresentado na seção 2.2.6, o IS combina o espectro de maior frequência em um só canal, eliminando as diferenças de fase e transmitindo somente parte da informação de lateralidade do som. Neste caso, a compactação gera perdas e, portanto, não é possível reconstituir o sinal original.

A operação de decodificação do IS consiste em varrer o vetor de Coeficientes Espectrais e, para cada valor em que estiver presente o *flag* $IS\_INTENSITY$, copiar o valor de áudio do canal esquerdo para o direito, multiplicado-o pelo fator de escala. A implementação em VHDL foi realizada utilizando um módulo para cálculo dos valores dos coeficientes espectrais e um módulo *TOP* com duas máquinas de estado, uma para leitura da memória ICS com os parâmetros de decodificação e outra para implementar todos os laços do algoritmo.

Algoritmo 3.4: Trecho de código do Laço do IS

```
for (g = 0; g < icsr->num_window_groups; g++) {
  for (b = 0; b < icsr->window_group_length[g]; b++) {
    for (sfb = 0; sfb < icsr->max_sfb; sfb++) {
      if (is_intensity(icsr, g, sfb)) {
        scale = (float)(powf(0.5, (0.25 * icsr->sf[g][sfb])));
          for (i = icsr->swb_offset[sfb]; i < min(icsr->swb_offset[sfb+1],
                ics->swb_offset_max); i++) {
              r_spec[(group*nshort)+i] = (l_spec[(group*nshort)+i])*scale;
              if (is_intensity(icsr, g, sfb) != invert_intensity(ics, g, sfb))
                  r_spec[(group*nshort)+i] = -r_spec[(group*nshort)+i];
          }
      }
    }
    group++;
  }
}
```

Conforme pode ser observado no trecho de código em C apresentado pelo algoritmo 3.4, temos ao todo 4 laços sendo três externos e um interno. Os três laços externos varrem os parâmetros de decodificação *ms_mask_present* e *ms_used* procurando por trechos de áudio que sejam definidos com o *flag* IS_INTENSITY. A varredura é feita seguindo a quantidade de grupos de janelas (*num_windows_groups*), o tamanho de cada janela (*windows_groups_lenght*) e o número de bandas de fatores de escala transmitidos por grupo (*max_sfb*). O laço interno, acionado somente para os trechos onde o *flag* está habilitado, varre a memória de coeficientes espectrais, efetuando as operações de potência, multiplicação e cópia. No caso da potência, a operação segue a Equação 3.22 onde *sf* é o fator de escala usado para o referido trecho de coeficientes espectrais.

$$scale = \frac{1}{2}^{sf/4} \tag{3.22}$$

A implementação em VHDL foi realizada definindo as duas máquinas de estado, a de recuperação dos dados da memória e a dos laços de varredura e cálculo. Ambas operam em paralelo mas sincronizadas com ligações em determinados estados. A Figura 3.22 mostra o diagrama de fluxo de dados de ambas as máquinas. Cada item do diagrama representa na realidade um ou mais estados na implementação em VHDL.

No caso do Laço Interno, as operações consistem em calcular o valor da escala (*scale*), efetuar a multiplicação do coeficiente espectral do canal esquerdo pela escala e gravá-lo na mesma posição no canal direito. Ao invés de implementar a função de potência para o cálculo do *scale*, optou-se por criar uma tabela com os valores pré-calculados. Para isso, observou-se que os valores de fator de escala, que variam de -255 a 254, poderiam ser simplificados. Em primeiro lugar, observamos que a Equação 3.22 pode ser transformada na Equação 3.23.

$$scale = \frac{1}{2^{sf/4}} \tag{3.23}$$

Podemos simplificar ainda a operação observando que a divisão do expoente sf por 4 resultará em somente quatro valores decimais possíveis (0; 0,25; 0,5; 0,75) uma vez que sf é sempre um número inteiro. Neste caso, a operação de multiplicação do coeficiente espectral pelo valor de *scale* pode ser realizada em dois passos simples. O primeiro sendo um *shift* do coeficiente espectral (SpecCoef) com deslocamento igual ao valor inteiro de

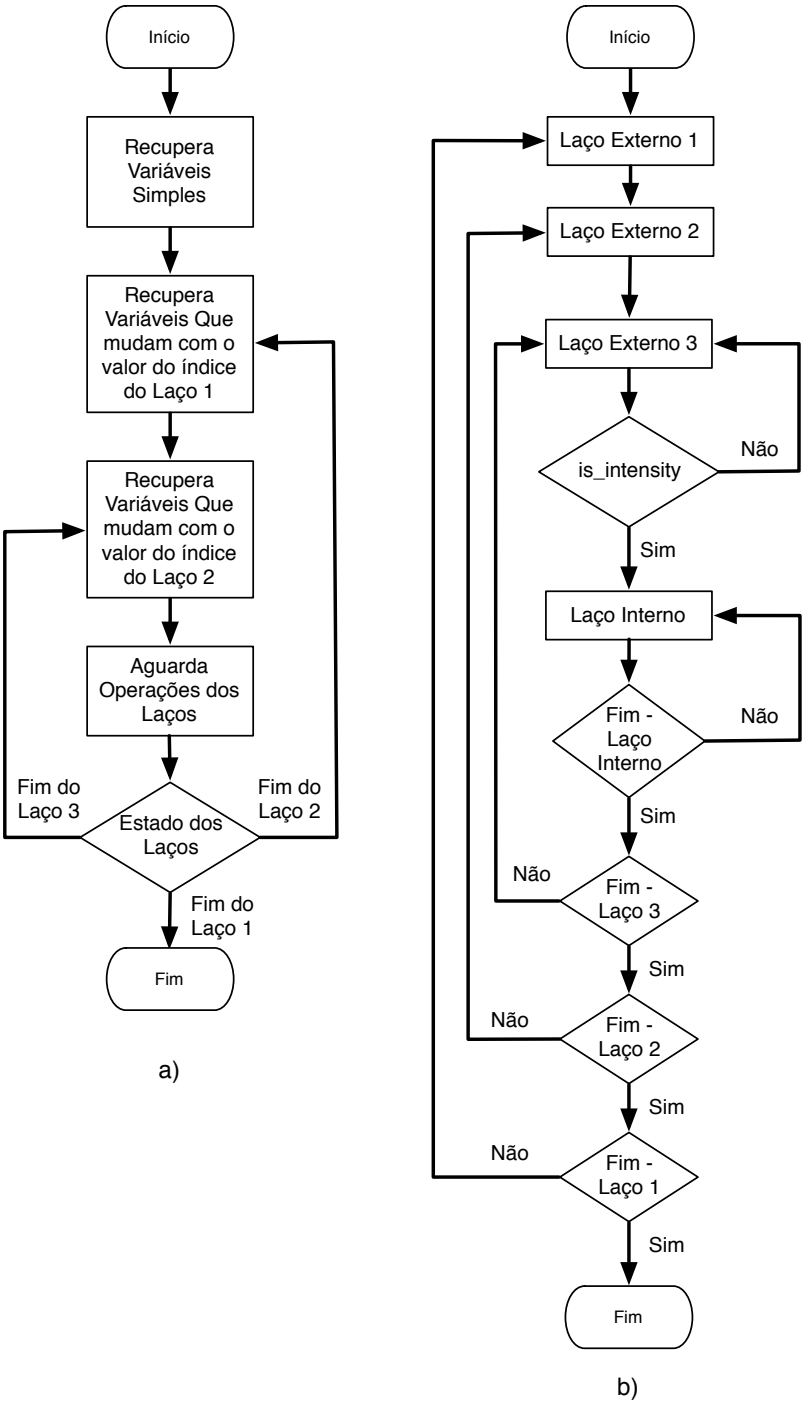

Figura 3.22: Diagrama de Fluxo de Dados: a) Máquina de acesso à memória. b) Máquina de Execução da operação IS.

$sf/4$. Shift right para valores positivos de sf e Shift left para valores negativos de sf. Em seguida, pré-calculamos os valores fracionários de *scale* (*scale_frac*), ou seja, $1/2^{(0)}$, $1/2^{(0,25)}$, $1/2^{(0,5)}$ e $1/2^{(0,75)}$ e multiplicamos pelo resultado do deslocamento anterior. Neste

caso, a operação final é apresentada na Equação 3.24.

$$SpecCoef\_Dir = SpecCoef\_Esq(srl \text{ ou } sll \text{ por } sf/4) \cdot scale\_frac \tag{3.24}$$

Para exemplificar o acesso à memória ICS efetuado internamente em cada uma das Ferramentas Espectrais, apresentamos os trecho de código VHDL pelo algoritmo 3.5.

Algoritmo 3.5: Trecho de código VHDL contendo o processo de acesso à mamória ICS

```
acesso_ics: process(clk, g)
variable ICS_DATA_IN_VAR    : std_logic_vector(15 downto 0);
variable sfb_tmp    : std_logic_vector(5 downto 0);
begin
   if (enable = '0' or reset = '1') then
    var_atualizadas <= '0';
    state_mem <= s0;
   elsif (clk'event and clk='1') then
    case state_mem is

    when s0 =>
        ics_addr <= M_ICS2 & M_NUM_WINDOW_GROUPS;
        state_mem <= s1;
    when s1 =>
        num_window_groups <= ics_data_in(7 downto 4);
        ics_addr <= M_ICS2 & M_MAX_SFB;
        state_mem <= s2;
    when s2 =>
        max_sfb <= ics_data_in(8 downto 3);
        ics_addr <= M_ICS1 & M_SWB_OFFSET_MAX;
        state_mem <= s3;
    when s3 =>
        swb_offset_max <= ics_data_in(10 downto 0);
        ics_addr <= M_ICS1 & M_MS_MASK_PRESENT;
        state_mem <= s4;
    when s4 =>
        ms_mask_present <= ics_data_in(0);
        state_mem <= s5;

    when s5 =>
        -- Define ADDR da proxima variavel (ICS + ADDR_VAR + POSICAO_MEMORIA)
        ics_addr <= M_ICS2 & M_WINDOW_GROUP_LENGTH(10 downto 1) & g(2);
        state_mem <= s6;
    when s6 =>
        ICS_DATA_IN_VAR := ics_data_in;
        ICS_DATA_IN_VAR := to_stdlogicvector(to_bitvector(ICS_DATA_IN_VAR) srl to_integer(
            unsigned(g(1 downto 0) & "00")));
        window_group_length <= ICS_DATA_IN_VAR(3 downto 0);
        state_mem <= s7;

    when s7 =>
        ics_addr <= M_ICS2 & (M_SFB_CB + (g & sfb(5 downto 2)));
        state_mem <= s8;
    when s8 =>
        ICS_DATA_IN_VAR := ics_data_in;
        ICS_DATA_IN_VAR := to_stdlogicvector(to_bitvector(ICS_DATA_IN_VAR) srl to_integer(
            unsigned(sfb(1 downto 0) & "00")));
        sfb_cb <= ICS_DATA_IN_VAR(3 downto 0);
        ics_addr <= M_ICS2 & (M_SF + (g & sfb));
        state_mem <= s9;
    when s9 =>
        sf <= ics_data_in(8 downto 0);
        ics_addr <= M_ICS2 & (M_SWB_OFFSET + sfb);
        sfb_tmp := sfb + 1;
        state_mem <= s10;
    when s10 =>
        swb_offset <= ics_data_in(10 downto 0);
```

```
            ics_addr <= M_ICS2 & (M_SWB_OFFSET + sfb_tmp);
            state_mem <= s11;
        when s11 =>
            swb_offset_1 <= ics_data_in(10 downto 0);
            ics_addr <= M_ICS1 & (M_MS_USED + (g & '0' & sfb(5 downto 4)));
            state_mem <= s12;
        when s12 =>
            ICS_DATA_IN_VAR := ics_data_in;
            ICS_DATA_IN_VAR := to_stdlogicvector(to_bitvector(ICS_DATA_IN_VAR)
                      srl to_integer(unsigned(sfb(3 downto 0))));
            ms_used <= ICS_DATA_IN_VAR(0);
            state_mem <= s_done_wait;
        when s_done_wait =>
            state_mem <= s_done;
        when s_done =>
            if (atualizar_vars = "01")    then state_mem <= s5;
            elsif (atualizar_vars = "10") then state_mem <= s7;
            else state_mem <= s_done;
            end if;
        when others => null;
        end case;
        end if;
    end process;
```

No trecho de código acima podemos observar que os estados $s0$ a $s4$ são acionados somente uma vez após o *reset/enable* e recuperam os valores de num_window_groups, max_sfb, swb_offset_max e ms_mask_present. Em cada estado, o endereço da próxima variável é definido na porta ics_addr e o valor da variável atual é recuperado pela porta ics_data_in, neste caso utilizando o acesso direto aos bits correspondentes à posição da variável no endereço de memória, ou seja, o mapa de memória é *hardcoded*.

Os estados $s5$ e $s6$ são utilizados para atualizar as variáveis que mudam com o índice g, enquanto os estados $s7$ a $s12$ atualizam as variáveis que dependem de ambos g e sfb. No estado $s5$ temos um exemplo de acesso a uma variável vetorial unidimensional que está armazenada na memória ICS nos endereços 62 e 63 conforme a Figura 3.21. Além disso, há 4 dados por endereço de memória. Portanto, a definição do endereço de memória é realizada considerando o canal a ser acessado, neste caso o canal 2, identificado pela constante *M_ICS2*. Em seguida pela constante que define o endereço da variável na memória ICS, no caso, *M_WINDOW_GROUP_LENGTH*. Como esta variável utiliza dois endereços de memória, foi definido a ela o início em um endereço par e o último bit do endereço varia de acordo com o bit mais significativo de g (que tem largura de 3 bits). Neste caso, o código é definido por:

```
    --Teste
      ics_addr <= M_ICS2 & M_WINDOW_GROUP_LENGTH(10 downto 1) & g(2);
```

Finalmente, para acessar o dado específico, que tem largura de 4 bits, utilizamos os dois bits menos significativos de g para definir o valor do Shift Right a ser aplicado no resgate do dado. Este valor é multiplicado por 4 acrescentando "00" ao final. O código final é:

```
    --
    window_group_length <= to_stdlogicvector(to_bitvector(ics_data_in) srl to_integer(unsigned(
        g(1 downto 0) & "00")));
```

O estado final *s_done* fica em *loop* aguardando a outra máquina solicitar atualização nos valores das variáveis de memória.

A máquina de estados que controla o algoritmo IS é apresentada no trecho de código da Figura 3.23. Neste, podemos observar os estados $s1$, $s2$ e $s3$ que controlam o início

e fim dos três laços seguidos pelos respectivos estados $s1\_wait$, $s2\_wait$ e $s3\_wait$ que aguardam a resposta de atualização da máquina de estados de acesso à memória ICS. O estado $s3$ verifica o *flag* $IS\_INTENSITY$ para acionar ou não o módulo de cálculo do IS, enquanto os estados $calcula\_is\_1$ a $calcula\_is\_5$ interagem com o módulo IS_INT e realizam a sincronização dos cálculos deste módulo com a entrada e saída de dados para as memórias dos Coeficientes Espectrais.

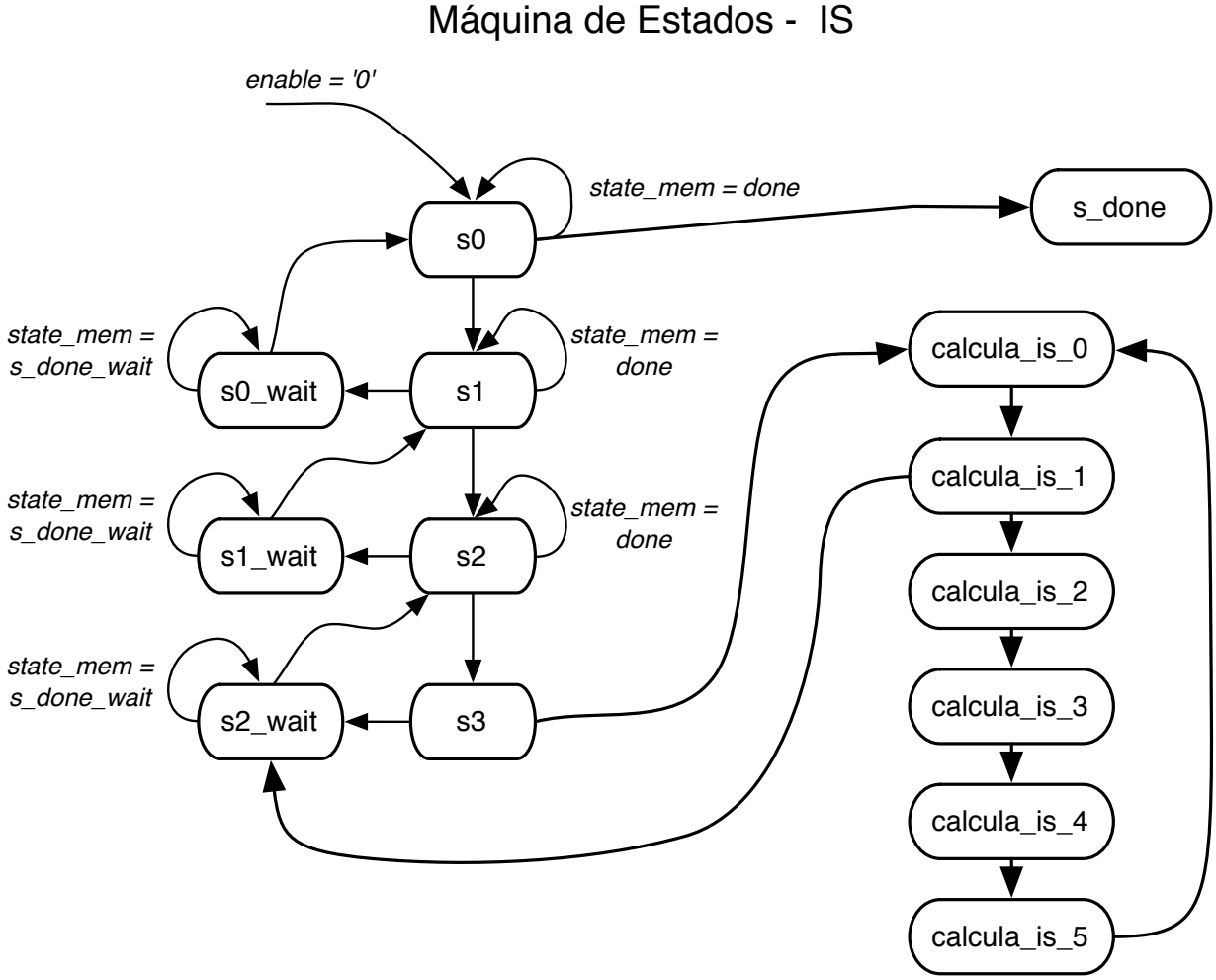

Figura 3.23: Diagrama de estados da Máquina de Estados que controla o IS

3.10.2 MS - *Mid/Side Stereo*

A segunda ferramenta espectral implementada foi o *Mid-Side Stereo* que, na etapa de codificação, soma os canais esquerdo e direito em um só canal central (*Mid*) e transforma a diferença entre eles em um outro canal (*Side*). Esta é uma técnica de compactação sem perda de informações.

A operação de decodificação do MS consiste em varrer o vetor de Coeficientes Espectrais e, para cada valor em que estiver presente o *flag* MS_USED, efetuar as operações de soma e subtração para reconstituir os canais esquerdo (*Left*) e direito (*Right*).

Algoritmo 3.6: Trecho de código do Laço do MS

```
for (g = 0; g < ics->num_window_groups; g++) {
    for (b = 0; b < ics->window_group_length[g]; b++) {
        for (sfb = 0; sfb < ics->max_sfb; sfb++) {
            if ((ics->ms_used[g][sfb] || ics->ms_mask_present == 2) &&
                !is_intensity(icsr, g, sfb) && !is_noise(ics, g, sfb))
            {
                for (i = ics->swb_offset[sfb]; i < min(ics->swb_offset[sfb+1],
                    ics->swb_offset_max); i++)
                {
                    k = (group * nshort) + i;
                    tmp = l_spec[k] - r_spec[k];
                    l_spec[k] = l_spec[k] + r_spec[k];
                    r_spec[k] = tmp;
                }
            }
        }
        group++;
    }
}
```

Conforme pode ser observado no algoritmo 3.6 em C, do mesmo modo do IS, o MS também possui 4 laços sendo três externos e um interno. Os três laços externos varrem os parâmetros de decodificação *ms_ mask_ present*, *ms_ used* e *sfb_ cb*, procurando por trechos de áudio que sejam codificados com a ferramenta MS. A varredura é feita doe modo semelhante ao IS, seguindo as variáveis *num_ windows_ groups*, *windows_ groups_ lenght* e *max_ sfb*. O laço interno, acionado somente para os trechos habilitados com MS, varre a memória de coeficientes espectrais, efetuando as operações de soma e subtração, invertendo a operação do codificador.

A implementação em VHDL foi realizada utilizando um só módulo com duas máquinas de estado, uma para leitura da memória ICS com os parâmetros de decodificação e outra para implementar todos os laços do algoritmo. Ambas as máquinas funcionam de maneira semelhante às apresentadas para a ferramenta IS, com exemplos já apresentados anteriormente.

3.10.3 TNS - *Temporal Noise Shaping*

No caso do *Temporal Noise Shaping*, sua implementação em VHDL consiste das duas máquinas de estado, a de acesso à memória ICS, com a mesma estrutura da apresentada para o IS e a de controle do algoritmo. Esta segunda máquina controla os três módulos de cálculo: Cálculo de LPC, Cálculo do Intervalo *Start/End* e Aplicação do Filtro.

O algoritmo de Cálculo dos coeficientes LPC usa os parâmetros do TNS armazenados na estrutura ICS como referência e executa três laços sequenciais com tamanho igual à ordem (N) do filtro para gerar os N coeficientes LPC. O primeiro laço cria o vetor de coeficientes de tamanho N e decodifica os parâmetros TNS gerando coeficientes em valores com sinal. O segundo laço executa uma quantização inversa nos coeficientes. O terceiro laço por sua vez possui dois laços sequenciais internos usados para converter os valores em coeficientes LPC através de várias iterações. Em VHDL, foi possível incorporar os dois primeiros laços ao terceiro executando todo o cálculo dos coeficientes LPC em um só passo, simplificando a complexidade do algoritmo.

O cálculo dos índices *Start* e *End* dos intervalos consiste em definir o mínimo entre três valores e utilizar este resultado como índice de um vetor que será novamente comparado

a outro valor para extrair o mínimo entre ambos. Apesar do procedimento ser igual para *Start* e *End*, a restrição de acesso ao vetor *max_tns_sfb* impediu o cálculo de ambos os valores em paralelo.

Finalmente, a aplicação do Filtro utiliza os coeficientes LPC e uma série de somas e multiplicações executadas em dois laços encadeados para modificar os valores dos Coeficientes Espectrais.

3.10.4 PNS - *Perceptual Noise Substitution*

A ferramenta PNS, última a ser implementada, tem função de adicionar ao áudio de saída todo o ruído branco removido na etapa de codificação. O PNS funciona a partir de um gerador de números aleatórios que são multiplicados por um fator de escala que normaliza o valor médio da energia do som na saída das amostras adequando-as ao trecho de áudio da janela. Neste caso, em cada bloco de áudio, todos os espaços com áudio igual a zero e que tenham o parâmetro NOISE (ruído) são substituídos.

O algoritmo do PNS utiliza os mesmos três laços externos do IS executando a varredura nos parâmetros de decodificação a partir das variáveis e *num_windows_groups*, *windows_groups_lenght* e *max_sfb*. Porém, neste caso, procura pelos trechos de áudio que são definidos como *IS_NOISE*, ou seja, ruídos para definir quais coeficientes espectrais devem ser substituídos por sinais aleatórios. Uma vez identificado o trecho, a função *gen_rand_vector* gera valores aleatórios de acordo com o nível de energia definido pelo parâmetro *scale_factor* e os grava nas respectivas posições dos coeficientes espectrais. O PNS funciona tanto para um canal mono quanto para um par de canais estéreo.

O trecho de código em C apresentado no algoritmo 3.7 representa a ideia fundamental básica por trás da ferramenta. Nele podemos ver a existência de dois laços sendo que o primeiro se encarrega de gerar os números aleatórios e aplicar um fator de escala de acordo com a magnitude definida pelo codificador. Além disso, a energia acumulada é calculada durante o laço e ao final, um novo fator de escala é calculado e utilizado para normalizar o valor médio de energia para o trecho de coeficientes espectrais.

Algoritmo 3.7: Trecho de código do Gerador de valores aleatórios

```
void gen_rand_vector(float *spec, signed short int scale_factor, unsigned short int size,
                                    unsigned int sub)
{
    unsigned short int i;
    float energy = 0.0;

    float scale = (float)1.0/(float)size;

    for (i = 0; i < size; i++)
    {
        float tmp = scale*(float)(int)ne_rng();

        spec[i] = tmp;
        energy += tmp*tmp;
    }

    scale = (float)1.0/(float)my_sqrt(energy);
    scale *=  pow_sf[scale_factor];
    for (i = 0; i < size; i++)
    {
        spec[i] *= scale;
    }
}
```

A implementação em VHDL consiste em quatro módulos de cálculo (Gerador de Números Aleatórios, Raiz Quadrada, Divisor, Potência de 2), um módulo de integração dos módulos de cálculo, e um módulo TOP responsável por gerenciar o acesso à memória compartilhada ICS e efetuar a integração do PNS com o restante das ferramentas espectrais.

O gerador de números aleatórios na verdade é implementado com um gerador de números pseudo-aleatórios de 16 bits que utilizam uma semente (*seed*) no momento em que o decodificador é ligado. Em hardware, este gerador produz um valor diferente por ciclo de *clock*.

O processo de geração de uma vetor de números aleatórios para serem substituídos no trecho de áudio começa por armazenar os N valores (N = tamanho do trecho a ser preenchido) e ao mesmo tempo ir acumulando em um registrador o valor de cada número ao quadrado. Este número acumulado representa a quantidade de energia do trecho de áudio. Finalmente, a raiz quadrada desta energia é calculada e o valor do fator de escala é utilizado para adequar o vetor de números aleatórios à quantidade total de energia requerida no trecho de áudio. Por fim, os valores são gravados nas respectivas posições dos coeficientes espectrais.

3.10.5 Integração das Ferramentas

Após implementar e testar cada um dos módulos, a tarefa final consistiu em integrar todos eles em um único módulo para funcionar junto ao restante do decodificador. Neste caso, utilizamos uma arquitetura simples com uma máquina de estados controlando a ordem de execução seqüencial de cada ferramenta espectral e os respectivos multiplexadores que controlam o acesso aos módulos de memória externos.

A Tabela 3.19 apresenta os resultados da síntese em FPGA com o uso de hardware do Módulo.

Tabela 3.19: Utilização de Hardware da Ferramentas Espectrais

Módulo	**Elementos Lógicos**	**Memória (bits)**	**Multiplicadores Dedicados (9x9-bit)**
Spectral Tools (TOP)	590	-	-
Intensity Stereo	1007	-	14
Mid/Side Stereo	270	-	-
Temporal Noise SHaping	1099	6080	16
Perceptual Noise Substitution	2584	1024	40
Total	**5550**	**7104**	**70**

A arquitetura final dos módulos das Ferramentas Espectrais pode ser visualizada na Figura 3.24.

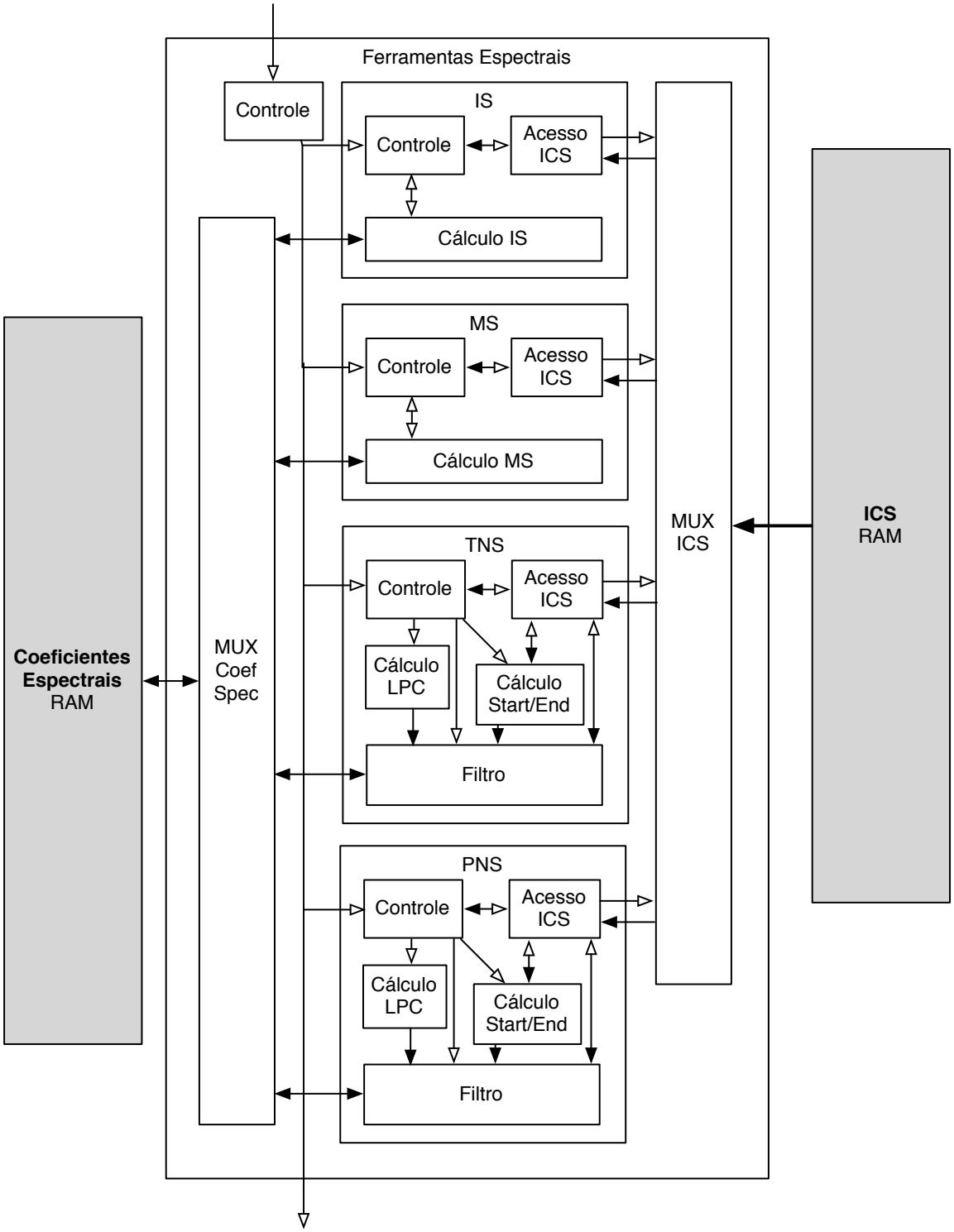

Figura 3.24: Diagrama das Ferramentas Espectrais

3.11 *Buffer* de Saída e Tocador de Áudio

Conforme a entrada sequencial dos dados de áudio codificados no *bitstream*, cada canal é decodificado individualmente e a etapa final do Banco de Filtros é realizada um canal por vez. Portanto, para que o decodificador possa tocar o áudio sem pausas durante o processo de decodificação, é necessário que as amostras de áudio sejam armazenadas em um *buffer* de saída. Neste processo, as 1024 amostras de cada canal são armazenadas no *buffer* e assim que todos os canais de um *raw data block* tenham sido decodificados, podem ser enviados ao tocador para que sejam reproduzidos.

Em paralelo às amostras que estejam sendo tocadas, uma segunda área do *buffer* deve estar disponível para que o decodificador possa armazenar as amostras dos canais do próximo *raw data block*.

A implementação do *Buffer* em Hardware considerou a limitação do tocador de saída da placa que só é capaz de reproduzir dois canais simultâneos. Neste caso, foram criadas duas áreas de memória capazes de armazenar 2048 amostras de áudio de 16 bits.

Apesar desta limitação, o decodificador é capaz de processar até 6 canais em tempo real, porém, neste caso, as amostras dos canais excedentes são descartadas nesta implementação.

A Figura 3.25 apresenta o diagrama de blocos do *Buffer* das amostras de áudio. Nele estão presentes duas máquinas de estado, uma para controle da entrada de dados vindos do Banco de Filtros Inverso e outra para controle da saída das amostras para a entrada do tocador que recebe os dados através de uma FIFO. As duas áreas de memória são chaveadas de modo que o papel de ambas alterna a cada ciclo de decodificação.

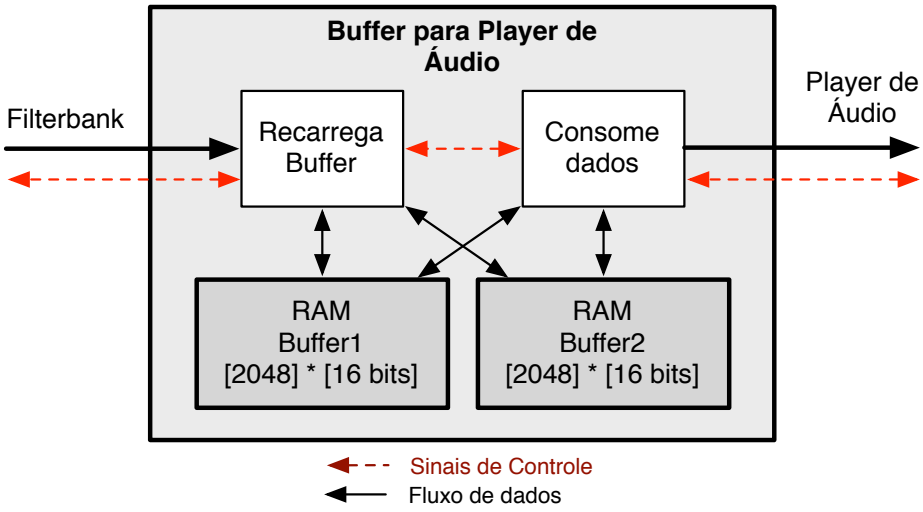

Figura 3.25: Arquitetura com as memórias compartilhadas

O tocador de áudio utilizado é o módulo *Audio Interface* disponibilizado com a placa DE2-70. Em sua versão original, o mesmo foi criado para receber dados exclusivamente via barramento. Portanto, foi necessário editar seu código para que o mesmo pudesse receber ao mesmo tempo dados do barramento ou dados diretamente do *Buffer*.

3.12 LATM/LOAS and MP4 Decoders

As últimas duas etapas de decodificação implementadas foram os decodificadores do protocolo de multiplexação LATM (*Low Overhead Audio Transport Multiplex*) do protocolo de transporte e sincronização LOAS (*Low Overhead Audio Stream*) e do formato de armazenamento de arquivos MP4FF (MPEG-4 *File Format*). Ambas as implementações feitas por software em C.

Os dois primeiros, LATM e LOAS, são combinados em um decodificador único de duas camadas. Delas, são extraídos os elementos *payload* que compõem os *raw data blocks* e parâmetros de configuração do decodificador como a taxa de amostragem para configurar o tocador de áudio.

No caso do MP4, um outro decodificador em software foi implementado. Como o formato do *container* pode armazenar diversos tipos de objetos incluindo dados de vídeo (MPEG-4 (H.264), MPEG-1 ou MPEG-2), áudio (AAC, MP1, MP2, MP3, CELP, HVXC, dentre outros), legendas, metadados e outros, é necessário buscar no arquivo a região contendo parâmetros gerais de configuração do decodificador de áudio como a taxa de amostragem e em seguida a região onde estão armazenados os *raw data blocks* do áudio AAC.

Ambos os decodificadores rodam no processador e são chamados em tempo de execução. A cada solicitação de um novo *byte* pelo decodificador, um dos decodificadores é acionado e o mesmo lê a fonte de entrada para extrair o elemento *payload* com largura de 1 *byte* a ser tratado pelo decodificador AAC.

Capítulo 4

Resultados

A solução final está implementada no FPGA Cyclone II (EP2C70F896C6) da Altera incluído no kit de desenvolvimento DE2-70 do fabricante TerAsic. Entre os recursos disponíveis na placa além do FPGA de aproximadamente 70 mil elementos lógicos, estamos utilizando o módulo de memória SSRAM de 2MB como memória do processador, um módulo de memória Flash de 8MB para armazenar o *bitstream* a ser decodificado e um tocador de áudio, além de botões, *switches* e LEDs para comando e controle.

Esta solução integra todos os módulos desde o Decodificador de Entropia até o Tocador de Áudio. Na arquitetura temos em software os decodificadores LATM/LOAS, MP4 e o *Bitstream Deformater* (AAC *parser*) sendo executados no processador Nios II e o restante dos módulos todos implementados em hardware. Apesar da natureza sequencial da decodificação, três etapas podem atuar em paralelo. A primeira é a sequência de *parsing* (LATM/LOAS ou MP4 seguido do *parser* AAC) seguida das ferramentas espectrais. A segunda etapa paralela é o Banco de Filtros Inverso que, após efetuar a etapa de pré-processamento da IMDCT, libera um dos canais de áudio para que o primeiro canal do próximo *raw data block* possa ser decodificado e armazenado na memória de Coeficientes Espectrais. A terceira e última etapa paralela é a do tocador de áudio que possui duas áreas de memória, uma para consumo (tocar) e outra para guardar as amostras do próximo bloco que está sendo decodificado.

A arquitetura final é ilustrada na Figura 4.1 onde podemos observar os módulos implementados em software representados pelos blocos brancos dentro do processador Nios II, seguido do barramento Avalon que conecta o processador às memórias SSRAM e FLASH. Os demais módulos em hardware são representados pelos blocos em cinza e estão separados em 7 blocos, 6 estão ligados ao processador via barramento e 1 (Ferramentas espectrais) fica independente e é acionado pelo decodificador de entropia e desligado pelo início do Banco de Filtros Inverso.

Após a integração final, como esperado, ouve significativa redução no tamanho do software sendo executado no processador cujo tamanho do binário caiu de 502 kB para 32 kB. A descrição da utilização do hardware do FPGA é apresentada na Tabela 4.1 onde podemos ver o uso de 21527 Elementos Lógicos no total, além de 121 elementos DSP de 9x9 bits e 679040 bits de memória.

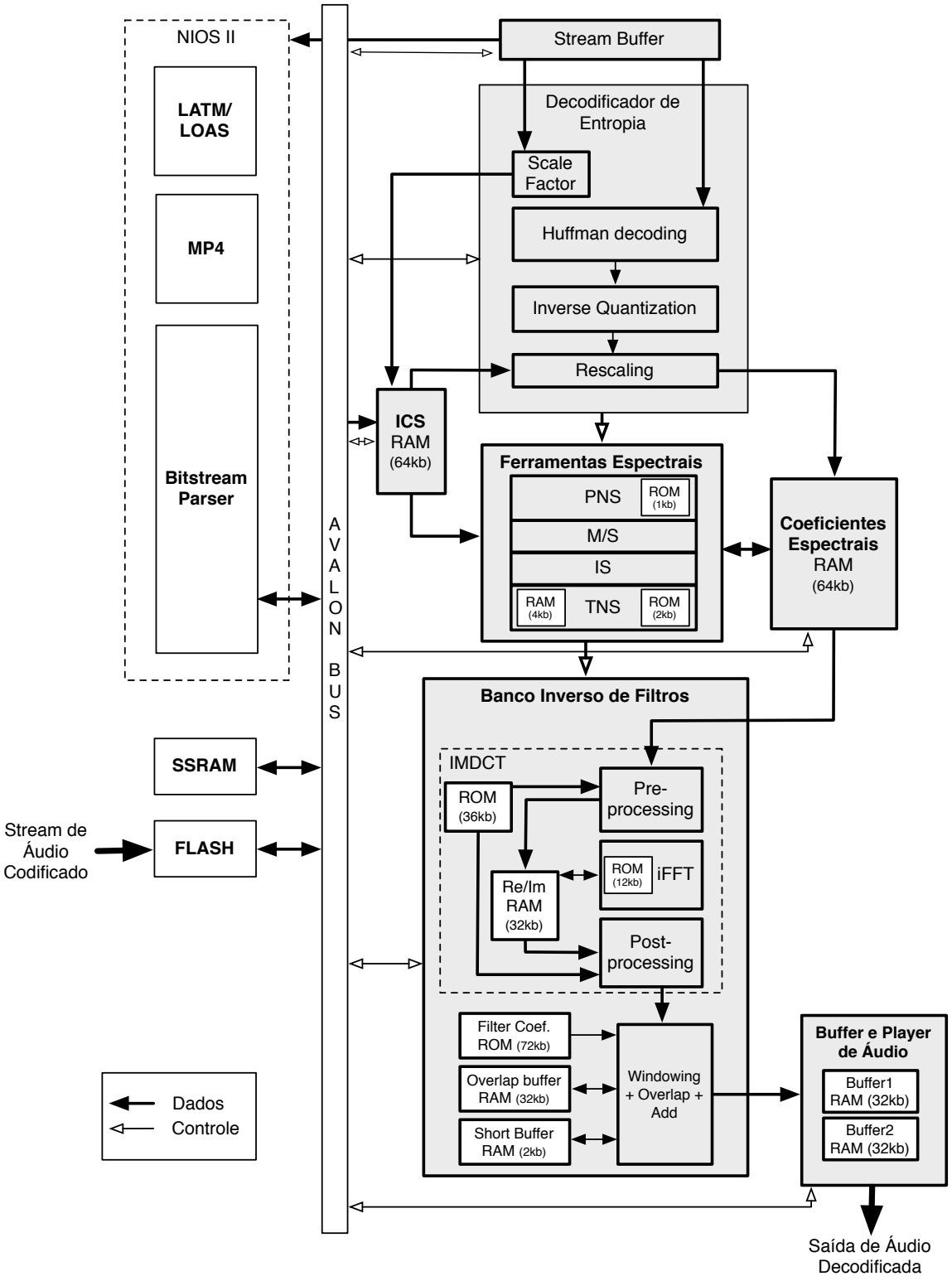

Figura 4.1: Arquitetura final da solução de Coprojeto HW/SW do Decodificador AAC-LC

Tabela 4.1: Utilização de Hardware da Solução Final

Módulo	**Elementos Lógicos**	**Memória (bits)**	**Multiplicadores Dedicados (9x9-bit)**
CPU + Barramento	2742	157184	4
Stream Buffer	510	-	-
Decodificador de Entropia (DFE, DCE, Quantização, Re-escalonador)	6393	35520	15
Memória Compartilhada (ICS, Coef Spec)	136	131072	
IS	1007	-	14
MS	270	-	-
TNS	1099	6080	16
PNS	2584	1024	40
Banco de Filtros Inverso	2956	265216	32
Buffer de Saída de Áudio	97	65536	-
Áudio Player	497	16384	-
Outros (Controladores de memória externa, UART, JTAG, etc.)	3236	1024	-
Total	**21527**	**679040**	**121**

4.1 Performance

Em termos de performance, a Figura 4.2 e a Tabela 4.2 apresentam a comparação entre as três versões do decodificador: exclusivamente em software, solução intermediária com poucos módulos em hardware e o desempenho da solução final. Nesta tabela é possível perceber que todas as funções que migraram para hardware tiveram um aumento significativo de performance. Além disso, percebe-se que a integração do *Stream Buffer* juntamente com a remoção da função de tocar o áudio a partir do processador reduziu praticamente pela metade o tempo necessário para as tarefas executadas em software.

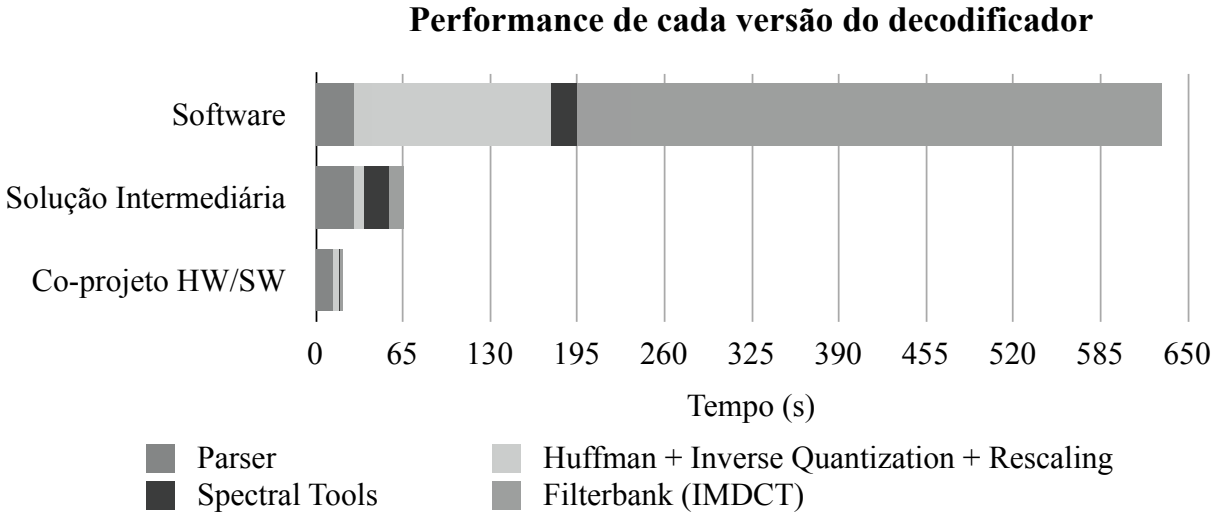

Figura 4.2: Gráfico ilustrando as performances das 3 versões do decodificador

Tabela 4.2: Performance de cada Etapa do Decodificador em Software

Módulos	**Solução por Software**[1] **Tempo (s)**	**Solução Intermediária**[2] **Tempo (s)**	**Solução Final**[3] **Tempo (s)**
Parser	28,00	28,00	12,44
Decodificação de Entropia (Huffman + Quantização Inversa + Re-escalonamento)	146,87	7,08	4,08
Ferramentas Espectrais[4]	19,02	19,02	1,26
IS	4,64	4,64	0,14
MS	4,12	4,12	0,21
TNS	10,12	10,12	0,87
PNS	0,16	0,16	0,04
Banco de Filtros	436,31	10,79	1,79
Total	**630,19**	**64,89**	**19,57**

[1]Processador Nios II a 100 MHz
[2]Processador Nios II a 100 MHz e Hardware a 50 MHz
[3]Processador Nios II a 50 MHz e Hardware a 50 MHz
[4]A linha de "Ferramentas Espectrais" representa a soma de IS, MS, TNS e PNS.

Os testes finais de performance foram executados incluindo os arquivos estéreo originais codificados a 256 kpbs juntamente com 5 outros arquivos que possuem 6 canais de áudio e estão codificados a 396 kbps. Em todos os casos temos a taxa de amostragem de 48 kHz. Os resultados podem ser observados na Tabela 4.3

Tabela 4.3: Performance final para áudio de 2 e 6 canais

Tempo de Decodificação	**2 canais (256 kbps)**	**Razão para Tempo-Real**	**6 canais (396 kbps)**	**Razão para Tempo-Real**
Total (Áudio de 60s)	14,96s	24,9%	31,97s	53,3%
Média por Bloco	5,19ms	24,9%	11,10ms	53,3%
Pico por Bloco	8,56ms	41,1%	16,9ms	81,1%

4.2 Qualidade

Em termos de qualidade do som, a análise foi realizada a partir do cálculo do *Signal-to-Noise Ratio* (SNR) conforme apresentado da seção 3.1. Os testes de qualidade foram realizados ao longo de todas as etapas de implementação dos módulos em hardware. Apresentamos nas Figuras 4.3, 4.4 os gráficos com a comparação entre a saída final tanto de um trecho com janela longa quanto de um trecho com janelas curtas. No caso são plotadas as amostras de saída do decodificador em software versus as amostras do decodificador em hardware.

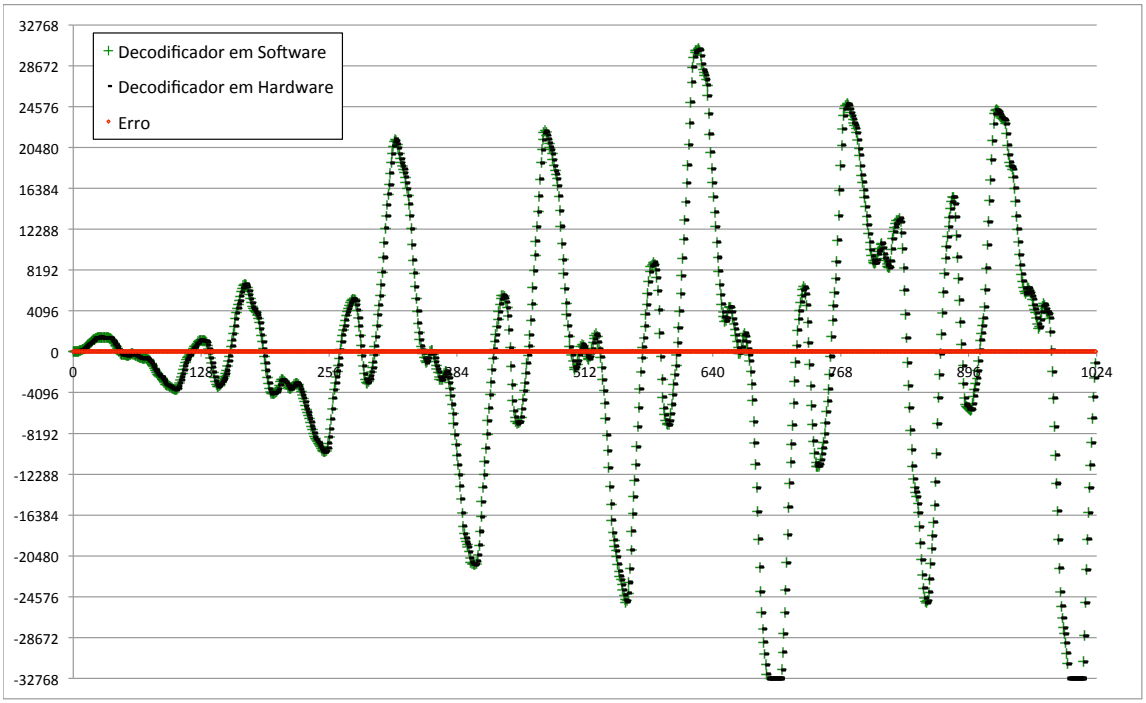

Figura 4.3: Gráfico ilustrando a comparação da saída de software vs hardware para janela *Only Long Sequence*

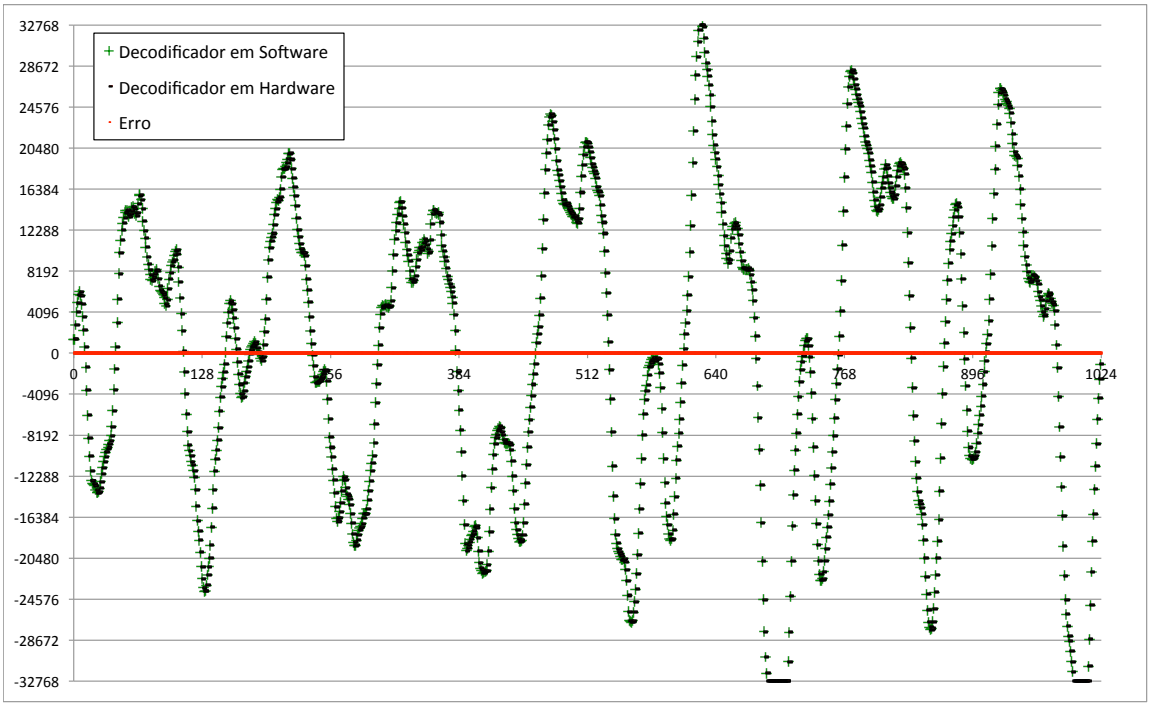

Figura 4.4: Gráfico ilustrando a comparação da saída de software vs hardware para janela *Eight Short Sequence*

Em testes mais exaustivos conduzidos por Daniel Chaves Café a partir de 153 arquivos de áudio testando a versão do decodificador em software contra o decodificador de referên-

cia da ISO, obteve-se uma variação de qualidade em SNR entre 65 dB e 81 dB conforme a distribuição apresentada na Tabela 4.4. Podemos ver que um pequeno percentual de amostras ficou entre 65 a 70 dB. A grande maioria, 75,8% ficou com qualidade acima de 75dB.

Tabela 4.4: Resultados de testes SNR em software

SNR	Percentual de Amostras
65 a 70dB	2,6%
70 a 75 dB	21,6%
Acima de 75dB	75,8%

Devido à limitação da memória do kit FPGA, a mesma quantidade de testes não pode ser realizada em tempo hábil com a solução final em hardware. Ao invés disso, foram selecionadas 5 arquivos de áudio e o resultados foram semelhantes aos obtidos em software. Apesar destes serem testes objetivos, os resultados acima de 65dB indicam ser bastante provável que os testes subjetivos mais apurados obtenham sucesso.

4.3 Consumo de Energia

Após obter a arquitetura final do sistema no FPGA, foi realizada uma estimativa aproximada do consumo de potência através da ferramenta PowerPlay Power Analyser dentro do ambiente do Quartus II da Altera. Neste caso, foi realizada uma análise padrão, sem dados de simulação, que estima o consumo total do sistema em que todos os módulos são considerados ativos durante todo o tempo. Deste resultado total, foi calculada uma média ponderada de acordo com o tempo em que cada módulo do sistema fica ativo durante o processo de decodificação obtendo-se um resultado aproximado do consumo real da arquitetura para o caso de decodificação de 6 canais de áudio. Os resultados de consumo de potência são apreentado na Tabela 4.5.

Tabela 4.5: Resultados do consumo de energia

Tipo de Consumo	Consumo (mW)
Potência Términa Total	457,80
Potência Térmica Dinâmica	297,18
Potência Térmica Estática	65,09
Média Ponderada da Potência Total	271,00

Observa-se que o FPGA Cyclone II utiliza tecnologia de fabricação de 90-nm e que o mesmo FPGA de uma geração mais atual, o Cyclone V, é fabricado em tecnologia de 28-nm sendo pelo menos 60% mais eficiente em termos de consumo de energia.

4.4 Comparações

Conforme visto na revisão de literatura, várias abordagens são possíveis para implementar o decodificador em hardware. Desde as soluções puramente em software até as solução completas em hardware dedicado.

Pela Tabela 4.6 podemos observar em primeiro lugar que além da solução proposta, somente a primeira solução apresentada que utiliza o processador RISC implementa a decodificação de 6 canais em tempo real. Todas as demais soluções se aplicam somente a dois canais. Além disso, nenhuma das soluções encontradas implementa o AAC na versão descrita pelo MPEG-4 onde está presente a ferramenta de processamento espectral PNS bem como não há menção em nenhum dos casos da decodificação do protocolo LATM/LOAS ou do MP4 em sua tarefa inicial de *parsing*.

As soluções puramente em hardware se destacam tanto pelo baixo consumo de energia quanto pela baixa frequência de *clock* necessária para decodificar áudios de dois canais.

Podemos ainda destacar a dificuldade em comparar os resultados por serem todas as soluções implementadas em plataformas bastante distintas. No caso das soluções que utilizam a técnica de coprojeto, cada uma adota um processador com características distintas em termos de capacidade de processamento (destacado em MIPS) e tamanho de memória *cache* tanto de dados quanto de instrução diferentes. No caso das implementações em hardware dedicado, não é possível fazer uma correlação direta entre a quantidade de elementos lógicos dos FPGAs e das portas lógicas em VLSI. Mesmo entre as soluções em FPGA, cada uma utiliza um fabricante e família distintos dificultando a relação direta.

A solução mais próxima em termos de plataforma é a de Renner (27) que, apesar de não utilizar a abordagem de coprojeto de hardware e software, implementa a solução em um mesmo FPGA. Porém, mesmo não possuindo as ferramentas de processamento espectral ou os decodificadores LATM/LOAS e MP4, utiliza uma área 23,3% maior e 64% menos memória.

Tabela 4.6: Comparação da Solução proposta com as arquiteturas encontradas na literatura

Referência	Takamizawa et al. (30)	Tsai, Liu e Wang (34)	Tsai e Liu (32)	Renner (27)	Liu and Tsai (23)	Zhou et al. (37)	Tao et. al. (31)	Solução Proposta
Arquitetura da Solução	Software + 2 Áreas de memória RAM	Hardware-Puro	Hardware-Puro	Hardware-Puro	Coprojeto HW/SW	Coprojeto HW/SW	Coprojeto HW/SW	Coprojeto HW/SW
Decodificador Implementado	MPEG-2 AAC-LC	MPEG-2 AAC-LC	MPEG-2 AAC-LC	MPEG-2 AAC-LC	MPEG-2 AAC-LC	MPEG-2 AAC-LC / MP3	MPEG-2 AAC-LC / MP3	MPEG-2/4 AAC-LC
Tecnologia	Processador de Propósito Geral	ASIC (TSMC 0,25μm)	ASIC (UMC 0,18μm)	FPGA (Cyclone II Altera)	VLSI / FPGA (Xilinx VertexE)	VLSI	FPGA (Xilinx XUP V2P	FPGA (Cyclone II Altera)
Processador	RISC NEC V380	-	-	-	ARM (ARM920T)	OpenRisc 1200	PowerPC	Nios II/f
Desempenho do Processador	158 MIPS	-	-	-	60 a 200 MIPS	250 MIPS @ 250MHz	N/A	50 MIPS
Freq. do Processador	133 MHz	-	-	-	41 MHz	10,6 MHz	300 MHz	50 MHz
Freq. do Hardware	-	3,3 MHz	1,3 MHz	4 MHz	22,8 MHz	10,6 MHz	N/A	50 MHz
N. de Canais	6	2	2	2	2	2	2	6
Tx. Amost. Áudio	48 kHz	44,1 kHz	44,1 kHz	48 kHz	44,1 kHz	44,1 kHz	48 kHz	48 kHz
Bitrate	432 kbps	128 kbps*	128 kbps*	128 kbps*	128 kbps*	128 kbps*	118 kbps	432 kbps
Power	300 mW	158 mW	2,45 mW	N/A	N/A	N/A	N/A	270 mW
LE (FPGA) / LG (ASIC)		82,2 k LG	102 k LG	26549 LE (248704 bits de memória)	6266 Slices	44,3 k LG + 16 k RAM + 4 k ROM	N/A	21527 LE (679040 Mem.)

Capítulo 5

Conclusões e Trabalhos Futuros

O texto apresentado procura oferecer ao leitor uma visão resumida sobre os processos envolvidos na codificação de áudio e mais específica sobre o padrão AAC. Apresenta-se também a abordagem de coprojeto juntamente com a tecnologia dos FPGAs, escolhida para implementar a solução. Em seguida, o processo de implementação é descrito apresentando como foi realizada a escolha dos módulos a serem desenvolvidos em hardware bem como os detalhes da implementação dos mesmos.

Os diversos livros, artigos e normas estudados foram de fundamental importância na compreensão teórica sobre a codificação de áudio. As normas que especificam o padrão MPEG-4 AAC são esclarecedoras no que se refere ao algoritmo, porém, o mesmo só é detalhado em linguagem de alto nível e portanto foi necessário os estudo de artigos que exploram arquiteturas de hardware específicas para cada tipo de operação computacional.

A implementação de uma solução de tamanha complexidade só foi possível através do trabalho modular onde cada etapa do decodificador foi isolada, implementada e testada de maneira separada até que estivesse funcionando de maneira adequada. A escolha do processador Nios II como plataforma de desenvolvimento ainda nos estágios iniciais do trabalho foi de grande valia por facilitar a implementação de *test benches* que integram a solução de referência completa em software com os módulos sendo desenvolvidos individualmente em hardware. Apesar da modularização facilitar o desenvolvimento de cada etapa, as necessidades de otimização de tempo e espaço na utilização dos recursos de hardware tornam a etapa de integração um desafio para que os módulos possam funcionar em sincronia e compartilhando recursos.

Existem ainda desafios específicos relacionados às ferramentas de software utilizadas para a implementação. Como ambiente de simulação, foi utilizado o QuestaSim 6.6, para síntese em FPGA foi utilizado o Quartus II v11 e para a programação do processador Nios II foram utilizados o ambiente de desenvolvimento de sistemas *SOPC Builder* juntamente com o ambiente de desenvolvimento de software Nios II IDE. Apesar da grande quantidade de recursos e sofisticação destas ferramentas, a quantidade de erros de compilação e documentação falha sobre os mesmos torna a curva de aprendizado deste ambiente um desafio em si.

Como resultado, temos uma solução de coprojeto de hardware e software que implementa um decodificador MPEG-4 AAC-LC. Os sistema funciona na placa DE2-70 que contém um FPGA Cyclone II do fabricante Altera e é capaz de reproduzir em tempo real áudio estéreo de até 256 kbps em tempo real funcionando a 18,3 MHz. Áudios de até 6

canais também podem ser decodificados a uma frequência de 50 MHz, porém não podem ser reproduzidos devido à limitação do reprodutor de áudio da placa que só é capaz de reproduzir 2 canais ao mesmo tempo.

A solução implementa as etapas de *parsing* tanto dos protocolos LATM/LOAS quanto do envelope MP4 e do AAC-LC em software. Os demais módulos de processamento, especificamente, o Decodificador de Entropia com a Quantização Inversa e Reescalonador, o *Stream Buffer*, as ferramentas de processamento espectral *Intensity Stereo*, *Mid/Side Stereo*, *Temporal Noise Shaping* e *Perceptual Noise Substitution*, o Banco de Filtros Inverso e o tocador de Áudio com seu *buffer* de memória são todos implementados em hardware.

Quando comparada às soluções encontradas na literatura, a proposta aqui apresentada se destaca por implementar o padrão MPEG-4 e atender aos requisitos de decodificação em tempo real para até 6 canais.

As diversas abordagens de implementação de uma solução de áudio encontradas na literatura demonstram que esta é uma solução complexa onde não há uma única ou a melhor abordagem. Para sistemas com poucas restrições no consumo de energia, as soluções em software puro tendem a ser mais adequadas pois podem ser rapidamente implementadas e expandidas para novos padrões. Além disso, processadores de propósito geral de baixo consumo de energia estão tornando cada vez mais viáveis sua escolha quando há requisitos moderados de processamento em tempo real. Por outro lado, as exigências de algoritmos cada vez mais complexos em dispositivos móveis com múltiplas funções tornam a pequisa na área de circuitos dedicados em hardware cada vez mais necessária.

Apesar da grande vantagem em termos de consumo de energia, soluções puramente em hardware são muito caras e levam muito tempo para serem desenvolvidas, o que as torna menos viáveis para aplicações em menor escala ou que necessitem ir para o mercado em um curto espaço de tempo. Ferramentas de Síntese de Alto Nível que traduzem algoritmos sequenciais em software para hardware de maneira automática como as que experimentamos no caso da FFT exigem conhecimento especializado e, apesar de gerarem bons resultados, ainda não apresentam o mesmo nível de otimização de uma solução manual.

A abordagem de coprojeto utilizada neste trabalho apresenta grande potencial pelo fato de se pode dosar o percentual entre hardware dedicado e solução por software até que os requisitos de desempenho desejados sejam alcançados. Esta abordagem tem ganhado força com os recentes lançamentos de fabricantes de FPGA onde processadores de propósito geral em silício com razoável poder de processamento e circuitos reconfiguráveis (FPGA) já estão sendo comercializados em SoCs (*System-on-a-Chip*).

Ressaltamos ainda que este trabalho gerou dois artigos de autoria conjunta com meu orientador, Ricardo Jacobi e co-orientador Pedro de Azevedo Berger. O primeiro artigo, entitulado "*Hardware and software co-design for the AAC audio decoder*", foi apresentado no *25th Symposium on Integrated Circuits and Systems Design* (SBCCI 2012) e o segundo artigo, entitulado "High-level design and synthesis of a MPEG-4 AAC IMDCT module"apreentado no 2013 IEEE *Fourth Latin American Symposium on Circuits and Systems* (LASCAS) estando ambos disponíveis para consulta na biblioteca digital do IEEExplore (http://ieeexplore.ieee.org).

Diversas melhorias e incrementos a esta solução são possíveis e como trabalhos futuros sugerimos:

- A implementação da ferramenta SBR - *Spectral Band Replication* para atender aos requisitos do *High Efficiency AAC Profile v1*;
- A implementação da ferramenta PS - *Parametric Stereo* para atender aos requisitos do *High Efficiency AAC Profile v2*;
- O compartilhamento dos multiplicadores e somadores entre as ferramentas de processamento espectral;
- O compartilhamento dos multiplicadores entre as ferramentas espectrais, a quantização inversa e o re-escalonador;
- A redução do número de bits dos fatores das tabelas de *twiddles* na IMDCT e dos fatores do janelamento do Banco de Filtros com o intuito de reduzir a área de memória sem comprometer a qualidade do áudio de saída;
- A implementação das etapas de *parser* dedicadas em hardware para serem comparadas com a solução de coprojeto;
- Testes mais elaborados tanto objetivos quanto subjetivos a fim de avaliar de maneira mais criteriosa a qualidade de áudio da solução.

Referências

[1] Televisão digital terrestre: Codificação de vídeo, áudio e multiplexação - Parte 2: Codificação de áudio, 2008. ix, 2, 12, 13, 19, 28, 32, 45

[2] Altera Corporation, San Jose, CA. *Avalon Bus Specification Reference Manual*, Julho 2003. 28, 38

[3] Altera Corporation, San Jose, CA. *Cyclone II Device Handbook*, 2008. 28

[4] Altera Corporation, San Jose, CA. *Nios II Processor Reference Handbook*, Maio 2011. 28, 38

[5] Altera Corporation, San Jose, CA. *Quartus II Handbook Version 11.0*, Maio 2011. 28

[6] Altera Corporation, San Jose, CA. *The Breakthrough Advantage for FPGAs with Tri-Gate Technology*, 2013. 28

[7] C. Bauer and M. Vinton. Joint optimization of scale factors and huffman code books for mpeg-4 aac. *Signal Processing, IEEE Transactions on*, 54(1):177–189, 2006. 29

[8] C.-H. Chen, B.-D. Liu, and J.-F. Yang. Recursive architectures for realizing modified discrete cosine transform and its inverse. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 50(1):38–45, 2003. 29, 46

[9] R. L. DRAKE, A. W. VOGL, and A. MITCHELL. *Gray's Anatomia para Estudantes.* Elsevier Editora, 2a edição edition, 2010. vii, 6

[10] F. Du, G. Du, Y. Song, D. Zhang, and M. Gao. An implementation of filterbank for mpeg-2 aac on fpga. In *Anti-counterfeiting, Security and Identification, 2008. ASID 2008. 2nd International Conference on*, pages 391–394, 2008. 29, 46

[11] P. Duhamel, Y. Mahieux, and J.-P. Petit. A fast algorithm for the implementation of filter banks based on 'time domain aliasing cancellation'. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pages 2209–2212 vol.3, 1991. 46

[12] M. Fingeroff. *High Level Synthesis Blue Book.* Mentor Graphics Corporation, 2010. 50

[13] R. Gluth. Regular fft-related transform kernels for dct/dst-based polyphase filter banks. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pages 2205–2208 vol.3, 1991. 46

[14] W. M. Hartmann. *Signals, Sound, and Sensation*. Springer Science+Business Media, Inc., 1998. vii, 8

[15] H. J. Hee, M. H. Sunwoo, and J. H. Moon. Novel non-linear inverse quantization algorithm and its architecture for digital audio codecs. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 357–360, 2007. 40

[16] J. Herre. Tns, quantization and coding methods temporal noise shaping, quantization and coding methods in perceptual audio coding: A tutorial introduction. 20, 29

[17] J. Herre. From joint stereo to spatial audio coding - recent progress and standartization. In *Proc. of the 7th Int. Conference on Digital Audio Effects (DAFx'X04)*, 2004. 29

[18] Information technology — Generic coding of moving pictures and associated audio information — Part 7: Advanced Audio Coding (AAC), 2006. 2, 28

[19] Information Technology - Coding of audiovisual objects - Part 14: MP4 File Format, 2003. 36

[20] Information Technology - Coding of audiovisual objects - Part 3: Audio, 2009. vii, 2, 12, 14, 28, 32

[21] S.-C. Lai, S. F. Lei, and C.-H. Luo. Common architecture design of novel recursive mdct and imdct algorithms for application to aac, aac in drm, and mp3 codecs. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 56(10):793–797, 2009. 46

[22] L. Li, H. Miao, X. Li, and D. Guo. Efficient architectures of mdct/imdct implementation for mpeg audio codec. In *Anti-counterfeiting, Security, and Identification in Communication, 2009. ASID 2009. 3rd International Conference on*, pages 156–159, 2009. 29, 46

[23] C.-N. Liu and T.-H. Tsai. Soc platform based design of mpeg-2/4 aac audio decoder. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 2851–2854 Vol. 3, 2005. 30, 31, 87

[24] N. Murthy and M. N. S. Swamy. A parallel/pipelined algorithm for the computation of mdct and imdct. In *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*, volume 4, pages IV–540–IV–543 vol.4, 2003. 46

[25] V. Nikolajevic and G. Fettweis. Computation of forward and inverse mdct using clenshaw's recurrence formula. *Signal Processing, IEEE Transactions on*, 51(5):1439–1444, 2003. 29, 46

[26] T. Painter and A. Spanias. Perceptual coding of digital audio. *Proceedings of the IEEE*, 88(4):451–515, 2000. 28

[27] A. Renner. Arquitetura de um decodificador de Áudio para o sistema brasileiro de televisão digital e sua implementação em fpga. Mestrado, Universidade Federal do Rio Grande do Sul, 2011. 30, 31, 38, 86, 87

[28] P. sheng Wu and Y.-T. Hwan. Efficient imdct core designs for audio signal processing. In *Signal Processing Systems, 2003. SIPS 2003. IEEE Workshop on*, pages 275–280, 2003. 29

[29] A. Spanias, T. Painter, and V. Atti. *Audio Signal Processing and Coding.* John Wiley & Sons, Inc., Hoboken, New Jersey, 2007. vii, 7, 8, 10, 11, 20, 21, 22, 28

[30] Y. Takamizawa, K. Nadehara, M. Boegli, M. Ikekawa, and I. Kuroda. Mpeg-2 aac 5.1-channel decoder software for a low-power embedded risc microprocessor. In *Signal Processing Systems, 1999. SiPS 99. 1999 IEEE Workshop on*, pages 351–360, 1999. 29, 31, 87

[31] Z. Tao, G. Buning, Q. Haojun, and Y. Fengping. Mp3 / aac audio decoder implementation based on hardware and software co-design. In *Image and Signal Processing (CISP), 2010 3rd International Congress on*, volume 8, pages 3695–3698, 2010. 30, 31, 87

[32] T.-H. Tsai and C.-N. Liu. Low-power system design for mpeg-2/4 aac audio decoder using pure asic approach. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 56(1):144–155, 2009. 30, 31, 87

[33] T.-H. Tsai, C.-N. Liu, H.-Y. Lin, H.-C. Liu, and C.-Y. Wu. A 1.4 mhz 0.21 mw mpeg-2/4 aac single chip decoder. In *Custom Integrated Circuits Conference, 2009. CICC '09. IEEE*, pages 653–656, 2009. 40

[34] T.-H. Tsai, C.-N. Liu, and Y.-W. Wang. A pure-asic design approach for mpeg-2 aac audio decoder. In *Information, Communications and Signal Processing, 2003 and Fourth Pacific Rim Conference on Multimedia. Proceedings of the 2003 Joint Conference of the Fourth International Conference on*, volume 3, pages 1633–1636 vol.3, 2003. 29, 31, 87

[35] T.-H. Tsai, H.-C. Liu, W.-C. Chang, and C.-N. Liu. A platform-based soc design for a multi-standard audio decoder. In *Communications, Circuits and Systems, 2008. ICCCAS 2008. International Conference on*, pages 758–761, 2008. 30

[36] Xilinx, Inc., San Jose, CA. *Virtex-7 FPGAs*, 2012. 28

[37] D. Zhou, P. Liu, J. Kong, Y. Zhang, B. He, and N. Deng. An soc based hw/sw co-design architecture for multi-standard audio decoding. In *Solid-State Circuits Conference, 2007. ASSCC '07. IEEE Asian*, pages 200–203, 2007. 30, 31, 87

[38] U. Zölzer. *Digital Audio Signal Processing.* John Wiley & Sons, Inc., Hoboken, New Jersey, 2nd edition edition, 2008. vii, 24, 28