UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE TELEINFORMÁTICA
GRADUAÇÃO EM ENGENHARIA DE TELEINFORMÁTICA

TIAGO GOMES CASTRO

UM CORE DE PROCESSAMENTO DIGITAL DE SINAIS MONOTÔNICOS PARA TELEFONIA

FORTALEZA
2016

TIAGO GOMES CASTRO

UM CORE DE PROCESSAMENTO DIGITAL DE SINAIS MONOTÔNICOS PARA TELEFONIA

Projeto de conclusão de curso apresentado ao curso de Graduação em Engenharia de Teleinformática do Departamento de Engenharia de Teleinformática da Universidade Federal do Ceará, como requisito parcial para obtenção do diploma de Engenheiro de Teleinformática.

Orientador: Professor Dr. Jarbas Aryel Nunes da Silveira

FORTALEZA
2016

Tiago Gomes Castro Um core de processamento digital de sinais monotônicos para telefonia/ Tiago Gomes Castro. – Fortaleza, 2016- 57 p. : il.; 30 cm.
Orientador:Professor Dr. Jarbas Aryel Nunes da Silveira
Coorientador:Eng. Me. Fábio Cisne Ribeiro
Projeto de conclusão de curso apresentado ao curso de Graduação em Engenharia de Teleinformática do Departamento de Engenharia de Teleinformática da Universidade Federal do Ceará, como requisito parcial para obtenção do diploma de Engenheiro de Teleinformática. – Universidade Federal do Ceará
Centro de Tecnologia
Departamento de Engenharia de Teleinformática
Graduação em Engenharia de Teleinformática, 2016.
1. Goertzel. 2. FPGA. I. Professor Dr. Jarbas Aryel Nunes da Silveira. II. Universidade Federal do Ceará. III. Centro e Tecnologia. IV. Um core de processamento digital de sinais monotônicos para telefonia

TIAGO GOMES CASTRO

UM CORE DE PROCESSAMENTO DIGITAL DE SINAIS MONOTÔNICOS PARA TELEFONIA

Projeto de conclusão de curso apresentado ao curso de Graduação em Engenharia de Teleinformática do Departamento de Engenharia de Teleinformática da Universidade Federal do Ceará, como requisito parcial para obtenção do diploma de Engenheiro de Teleinformática.

Trabalho aprovado em ___/___/_____.

BANCA EXAMINADORA

Professor Dr. Jarbas Aryel Nunes da Silveira(Orientador)
Universidade Federal do Ceará(UFC)

Eng. Me. Fábio Cisne Ribeiro(Coorientador)
Universidade Federal do Ceará(UFC)

Prof. D.r Paulo Cesar Cortez
Universidade Federal do Ceará(UFC)

Prof. M.e Ricardo Jardel Nunes Da Silveira
Universidade Federal do Ceará(UFC)

Dedico este trabalho à minha família. Em especial Natalia, cúmplice em todos os momentos.

AGRADECIMENTOS

Agradeço a todos os amigos que fiz nesta empreitada, em especial à Ádria, Alan Cadore, Filipe Lins, George Harinson, João Marcelo, Luciene Lira, Marciel, Ridley e Victor Fernandes, com os quais compartilhei muitos momentos bons, e alguns tensos também.

Agradecimentos aos amigos Cincinato, Fábio Cisne, Jacques Bessa, José Adriano Filho e Nícolas Araujo, com os quais tive um grande aprendizado em meu primeiro projeto de mercado, no Lesc.

Agradeço aos meus professores, fontes de conhecimento infinito, capaz de ensinar não só os conteúdos técnicos, mas verdadeiras lições de vida também.

Um agradecimento especial ao professor Jarbas, o qual sempre acreditou em mim. Com seus ensinamentos, suas histórias e seus depoimentos, que nos mostra que apesar de difícil, a engenharia tem uma beleza sensível a aqueles que se empolgam com seus desafios.

Agradeço ao Lesc, laboratório que entre idas e vindas, estou desde 2010. Onde conheci pessoas acolhedoras, divertidas e acima de tudo, ultra responsáveis com os projetos. Sinto orgulho de ser um pequena parte da história deste laboratório.

Agradeço à minha família, em especial aos meus pais, Amorim e Josiêda, e minha tia Rosa Maria por todo apoio, ensinamento, cobranças e carinhos.

Um agradecimento mais que especial à Natalia Maria, minha companheira de todos os momentos. Mulher de índole e temperamento forte, mas com um coração enorme e caridoso. Obrigado por todo o apoio, paciência, cobrança e sobretudo amor.

RESUMO

Com a tecnologia tendo grandes avanços nas últimas décadas, a taxa de renovação de dispositivos eletrônicos aumenta também, fazendo com que alguns dispositivos tenham sua fabricação descontinuada. Este trabalho tem como um de seus propósitos evitar um novo reprojeto das funções de processamento de sinalização monotônica, pois neste trabalho foi implementado um core de processamento descrito em HDL, sendo portável para diferentes FPGAs, pois a linguagem de descrição de hardware é universal. O core descrito neste trabalho é um módulo controlado, a ser inserido em um sistema, que tem como propósito ser o bloco responsável pelo processamento matemático dos sinais monotônicos do sistema. No desenvolvimento deste trabalho foi aplicado o algoritmo de Goertzel, que é derivado da transformada discreta de Fourier. Os resultados dos tempos de processamentos mostram que o core proposto neste trabalho tem uma bom desempenho, alcançando um processamento máximo de 222 canais, tendo como meta inicial 8, em um intervalo de tempo de $125\mu s$, denotando um alto poder de processamento para um valor de clock de apenas $80MHz$.

Palavras-chave: Goertzel. FPGA. DSP. VHDL.

ABSTRACT

With technology taking great strides in recent decades, the electronic devices refresh rate also increases, causing some devices having their manufacture discontinued. This work has as one of its purposes to avoid a new redesign of monotonic signaling processing functions, because in this work was implemented a processing core described in HDL, and it is portable to different FPGAs, without the need to make changes to the code. The core described in this paper, is a controlled module, to be inserted in a system, which aims to be responsible for mathematical processing of the monotonic signals of system. In developing this work, a processing tool was applied, the Goertzel algorithm, which is a derivative of Discrete Fourier Transform. The results of the processing times have shown the that work proposed has a good performance, reaching a maximum throughput of 222 channels, with the initial target 8, on a $125\mu s$ time interval, denoting a high processing power for a clock value of only $80MHz$

Keywords: Goertzel. FPGA. DSP. VHDL.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Devido ao constante crescimento da tecnologia e sua participação cada vez mais presente no cotidiano, os sistemas digitais de computação vêm se tornando ferramenta de grande utilidade para o desenvolvimento da sociedade. Sua participação permeia uma vasta gama de atividades, como quais, medicina, agricultura, controle aéreo, etc, fazendo com que seus avanços causem impactos significativos nos processos aos quais fazem parte.

Os sistemas de telecomunicações também tiram proveito desta evolução tecnológica, tendo na década de 60 iniciado um ciclo de digitalização de seus sistemas, que estende-se até hoje, que tem como características principais a compactação do hardware e o aumento do uso de softwares, o que tem como resultado equipamentos cada vez menores e com maior poder de processamento. No princípio, a digitalização teve maior ênfase nos sistemas de transmissão, fazendo uso da técnica de digitalização de sinais denominada PCM (Pulse Code Modulation). Não demorou muito e a digitalização foi também adotada nas centrais telefônicas. Mesmo com os avanços tecnológicos, na telefonia ainda hoje há uma coexistência de centrais telefônicas analógicas e digitais, com um decréscimo progressivo das centrais analógicas e incremento das centrais digitais.

Para a correta comunicação das centrais telefônicas, seja com outras centrais ou com usuários, são usados sinalizações acústicas, que são sinais audíveis na faixa de frequências $[0Hz, 4000Hz]$. Alguns exemplos destas sinalizações são o tom de discagem, que consiste em um sinal contínuo de $425Hz$, outro exemplo é o tom de ocupado, que também é sinal de $425Hz$ com duração de $250ms$ e interrupção de $250ms$ sucessivamente.

Para o desenvolvimento deste trabalho, foram necessários conhecimentos em desenvolvimento de hardware em VHDL, para isto, foram utilizados (PEDRONI, 2010) e (COSTA; MESQUITA; PINHEIRO, 2011), além disso, para estudo da ferramenta de desenvolvimento, foram utilizados (XILINX, 2012d), (XILINX, 2012c) (XILINX, 2012b) e (XILINX, 2012a).

1.1 Motivação

De maneira geral, as centrais telefônicas fazem o processamentos dos áudios com uso de processadores de sinais, os quais são embarcados com firmware para devido processamento. Com as atualizações das tecnologias, alguns modelos de processadores de sinais utilizados nas centrais podem ter sua fabricação descontinuada, fazendo com que um novo firmware para outro modelo de DSP tenha que ser projetado, demandando investimento financeiro e de pessoal, tendo o fato de que o código do firmware é totalmente dependente da plataforma alvo.

Uma alternativa para este problema está no uso de FPGAs(*Field Programmable*

Gate Array), pois com o código do processamento escrito em uma linguagem de descrição de hardware, VHDL ou Verilog por exemplo, e com o uso de uma ferramenta de síntese, este código pode ser reutilizado para diferente FPGAs, pois a única diferença é a síntese para a FPGA alvo, permanecendo o código inalterado. (ALI, 1996) e (LINDH JOHAN STäNER, 1996) mostram em seus trabalhos, que FPGAs apresentam um meio eficiente para desenvolvimento e produção de sistemas digitais.

1.2 Objetivos

A seguir são apresentados os objetivos gerais e específicos para este trabalho.

1.2.1 Objetivos Gerais

O principal objetivo deste trabalho é desenvolver um core de hardware, com arquitetura de 16 bits, descrito através da linguagem VHDL, para a detecção de sinais monotônicos da telefonia, tendo como plataforma alvo uma FPGA da *Xilinx*.

1.2.2 Objetivos Específicos

O trabalho proposto tem os seguintes objetivos específicos:

1. Desenvolver um hardware controlado, com interface simples, para fácil inserção em um sistema;
2. Processar oito canais de detecção;
3. Realizar o processamento dos canais sem perder amostras de áudio.

1.3 Organização do trabalho

O trabalho está organizado em cinco capítulos, incluindo este capítulo de introdução. O segundo capítulo faz um resumo da teoria de processamento de sinais utilizado na detecção dos áudios, aborda temas como a transformada de fourier, transformada discreta de fourier e algoritmo de goertzel. O terceiro detalha o desenvolvimento do hardware do core de detecção de tons, descrevendo todos seus blocos de hardware internos, assim como a comunicação entre eles e suas funcionalidades. O quarto capítulo apresenta os resultados obtidos da simulação do hardware e da ferramenta de síntese. O ultimo capítulo faz a conclusão e apresenta sugestão para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo está a base teórica, de forma resumida, das operações matemáticas efetuadas neste trabalho para o processamentos do áudios. Na Seção 2.1 é apresentada a transformada de *Fourier*, assim como sua importância na área de processamento de sinais. A transformada discreta de *Fourier* é apresentada na Seção 2.2, esta que é uma versão discretizada da transformada de *Fourier*. Na última seção deste capítulo, 2.3, apresentamos o algoritmo de *Goertzel*, que é uma versão derivada da transformada discreta de *Fourier*. É com o uso do algoritmo de *Goertzel* que são efetuados os processamentos de sinais deste trabalho. As seções abordadas neste capítulo, tiveram como fonte de estudo (OPPENHEIM, 1998).

2.1 Transformada de Fourier

A transformada de *Fourier* é uma operação matemática de fundamental importância no campo de processamento de sinais. Em essência, este operador faz a transição de funções que têm se comportamento descrito no tempo, para o domínio da frequência, com o caminho inverso também sendo possível, permitindo a análise das componentes de frequência do sinal.

A transformada de *Fourier* tem sua origem na série de *Fourier*, que é uma forma de representação de funções periódicas através da combinação linear de funções trigonométricas. A equação 2.5 representa a série de *Fourier*. Os valores do parâmetros da série são encontrados usando as relações apresentadas em 2.2.

$$f(t) = a_0 + \sum_{n=1}^{\infty}[a_n cos(n\omega_0 t) + b_n sen(n\omega_0 t)] \tag{2.1}$$

$$a_0 = \frac{1}{T}\int_{t_0}^{t_0+T} f(t)dt \tag{2.2}$$

$$a_n = \frac{2}{T}\int_{t_0}^{t_0+T} f(t)cos(n\omega_0 t)dt \tag{2.3}$$

$$b_n = \frac{2}{T}\int_{t_0}^{t_0+T} f(t)sen(n\omega_0 t)dt \tag{2.4}$$

Representado a série de *Fourier* na forma exponencial, temos:

$$f(t) = \sum_{n=-\infty}^{\infty} C_n e^{jn\omega_0 t}, \text{ tendo o valor de } c_n \text{obtido de:} \tag{2.5}$$

$$\tag{2.6}$$

$$C_n = \frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)e^{-jn\omega_0 t}dt \tag{2.7}$$

Para um caso limite da série de *Fourier*, fazendo $T \to \infty$, o que significa dizer que a função é aperiódica, a distância entre as harmônicas $n\omega_0$ e $(n+1)\omega_0$ torna-se cada vez menor. Matematicamente temos:

$$\Delta\omega = (n+1)\omega_0 - n\omega_0 = \omega_0 = \frac{2\pi}{T} \Rightarrow \frac{1}{T} = \frac{\Delta\omega}{2\pi} \tag{2.8}$$

$$\text{Fazendo } T \to \infty \Rightarrow \frac{1}{T} \to \frac{d\omega}{2\pi} \tag{2.9}$$

Portando, fazer $T \to \infty \Rightarrow n\omega_0 \to \omega$, ou seja, frequência deixa de ser uma variável discreta e passa a ser uma variável contínua. Portando a partir de 2.5, tendo que $T \to \infty$, temos:

$$C_n T \to \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt \tag{2.10}$$

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt \tag{2.11}$$

A integral presente na equação 2.10 é conhecida como a transformada de *Fourier*. A transformada inversa de *Fourier* é obtida através de 2.5, fazendo $T \to \infty$, como é mostrado a seguir:

$$f(t) = \frac{T}{T}\sum_{n=-\infty}^{\infty} C_n e^{jn\omega_0 t} = \sum_{n=-\infty}^{\infty} (C_n T)e^{jn\omega_0 t}\frac{1}{T} \tag{2.12}$$

$$\tag{2.13}$$

Com $T \to \infty$, temos que o somatório torna-se uma integral e que:

$$C_n T \to F(\omega) \tag{2.14}$$

$$\frac{1}{T} \to \frac{d\omega}{2\pi} \tag{2.15}$$

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{j\omega t}d\omega \tag{2.16}$$

2.2 Transformada Discreta de Fourier

Veremos nessa seção que a transformada discreta de *Fourier* (DFT) é uma versão discretizada da transformada de *Fourier*, e que o resultado do cálculo dos coeficientes da DFT fornece uma aproximação para seus respectivos coeficientes da série de *Fourier*.

Considerando uma sequência periódica $x[n]$, com período N, tal que $x[n] = x[n + rN]$, com $r, n \in Inteiros$. Temos que a representação de $x[n]$ pela série de *Fourier* é dada por $x[n] = \sum_{k=0}^{N-1} X[k]e^{j\frac{2\pi}{N}kn}$, com o cálculo dos coeficientes dado por $X[k] = \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}kn}$. Fazendo $W_N = e^{-j\frac{2\pi}{N}}$, temos:

$$\text{Equação de análise :} x[n] = \sum_{k=0}^{N-1} X[k]W_N^{kn} \tag{2.17}$$

$$\text{Equação de síntese :} X[k] = \sum_{n=0}^{N-1} x[n]W_N^{-kn} \tag{2.18}$$

Que é a representação da série discreta de *Fourier* (DFS). Do cálculo dos coeficientes da DFS, temos que:

$$X[k] = \begin{cases} \sum_{n=0}^{N-1} x[n]W_N^{kn}, & 0 \leq k \leq N-1, \\ 0, & \textit{Caso contrário} \end{cases} \tag{2.19}$$

$$x[n] = \begin{cases} \sum_{k=0}^{N-1} X[k]W_N^{-kn}, & 0 \leq k \leq N-1, \\ 0, & \textit{Caso contrário} \end{cases} \tag{2.20}$$

O equação 2.19 diz respeito a cálculo da transformada discreta de *Fourier*

2.3 Algoritmo de Goertzel

O algoritmo de *Goertzel* é uma derivação direta da transformada discreta de *Fourier*, usado para reduzir o tempo de computação, portanto, diminuir o custo computacional do cálculo dos coeficientes da DFT. Tem seu uso indicado para casos em que é necessário calcular apenas alguns pontos do espectro de frequência. O algoritmo de Goertzel é amplamente utilizado na área de processamento de sinais, em (ZáPLATA, 2014) é apresentado este algoritmo sendo usado como um filtro. (SELMAN, 2007) mostra uma comparação, que inclui o algoritmo de Goertzel, mostrando seu desempenho em relação à outra técnicas na detecção DTMF. (YUYING, 2014), em seu trabalho, fala sobre a utilização do algoritmo de Goertzel na detecção de tons DTMF, assim como sua implementação em *MATLAB*.

Na Seção 2.2, definimos $W_N = e^{-j\frac{2\pi}{N}}$, fazendo $n = N$, temos:

$$W_N^{-kN} = e^{j\frac{2\pi}{N}kN} = e^{j2\pi k} = 1 \tag{2.21}$$

Portanto temos a partir da equação 2.19:

$$X[k] = \sum_{r=0}^{N-1} x[n] W_N^{kr} = W_N^{-kN} \sum_{n=0}^{N-1} x[n] W_n^{kr} = \sum_{n=0}^{N-1} x[n] W_n^{-k(N-r)} \tag{2.22}$$

Definindo a função de resposta ao impulso temos:

$$h_k[n] = W_N^{-kn} u[n] = e^{j\frac{2\pi}{N}kn} u[n] \tag{2.23}$$

Com a resposta ao impulso temos como calcular a saída:

$$y_k[n] = x[n] * h_k[n] \tag{2.24}$$

Podemos notar que:

$$y_k[n] = \sum_{r=-\infty}^{\infty} x[r] h_k[n-r] \tag{2.25}$$

$$= \sum_{r=0}^{\infty} x[r] W_N^{-k(n-r)} u[n-r], \textit{ pois } x[r] = 0, \textit{ para } r < 0 \tag{2.26}$$

$$= \sum_{r=0}^{n} x[r] W_N^{-k(n-r)}, \textit{ pois } u[n-r] = 0, \textit{ para } r > n \tag{2.27}$$

O valor de $y_k[n]$ para $n = N$ é:

$$y_k[N] = \sum_{r=0}^{N} x[r] W_N^{-k(N-r)} \tag{2.28}$$

$$= \sum_{r=0}^{N-1} x[r] W_N^{-k(N-r)}, \textit{ ou seja} \tag{2.29}$$

$$X[k] = y_k[n]|_{n=N} \tag{2.30}$$

Portanto podemos interpretar que a DFT de uma sequência $x[n]$ é a saída de um sistema linear invariante no tempo, com a resposta impulsiva $W_N^{-kn} u[n]$, calculada no instante $n = N$.

Da resposta ao impulso, $h_k[n]$, temos que a função de transferência do sistema é:

$$H_k(z) = \frac{1}{1 - W_N^{-k} z^{-1}} \tag{2.31}$$

E a equação de diferenças:

$$y_k[n] = x[n] + W_N^{-k} y_k[n-1], \textit{ com } y_k[-1] = 0 \tag{2.32}$$

Para cada $y_k[n]$ calculado, são executadas 4 multiplicações reais, ou 1 complexa. Como deve ser calculado até $n = N$, têm-se N multiplicações complexas e $4N$ multiplicações reais.

A partir da expressão de $H_k(z)$, temos:

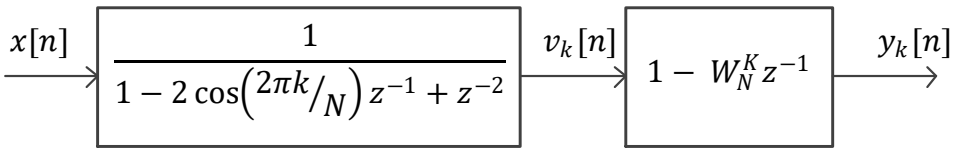
$$H_k(z) = H_k(z)\frac{1 - W_N^{-k} z^{-1}}{1 - W_N^{-k} z^{-1}} \tag{2.33}$$

$$= \frac{1 - W_N^{-k} z^{-1}}{(1 - W_N^{-k} z^{-1})(1 - W_N^{-k} z^{-1})} \tag{2.34}$$

$$= \frac{1 - W_N^{-k} z^{-1}}{1 - 2\cos(\frac{2\pi k}{N}) z^{-1} + z^{-2}} \tag{2.35}$$

$$= \left(\frac{1}{1 - 2\cos(\frac{2\pi k}{N}) z^{-1} + z^{-2}}\right)\left(1 - W_N^{-k} z^{-1}\right) \tag{2.36}$$

O sistema $H_k(z)$ pode ser implementado através da seguinte forma:



Temos que:

$$v_k[n] = x[n] + 2\cos(\frac{2\pi k}{N}) v_k[n-1] - v_k[n-2], v_k[-1] = v_k[-2] = 0, \textit{ e} \tag{2.37}$$

$$y_k[n] = v_k[n] - W_N^k v_k[n-1] \tag{2.38}$$

Temos que o sinal $v_k[n]$ é calculado para o intervalo $0 \leq n \leq N$, e o sinal $y_k[n]$ só precisa ser calculado para $n = N$. Para o cálculo do algoritmo, também é preciso armazenar os coeficientes $\cos(\frac{2\pi k}{N})$ e W_N^k para cada k.

3 DESENVOLVIMENTO

Neste capítulo é apresentado o processo de desenvolvimento do core de detecção de tons, abordando as metodologias de desenvolvimento e teste, assim como explicação detalhada dos blocos pertencentes ao projeto, apresentando funcionalidades e de que maneira se comunicam com outros blocos de hardware.

O core proposto foi projetado para ser inserido em um sistema, no qual, ele será responsável pelo processamento matemático dos áudios de sinalização monotônicos, de forma que ele é um elemento controlado por esse sistema.

3.1 Metodologia de Desenvolvimento

Para o desenvolvimento deste trabalho foi aplicada a abordagem top-down, a qual se consiste em iniciar o desenvolvimento a partir dos blocos mais externos, baixando o nível de desenvolvimento até os blocos mais internos. Seguindo essa abordagem, o desenvolvimento seguiu a ordem: *Top Module; Tone Detector; Command Handler; Memory Control Access; Memory; Signal Process; Message Handler* e *Message Buffer*.

3.2 Ambiente de Desenvolvimento

Pelo fato deste trabalho ser desenvolvido tendo como alvo uma FPGA, o ambiente de desenvolvimento utilizado foi o software *Integrated Software Environment* (ISE) 14.7, tendo como hardware alvo a FGPA XC6VLX240T, da família virtex 6. Tanto o ambiente de desenvolvimento, quanto a FPGA são produzidos pela *Xilinx*.

3.3 Parâmetros da Detecção

Os áudios detectados pelo core apresentado neste trabalho são sinais monotônicos, que podem ser compostos por intercalações de pulsos e pausas. Os áudios também podem ter repetições de sequências de pulsos e pausas, essa característica é chamada de cadência. Essas características podem ser melhor visualizadas na Figura 1.

Para a detecção do áudio ser efetuada com sucesso, todos os parâmetros devem ser passados para o detector de tons. No total são 9 parâmetros, que detalham as características do áudio a ser detectado, são eles: *MaxPulse*; *MinPulse*; *MaxPause*; *MinPause*; *Attenuation*; *Cadence*; *Threshold*; *Coeff1* e *Coeff2*. Cada parâmetros está detalhado a seguir.

- **MaxPulse e MinPulse**: Os primeiros parâmetros da detecção são relativos a duração de pulso do áudio. *MaxPulse* e *MinPulse* são respectivos á duração máxima e mínima que o sinal deve ter. Estando o sinal processado fora desses limites, implica que este áudio não está de acordo com o que se esperava, portanto, será entendido como áudio não detectado.

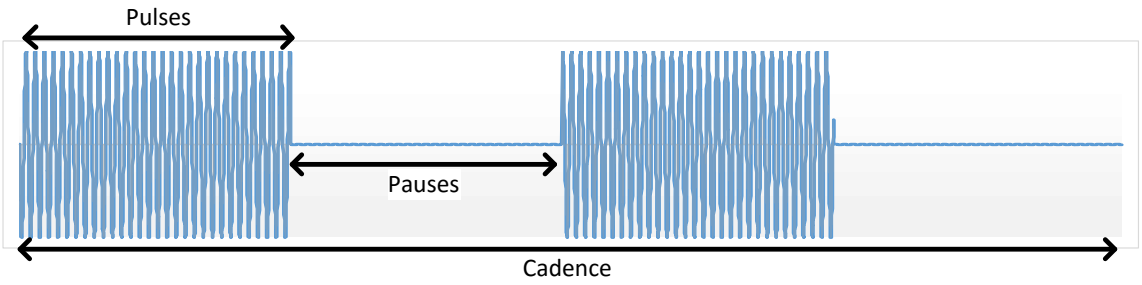


Figura 1 – Pulsos, Pausas e Cadência

- **MaxPause e MinPause**: Análogo aos parâmetros de duração de pulso, só que *MaxPause* e *MinPause* se referem à duaração das pausas presentes no sinal. Entende-se como pausa, a parte do áudio que está presente o silêncio.

- **Attenuation**: Dependendo da potência do sinal que estiver chegando na porta, talvez seja necessário fazer uma atenuação de cada amostra do sinal. A atenuação é efetuada reduzindo a amplitude do sinal sempre em potências de dois. De acordo com a fórmula a seguir, onde, S é a amostra que está chegando na porta, e Sa é o valor da amostra atenuada.

$$Sa = \frac{S}{2^{Attenuation}} \tag{3.1}$$

- **Cadence**: A cadência é uma repetição de uma sequência de pulso e pausa. Na Figura 1 pode ser visto um sinal com cadência 0, ou seja, o sinal não tem repetição. Um exemplo de sinal com cadência é o tom de ocupado, utilizado na telefonia.

- **Threshold**: Ao final do processamento de uma sequência de amostras, é feito o cálculo da sua potência, tendo este valor, temos que ter algum parâmetro que informe se esta potência refere-se a pulso ou pausa, este parâmetro de definição é o *Threshold*, que é o limiar utilizado para fazer tal classificação, que segue a seguinte lei:

$$\textit{Sequência} = \begin{cases} pulso, & \text{se} \quad \textit{Potência} \geq Threshold, \\ pausa, & \text{se} \quad \textit{Potência} < Threshold. \end{cases} \tag{3.2}$$

- **Coeff1 e Coeff2**: Estes parâmetros são relativos a frequência do áudio que se deseja detectar, *Coeff1* e *Coeff2* informam para o detector de tons a faixa de frequência em que o sinal a ser processado deve estar presente. Neste trabalho proposto, este intervalo é de $20Hz$, ou seja, se o áudio a ser processado tem frequência igual $425Hz$, os valores de *Coeff1* e *Coeff2* são correspondentes à $415Hz$ e $435Hz$.

Os valores dos coeficientes vêm da equação:

$$Coefficient = 2\cos\left(\frac{2\pi f}{f_s}\right) \tag{3.3}$$

Onde f e f_s são respectivamente a frequência desejada e a frequência de amostragem. Como na telefonia a frequência de amostragem dos sinais é $8KHz$, a máxima frequência detectável é $4KHz$, como pode ser provado no teorema da amostragem de Nyquist.

3.4 Desenvolvimento do Core

3.4.1 Top Module

A instância mais alta do projeto é o *Top Module*, neste bloco, ficam instanciados o hardware que realmente será sintetizado, *Tone Detector*, e também a unidade de testes, que é responsável pela inserção de comandos e áudios, assim como recebimentos de mensagens para análise dos resultados. A Figura 2 ilustra o *Top Module*.

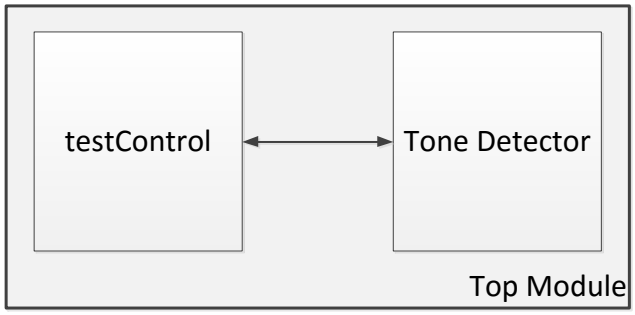


Figura 2 – Top Module

3.4.2 Tone Detector

O módulo *Tone Detector*, assim com o *Top Module*, é um bloco que somente instancia outros blocos, a diferença aqui é que este bloco instancia o Hardware a ser sintetizado. Dentro dele estão os módulos que são responsáveis por todo o processamento. O *Tone Detector* é responsável por interligar as interfaces de comando, mensagens e áudio aos blocos responsáveis por cada tarefa. A Figura 3 mostra uma visão geral deste módulo.

Nas próximas Seções será detalhado cada bloco contido dentro do *Tone Detector*, seguindo a sequência: *Command Handler; Memory Access Controller; Signal Processor e Message Handler.*

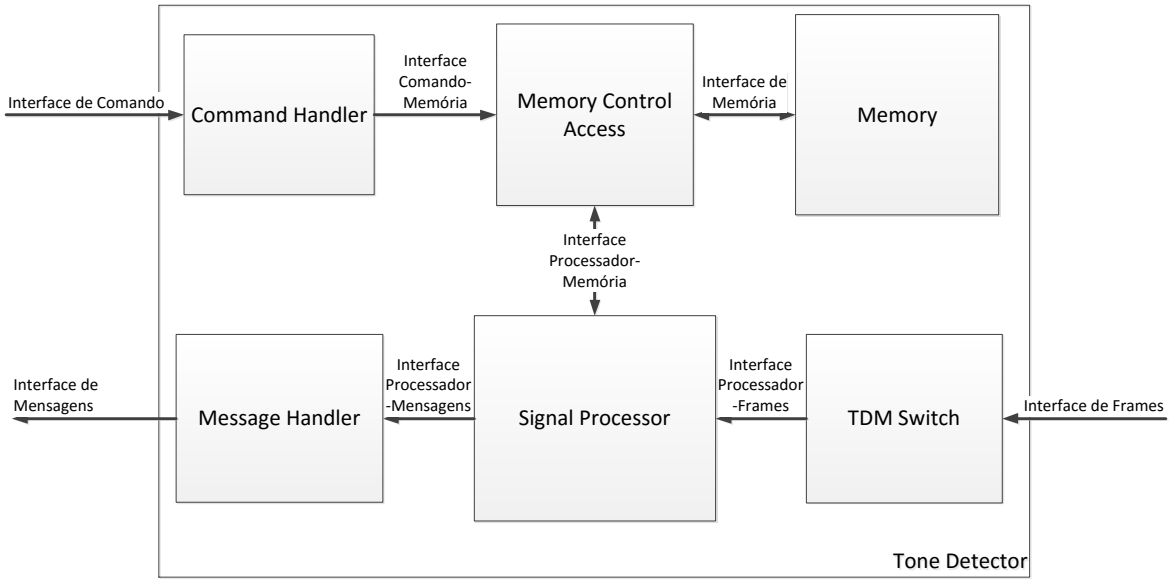


Figura 3 – Módulo Tone Detector

3.4.3 Tratamento dos Comandos

Para o processamento de um áudio ter seu início, os parâmetros da detecção para este áudio devem estar presente na região de memória reservada para a porta na qual o tom irá ser detectado. A passagem dos parâmetros é feita através da interface de comando do *Tone Detector*, esta interface está conectada com o bloco de hardware *Command Handler*, que é responsável por decodificar a porta na qual será efetuada a detecção, e escrever na memória os parâmetros para tal detecção.

Internamente ao *Command Handler*, existe um buffer de comando, no qual ficam armazenados os parâmetros, temporariamente, até que o acesso à memória seja liberado pelo controlador de acesso à memória. O *Command Buffer* é uma pequena memória de 7x16 bits, a qual recebe os parâmetros para a detecção. O módulo *Command Handler* é representado pela Figura 4, a seguir.

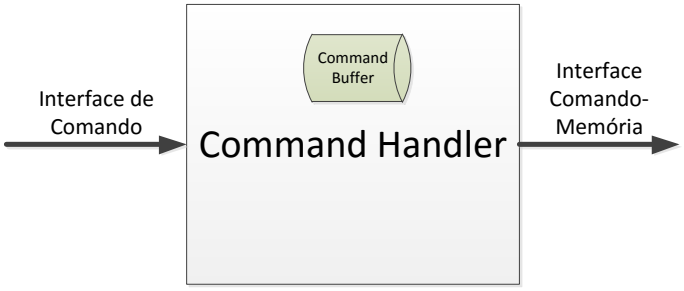


Figura 4 – Command Handler

Command Handler

O módulo *command handler* é responsável por receber os parâmetros do sinal a ser detectado e escrever esses parâmetros na memória. À medida que cada parâmetro é recebidos, são escritos no *command buffer*, para serem posteriormente escritos na memória.

A comunicação na interface de comando é estabelecida através de um *handshake*, onde primeiramente o controlador do core faz um pedido de requisição de envio dos parâmetros, após a requisição ser atendida, os mesmos serão enviados. A requisição, recebimento dos parâmetros, armazenamento no *command buffer* e escrita na memória seguem uma sequência, que está definida na máquina de estados do *command handler*, que é representada pelo diagrama de máquina de estado na Figura 5.

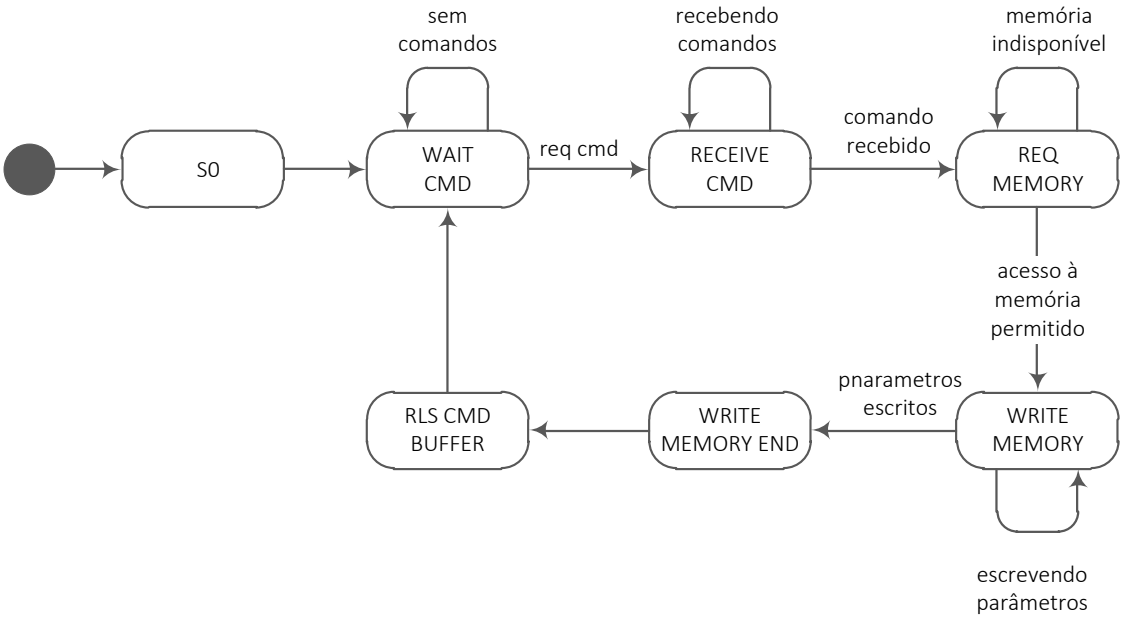


Figura 5 – Máquina de Estados do Command Handler

O primeiro estado, **S0**, é o estado em que este módulo entra no reset do sistema, sua função é resetar os sinais contidos no *command handler*. Após o período de reset, a máquina avança para o estado **WAIT_CMD**, no qual fica esperando uma requisição de recebimento de comando. Enquanto não houver requisição, a máquina permanecerá no mesmo estado.

Com o recebimento da requisição, o estado é alterado para **RECEIVE_CMD**, dando início ao recebimento dos parâmetros, contidos no comando. É recebido uma *WORD* de 16 bits por vez, cada uma contendo um ou mais parâmetros. A sequência, e a organização dos parâmetros dentro de cada *WORD* é mostrada na Figura 6:

Sequência	Bits 15..8	Bits 7..0
0	Port Number	
1	Max Pulse	Min Pulse
2	Max Pause	Min Pause
3	Attenuation	Cadence
4	Threshold	
5	Coefficient 1	
6	Coefficient 2	

Figura 6 – Parâmetros da Detecção

Após recebimento de todos os parâmetros, a máquina de estados avança para o estado **REQ_MEMORY**, que é o estado responsável por fazer a requisição de acesso à memória. O módulo responsável por atender esta solicitação é o *Memory Access Controler*, que será detalhado na Seção 3.4.4.

Com o acesso à memória liberado, acontece um avanço na máquina de estado, indo para o estado **WRITE_MEMORY**, iniciando assim a escrita dos parâmetros na memória. Cada canal tem um espaço reservado na memória de 12 *WORDS* de 16 bits, onde ficam contidos os parâmetros recebidos via comando, e outros parâmetros que são utilizados pelo *Signal Processor*. O detalhamento de cada parâmetro e sua utilização será explanado na Seção 3.4.6.

Com a escrita dos parâmetros finalizada, a máquina de estado do *Command Handler* avança para o estado **WRITE_MEMORY_END**. Esse estado fecha a conexão com a memória, fazendo com que ela esteja disponível no próximo ciclo de processamento do *Signal Processor*.

Depois de concluída a escrita dos parâmetros, a máquina avança para o último estado, que é responsável por resetar alguns sinais internos e o buffer de comando, o deixando pronto para receber um novo comando, esse estado é o **RLS_CMD_BUFFER**. Após este reset, a máquina de estados do *Command Handler* volta para o estado **WAIT_CMD**, assim ficando disponível para receber um novo comando. Enquanto *Command Handler* não voltar ao estado

WAIT_CMD, qualquer requisição de um novo comando, não será atendida.

3.4.4 Controle de Acesso à Memória

Devido ao acesso de memória poder ser requisitado por dois blocos de hardware, um controlador de acesso foi necessário, para que não houvesse uma política no acesso à memória. O bloco responsável por esse controle é *Memory Access Control*, representado pela Figura 7.

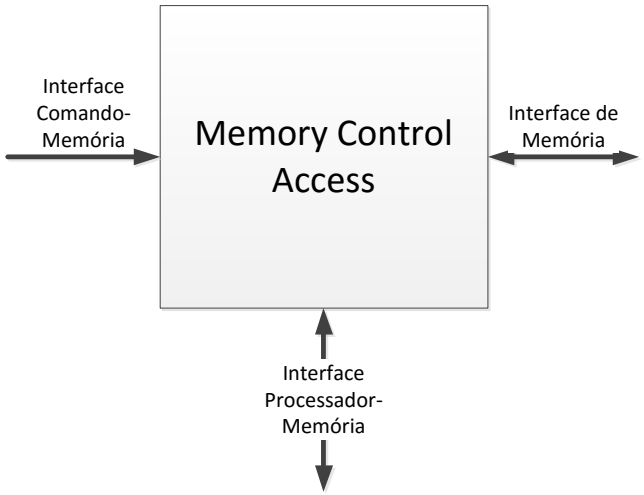


Figura 7 – Bloco de controle de acesso à memória

Este módulo tem seu funcionamento semelhante a um multiplexador, fazendo com que em alguns momentos o acesso à memória seja do processador de sinais, e em outros momentos este acesso seja do *command handler*.

O funcionamento do *Memory Access Control* é regido através de uma máquina de estados, que fica responsável por liberar o acesso à memória. A máquina de estados do controle de acesso é apresentada na Figura 8:

O estado inicial **S0** é o estado de *reset*, sendo responsável por resetar os sinais internos do módulo, e deixando este apto para seu funcionamento. Depois do reset dos sinais internos, a máquina de estado avança para o estado **PROCESSOR_TIME**, que é o estado em que o acesso à memória fica sob o domínio do *Signal Processor*.

Devido à maior prioridade que o processamento tem, em relação ao recebimento dos parâmetros vindos do *command handler*, o controlador só libera o acesso da memória ao *Command Handler* na ocorrência de dois eventos simultaneamente, o primeiro é se houver uma requisição do *command handler*, o segundo evento acontece quando o *Signal Processor* não estiver processamento nenhum sinal, neste momento o processador fica parado, esperando novas amostras. Esse comportamento do *Signal Processor* será explicado na Seção 3.4.6.2.

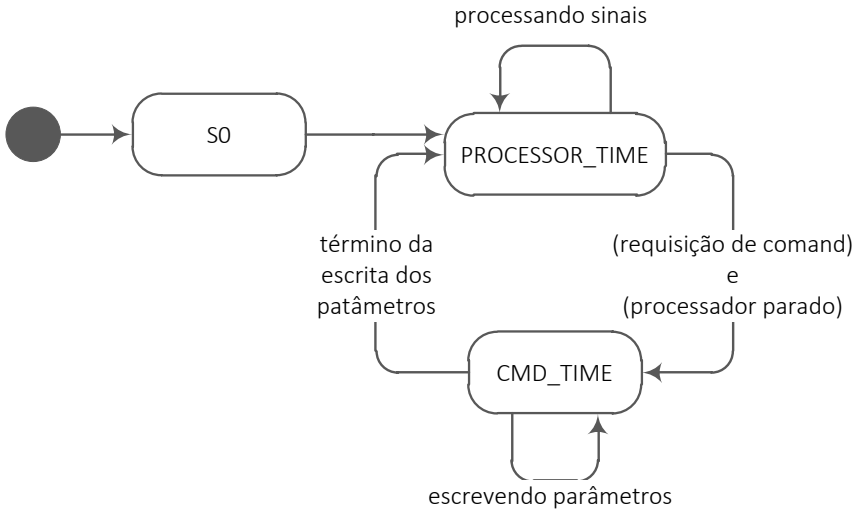


Figura 8 – Máquina de Estado do Controle de Acesso à Memória

Na ocorrência dos eventos citados, a máquina de estados avança para o estado **CMD_TIME**, liberando assim o *command handler* para efetuar escrita dos parâmetros para uma nova detecção. Ao final da escrita dos parâmetros a máquina de estados do *Memory Access Control* retorna para o estado **PROCESSOR_TIME**, assim disponibilizando a memória para o processador novamente.

Devido ao fato de a memória ser acessada por dois blocos de hardware diferentes, tem que haver alguma margem de segurança para que não sejam criadas regiões críticas na memória. Esta margem é assegurada baseada no tempo em que o processador efetua suas operações na memória, pois, devido à velocidade com que acontece o processamento, existe tempo suficiente para tratar todos os oito canais e ainda receber comandos. Os detalhes dos tempos serão apresentados na Seção 4, que aborda e discute os resultados deste trabalho.

3.4.5 Memória

Para que um áudio, que chega através de um canal de entrada, ser detectado corretamente pelo processador, é necessário que o mesmo esteja de acordo com os parâmetros da detecção passados via comando. Para validar se o áudio tem as características descritas de acordo com os parâmetros, são necessárias algumas variáveis controle. Ambos, parâmetros e variáveis de controle, de todos os canais, ficam na memória do *Tone Detector*.

Cada canal tem sua região na memória, são oito canais e cada região ocupa doze posições de dezesseis bits, portanto totalizando uma memória de 96x16 bits de tamanho. A Figura 9 mostra a organização dos parâmetros e variáveis para cada canal de detecção.

Endereço	Bits 15..12	Bits 11..8	Bits 7..4	Bits 3..0
0	Counter of Samples		Flag Message Sent	State
1	Max Pulse		Min Pulse	
2	Max Pause		Min Pause	
3	Attenuation		Counter of Cadence	Cadence
4	Threshold			
5	Coefficient 1			
6	Coefficient 2			
7	Sample(-1) Coefficient 1			
8	Sample(-2) Coefficient 1			
9	Sample(-1) Coefficient 2			
10	Sample(-2) Coefficient 2			
11	Counter of Pulse		Counter of Pause	

Figura 9 – Organização da memória para cada canal

As variáveis de controle utilizadas no processamento dos áudios são: *State*; *Flag Message Sent*; *Counter of Cadence*; *Sample(-1) Coefficient 1*; *Sample(-2) Coefficient 1*; *Sample(-1) Coefficient 2*; *Sample(-2) Coefficient 2*; *Counter of Samples*; *Counter of Pulses* e *Counter of Pauses*. A seguir é mostrada a utilidade de cada variável.

- **State**: Variável que guarda qual o estado da detecção em que o áudio está. São quatro estados possíveis, que são pertencentes à máquina de estados da detecção, que será detalhada na Seção 3.4.6.1.

- **Flag Message Sent**: Flag que informa se a mensagem da detecção, sendo efetuada com sucesso ou não, foi enviada ao *Message Handler*, para através deste, ser enviada para o responsável por receber as mensagens do *Tone Detector*.

- **Counter of Cadence**: Variável responsável por guardar a contagem de cadências que o sinal tiver ao longo da detecção. Essa variável só tem seu valor alterado quando o valor da cadência é maior que zero, ou seja, o sinal que será detectado tem repetições.

- **Samples Coefficient**: Por conta da utilização do algoritmo de Goertzel, que é usado para efetuar as detecções do *Tone Detector*, não é necessário guardarmos todas as amostras para verificar se uma frequência específica está presente no sinal, só é preciso guardar os dois últimos resultados. Este dois últimos resultado são respectivamente *Sample(-1)* e *Sample(-2)*. Como o processamento acontece sempre com um cálculo do algoritmo de Goertzel para duas frequência, que são representadas pelos parâmetros *Coeff1* e *Coeff2*, temos que guardar os dois últimos resultados para cada coeficiente, assim temos *Sample(-1) Coefficient 1* e *Sample(-2) Coefficient 1*, que são os últimos resultado para o primeiro coeficiente, e *Sample(-1) Coefficient 2* e *Sample(-2) Coefficient 2*, últimos resultados para o segundo coeficiente.

- **Counter of Samples**: É a variável responsável por fazer a contagem de cada amostra de áudio que estiver chegando nos canais de recepção. Cada oitenta amostras de sinal, que corresponde a $10ms$ de áudio, equivale a um frame, que dependendo da potência pode ser pulso ou pausa.

- **Counter of Pulses**: Variável que guarda a quantidade de pulsos presentes no sinal que está sendo processado. A cada frame de áudio detectado, esta variável tem seu valor incrementado de um.

- **Counter of Pauses**: Análogo à *Counter of Pulses*, só que a variável *Counter of Pauses* é responsável por fazer a contagem das pausas.

3.4.6 Processador de Sinal

Nesta Seção é apresentado o principal bloco de hardware deste trabalho, o *Signal Processor*, representado pela Figura 10, é responsável por efetuar processamento dos áudios que chegam nos canais de entrada, verificar se a detecção ocorreu de acordo com os parâmetros recebidos e envio da mensagem ao módulo de tratamento de mensagens, informando se a detecção aconteceu com sucesso, ou não.

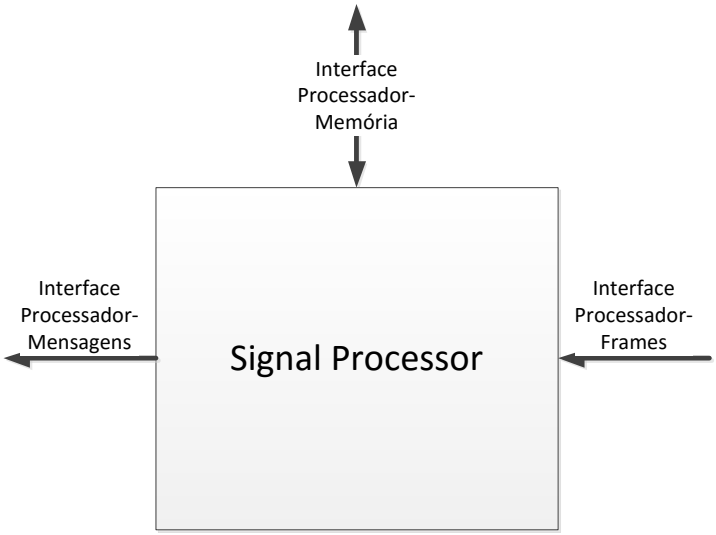


Figura 10 – Bloco do Processador de Sinais

No módulo *Signal Processor* existem duas máquinas de estado, que juntas realizam as tarefas pertinentes ao processamento dos sinais. A primeira é a máquina de estados da detecção, que determina em qual passo a detecção está, assim como a definição de quais os próximos passos, sendo uma para uma para cada canal e estando as informações para seu funcionamento na região de memória de cada canal. Esta máquina tem quatro estados, que são responsáveis por identificar se o sinal tem amplitude suficiente para ser processado, detectar pulsos, pausas e enviar mensagens.

A segunda máquina de estado é a responsável pelo gerenciamento do hardware do *Signal Processor*, esta é única para todos canais. Seu andamento é definido através de análise do estado atual, para cada canal, da máquina de estados da detecção. A máquina de estados do hardware é quem efetivamente lê da memória cada canal, efetua cálculos, verificações e envio de mensagens.

Nas Seções 3.4.6.1 e 3.4.6.2 serão detalhadas as máquinas de estados presentes no *Signal Processor*, demonstrando todos os estados, assim como que a máquina de estado da detecção influi no andamento da máquina do hardware.

3.4.6.1 Máquina de Estados da Detecção

A máquina de estados da detecção, como dito antes, tem como papel gerenciar o processamento dos áudios. Cada canal de detecção tem sua própria máquina, pelo fato de que o processamento dos áudio são independentes entre si. Para tal gerenciamento, esta máquina é constituída pelos estados *DT_STATE0*, *DT_STATE1*, *DT_STATE2* e *DT_STATE3*, que de maneira resumida, são respectivamente responsáveis por verificar se o sinal em processamento tem amplitude necessária para início da detecção, detectar pulsos presentes no sinal, detectar intervalos de pausas e envio da mensagem da detecção, a Figura 11 mostra o diagrama da máquina de estado da detecção.

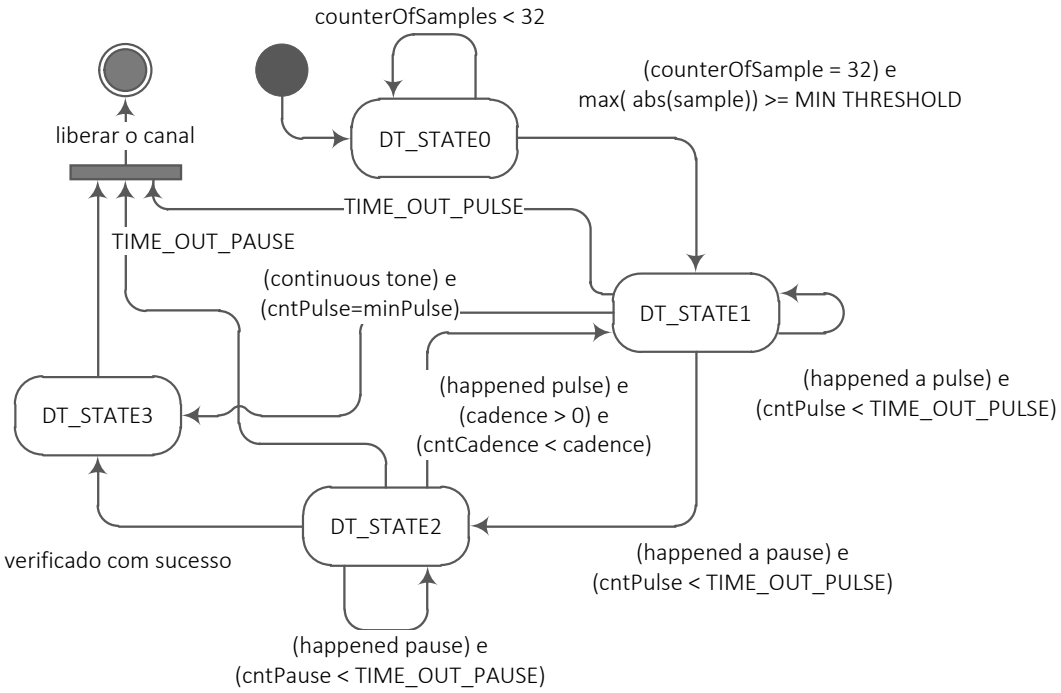


Figura 11 – Máquina de Estados da Detecção

Estado da Detecção 0

Após o recebimento do comando, contendo os parâmetros da detecção, e a escrita destes na memória, o canal está pronto para iniciar a detecção do áudio. O primeiro passo a ser feito no processamento do sinal, é a verificação das amplitudes das primeiras amostras, fazendo com que o processamento não seja iniciado até que um sinal contendo pulsos seja conectado na porta de entrada do canal.

A verificação é feita após o recebimento de trinta e duas amostras consecutivas, o que equivale a $4ms$ de áudio, sempre guardando na memória o módulo da amostra de maior valor absoluto. Após as trinta e duas amostras serem recebidas, pega-se a amostra armazenada na memória, fazendo a comparação desta com a constante *MIN_THRESHOLD*. Se a amostra for maior ou igual, a máquina de estados avança para o estado *DT_STATE1*, se não, permanece no estado *DT_STATE0*, e refaz a verificação de mais trinta e duas amostras.

Estado da Detecção 1

Após constatar que o sinal é detectável, é dado início ao processamento dos pulsos presentes no áudio, que é efetuado no estado *DT_STATE1*. Para a detecção dos pulsos, é utilizado o algoritmo de *Goertzel*, que faz o cálculo de uma DFT de forma iterativa, para uma única frequência, não necessitando guardar todas as amostras para o resultado final. São necessárias 80 amostras de áudio para a detecção de um pulso, equivalendo a $10ms$ de áudio.

A máquina de estados da detecção permanecerá no estado *DT_STATE1*, contabilizando pulsos enquanto não acontecer um *timeout*. O valor de *timeout* está definido na constante, definida em tempo de projeto, *TIME_OUT_PULSE*, que tem valor igual a 250 pulsos, ou seja, um áudio que tiver uma sequência de pulsos maior ou igual a $2500ms$, terá sua detecção interrompida, e será enviada uma mensagem indicando que aconteceu um *timeout* de pulso. Não acontecendo *timeout*, existe duas possibilidades para a saída do estado *DT_STATE1*.

A primeira possibilidade é se ao decorrer da contabilização dos frames de pulsos, for detectado um frame de pausa. Se essa ação acontecer, será incrementado o valor da variável *Counter of Pauses*, e o estado será alterado para *DT_STATE2*, que é o responsável por contabilizar os frames de pausas.

A outra possibilidade é se o áudio for contínuo e a variável *Counter of Pulses* e o parâmetro *Min Pulse* forem iguais. A informação de que um áudio é contínuo é passado através do parâmetro *Min Pause*, se seu valor for zero, implica que o áudio que se quer detectar não tem frames de pausas, Figura 12.

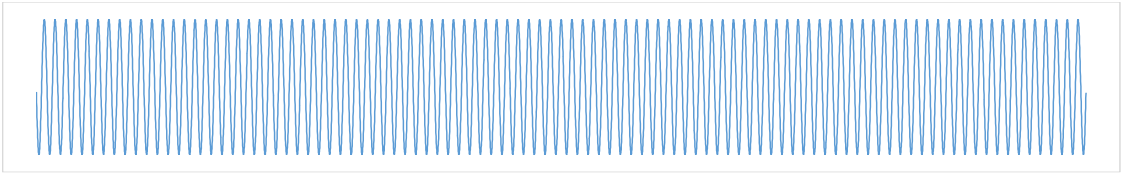


Figura 12 – Pulso Contínuo

Sendo o sinal contínuo, só é necessário que o valor de *Counter of Pulses* se iguale a *Min Pulse*. Se isso ocorrer, o sinal é considerado detectado e o próximo estado será *DT_STATE3*, que é o estado responsável por enviar a mensagem de detecção.

Estado da Detecção 2

O estado *DT_STATE2* é o análogo ao estado *DT_STATE1* para as pausas. O cálculo matemático efetuado também é o algoritmo de *Goertzel*, diferenciando apenas, que nesse estado são contabilizadas as pausas. A maior diferença entre estes estados está na verificação, pois os únicos tipos de sinais que o *DT_STATE1* detecta são áudios contínuos, pois estes não precisam processar pausas. Para ocorrer a detecção de todas as outras variações de sinais, é necessários que a máquina de estados passe pelo *DT_STATE2*.

Depois de entrar no estado *DT_STATE2*, é iniciado o processamento das pausas e a variável *Counter of Pauses* é incrementada para cada frame de pausa detectado, enquanto não houver um *timeout* de pausas, esta variável é incrementada. O valor do *timeout* de pausas é definido através da constante *TIME_OUT_PAUSE*, que tem valor igual a duzentos, implicando em um intervalo de tempo de $2000ms$. Igualmente ao *DT_STATE1*, se houver *timeout* de pausa, a detecção é interrompida e uma mensagem de erro será enviada para sinalizar o erro ocorrido. Estando a detecção no *DT_STATE2*, há três possibilidades de tipos sinais a serem detectados: Sinais sem cadência com pausa contínua; Sinais sem cadência com pausa delimitada; Sinais com cadência. Este são respectivamente representados na Figura 13.

Áudios com pausa contínua, são aqueles em que não existe a necessidade de verificação da quantidade máxima de pausas contidas no sinal. O parâmetro *Max Pause* é o responsável transmitir essa informação ao processamento, como pode ser visto na Eq. 3.4.

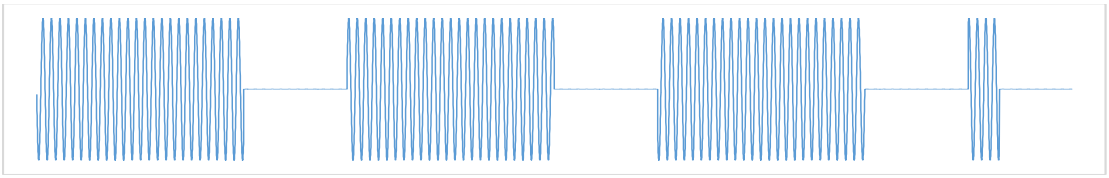
$$Pausa = \begin{cases} \textit{Contínua}, & \text{se} \quad \textit{Pausa Máxima} = 0, \\ \textit{Delimitada}, & \text{se} \quad \textit{Pausa Máxima} > 0. \end{cases} \tag{3.4}$$

Para sinais com pausa, a contagem dos frames de pausa acontece até que o valor mínimo seja alcançado, ou seja, a variável *Counter Of Pauses* tem seu valor incrementado até que seja igual ao parâmetro *Min Pause*. Chegando a este ponto, o sinal é considerado como detectado, e máquina de estado avança para o estado *DT_STATE3*, para o envio da mensagem da detecção.

Para áudios sem cadência, com pausa delimitada, sempre é esperado um frame de pulso após os frames de pausas, este pulso serve para informar ao processador que o período de pausa terminou. Após a chegada do pulso delimitador, o próximo passo é a verificação dos parâmetros de largura máxima e mínima dos pulsos e pausas. É dito que a detecção aconteceu com sucesso, se as variáveis *Counter of Pulses* e *Counter of Pauses* estiverem dentro dos limites máximos e mínimos, ou seja, $(MinPulse \leq CounterOfPulses \leq MaxPulse)$ e $(MinPause \leq CounterOfPauses \leq MaxPause)$. Estando o sinal dentros dos limites especificados pelos parâmetros, este áudio é considerado detectado, e a máquina de estados

a) Pausa Contínua

b) Pausa Delimitada



c) Cadência

Figura 13 – Áudios Detectáveis

avança para o estado *DT_STATE3*. Se os limites não forem respeitado, a máquina volta ao estado *DT_STATE1*, recomeçando o processamento.

Um áudio com cadência é aquele composto por repetições de um sinal com pausa delimitada, ou seja, uma sequência de pulsos e pausas, como pode ser visto na Figura 13-c. Um sinal cadenciado faz a contagem dos frames de pulsos e pausas da mesma maneira, a diferença é que o pulso delimitador também é o início de uma nova sequência. Ao dar-se início a uma nova sequência, o valor da variável *Counter of Cadence* é incrementado, *Counter of Pulses* recebe o valor um, por se tratar do início de uma nova sequência, e *Counter of Pauses* recebe o valor zero, e máquina de estado volta para o estado *DT_STATE1*, para o processamento dos novos pulsos. O sinal com cadência só será considerado detectado quando a variável *Counter of Cadence* for igual ao parâmetro *Cadence*, assim avançando a máquina de estados da detecção para o estado *DT_STATE3*, para o envio da mensagem de detecção.

Estado da Detecção 3

Quando a máquina de estados da detecção chega ao estado *DT_STATE3*, significa que a detecção do canal em questão foi concluída com sucesso, restando somente o envio da mensagem da detecção, que consiste em uma *WORD* de 16 bits, dos quais, o byte mais

significativo tem como valor a constante *TS_PROC_SUCCESS*, que é definida em tempo de projeto, e neste trabalho está com o valor 0xAA. No byte menos significativo tem como valor a porta na qual a detecção foi efetuada, então, um exemplo de uma mensagem de detecção seria o valor 0xAA03, que expressa que o áudio inserido no canal três foi detectado de acordo com os parâmetros passados via comando.

3.4.6.2 Máquina de Estados do Hardware

A máquina de estado da detecção, como apresentado na Seção 3.4.6.1, é responsável pelo processamento de cada canal de áudio, abstraindo as operações de hardware que são necessárias para que a detecção aconteça com sucesso. As operações de hardware são gerenciadas pela máquina de estados de hardware, Figura 14, que é única para todos os canais. O hardware do *Signal Processor* tem como funções: Leitura e escrita na memória; Cálculos matemáticos do algoritmo de *Goertzel* e Envio de mensagens. A seguir são explicados todas as ações e as lógicas de alternância de estado para esta máquina.

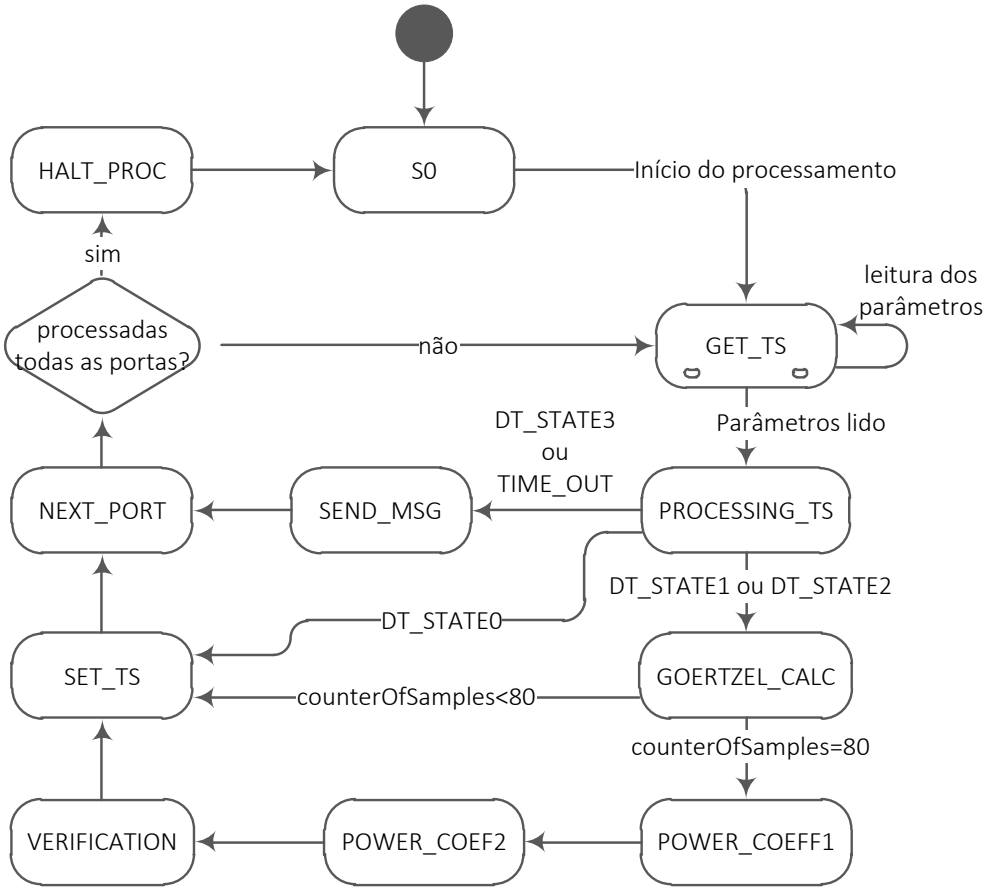


Figura 14 – Hardware State Machine

S0

É o estado inicial da máquina de estados, responsável por resetar os sinais utilizados pelo hardware do *Signal Processor*. A máquina permanece nesse estado apenas um ciclo de clock, avançando para o estado **GET_TS**.

GET_TS

O estado **GET_TS** tem como objetivo fazer a leitura da região de memória de cada canal, para isso o *Signal Processor* conta com um buffer interno de doze posições de dezesseis bits. Este buffer é utilizado para que durante o processamento de cada canal, não seja necessário fazer leituras ou escrita na memória.

Para não haver perca de tempo de processamento, o estado **GET_TS** só lê toda a região de um canal, se o mesmo estiver sendo utilizado para algum processamento. A informação de que o canal não está processando consta no primeiro endereço de memória, dentro da região de memória de cada canal, se este valor for igual a 0xFFFF implica que este canal não está sendo usado para nenhuma detecção.

Com o valor da primeira *word* de posição de memória do canal diferente de 0xFFFF, inicia-se a leitura das *words* restantes, e a escrita deste valores no buffer interno. Com a leitura da memória do canal concluída, é dado início do processamento do canal, avançando a máquina de estados para o estado **PROCESSING_TS**.

PROCESSING_TS

Este estado efetua um pré-processamento das informações lidas da memória, pertinentes ao canal que está sendo processado. Este pré-processamento define o próximo estado baseado no estado atual da máquina de estados da detecção do canal. Existem três estados possíveis para os quais a máquina pode avançar: **SET_TS**; **GOERTZEL_CALC**; **SEND_MSG**.

A máquina de estados do hardware avança para estado **SET_TS** se a máquina de estados da detecção, para o canal, estiver no estado *DT_STATE0*. Nessa situação é que acontece a coleta das primeiras amostras do áudio, como descrito na Seção 3.4.6.1, na explicação do estado *DT_STATE0*. Após a coleta de uma amostra, é feita a comparação se esta tem magnitude maior que a amostra já guardada, nessa comparação sempre permanece a amostra de maior magnitude. Depois de guardada a maior amostra, a variável *Counter of Samples*, que faz a contagem das amostras, é incrementada. O estado **SET_TS** é responsável por copiar a buffer interno para a região de memória do canal, isto é, escrever as alterações que foram feitas pelo processamento.

Uma outra possibilidade de avanço de estado é para **GOERTZEL_CALC**, para isto, o canal tem que está em fase de processamento de pulsos ou pausas, isto é, a máquina de

detecção deste canal tem que está no estado **DT_STATE1** ou **DT_STATE2**, os quais fazem uso do algoritmo de *Goertzel*, que é calculado no estado **GOERTZEL_CALC**.

O último estado possível de ser atingido através do **PROCESSING_TS** é o estado **SEND_MSG**, que é o responsável pela geração e envio das mensagens, essa situação pode ser alcançada se a máquina de estado da detecção estiver no estado **DT_STATE3**, isto é, áudio processado com sucesso, ou se tiver acontecido um *timeout* de pulsos ou um *timeout* de pausas.

GOERTZEL_CALC

Quando a máquina de estados do hardware chega a este estado, implica que frames de pulso ou de pausa estão sendo processados pelo canal de detecção, ou seja, quando a máquina de estado da detecção estiver nos estados **DT_STATE1** ou **DT_STATE2**, o algoritmo de *Goertzel* está sendo calculado para este canal. Este cálculo consiste de duas etapas, na primeira, é feito um cálculo em oitenta amostras seguidas, equivalentes a $10ms$ de áudio, e na segunda etapa é calculada a amplitude na frequência desejada.

A etapa em que o algoritmo está é que decidirá o próximo estado da máquina de estado do hardware. A primeira etapa é sempre calculada para todas as amostras, e a segunda etapa de cálculo é feita na chegada da última amostra. O estado **GOERTZEL_CALC** é responsável pelo cálculo da primeira etapa, e os estados **POWER_COEFF1** **POWER_COEFF2** efetuam os cálculos da segunda fase. Cada etapa é feita para os dois coeficientes de frequência, que são as variáveis *Coefficient 1* e *Coefficient 2*.

O cálculo da primeira fase é realizado tendo os valores da amostra atual do áudio, *Sample(0)*, e os dois últimos resultados do processamento, que estão salvos nas variáveis *Sample(-1)* e *Sample(-1)*. Para o cálculo é utilizada a variável auxiliar *SampleAux*. A expressão matemátiva para o processamento é apresentada a seguir. Este cálculo é feito para cada um dos coeficientes presente na memória.

$$SampleAux = Sample(0) + 2 \cdot Coefficient \cdot Sample(-1) - Sample(-2) \tag{3.5}$$

$$Sample(-2) = Sample(-1) \tag{3.6}$$

$$Sample(-1) = sampleAux \tag{3.7}$$

Quando efetuado o cálculo do octogésimo frame, nesse instante o valor da variável *Counter of Samples* será igual a oitenta, a máquina de estados avança para o estado **POWER_COEFF1**, e inicia o cálculo da segunda etapa do algoritmo de *Goertzel*.

POWER_COEFF1 e POWER_COEFF2

Os estados **POWER_COEFF1** e **POWER_COEFF2** são responsáveis pelo cálculo das amplitudes das frequências correspondentes aos *Coefficient 1* e *Coefficient 2*, respectivamente. A principio esses cálculos poderiam ser realizados no mesmo estado, sem alteração dos resultados, mas o resultado da ferramenta de síntese para o hardware da FGPA conseguiu um clock máximo de $20MHz$. Com os cálculos das amplitudes divididos em dois estados, um para cada coeficiente de frequência, a ferramenta conseguiu sintetizar o hardware com um clock máximo de $88.402MHz$.

Com a primeira etapa do algoritmo de *Goertzel* concluída, isto é, o processamento de oitenta amostras seguidas do áudio, a máquina de estado do hardware avança para o estado **POWER_COEFF1**, e consequentemente para o estado **POWER_COEFF2**. O resultado do cálculo das amplitudes para os dois coeficientes de frequência, *Coefficient 1* e *Coefficient 2*, são escritos respectivamente em dois sinais internos do *Tone Detector*, que são *POT1_s* e *POT2_s*. A expressão matemática utilizada para o cálculo da potência para cada coeficiente é apresentado a seguir na equação 3.8.

$$Power = Sample(-1)^2 + Sample(-2)^2 - 2 \cdot Sample(-1) \cdot Sample(-2) \tag{3.8}$$

Com o cálculo das amplitudes concluído, a máquina de estados do hardware avança para o estado **VERIFICATION**, onde serão verificadas as variáveis de processamento do áudio, e que através de comparações destas com os parâmetros, será tomada a decisão do próximo passo da detecção, do canal que está sendo processado.

VERIFICATION

Na Seção 3.4.6.1 abordamos os estados *DT_STATE1* e *DT_STATE2* da máquina de estados da detecção, estes estados, quando concluíam seus processamentos dos frames, seja de pulso ou de pausa, iniciavam uma fase de verificação para, baseado no parâmetros da detecção, saber se o áudio havia sido detectado ou não. As transições que acontecem a partir dos estados *DT_STATE1* e *DT_STATE2* da máquina de estados da detecção são decididas, a nível de hardware, no estado **VERIFICATION** da máquina de estados do hardware.

SET_TS

Para o processamento de cada canal, é necessário copiar sua região na memória para o buffer interno. Com a cópia da memória efetua-se os devidos cálculos, incrementos e etc, mas todas as alterações são escritas no buffer interno, e não na memória do *Tone Detector*.

A escrita das alterações da região de memória, para cada canal, é efetuada no estado **SET_TS**. É escrita uma *WORD* de dezesseis bits em cada ciclo de clock, totalizando doze períodos de clock para a escrita de toda a região de cada canal de detecção.

SEND_MESSAGE

O estado **SEND_MESSAGE** é responsável pelo envio das mensagens a nível de hardware, sejam mensagens de sucesso da detecção ou de ocorrência de *timeout*. Em ambos os casos, após o envio da mensagem, a região de memória correspondente ao canal será liberada, para que esteja disponível para um novo processamento.

Para a entrega da mensagem só é necessário verificar se ainda há espaço disponível no buffer de mensagens, esta verificação é feita através do sinal *MSG_BUF_FULL*, sinal este que tem valor lógico baixo quando o buffer ainda tem espaço. Este sinal vem diretamente do *Message Handler*, que é o bloco de hardware responsável pelo envio das mensagens do *Tone Detector*. Esta entrega da mensagem é efetuada em um único ciclo de clock, caso houver espaço, pois o *Message Handler* está sempre esperando pelo recebimento de mensagens, isto faz com que não seja perdido tempo de processamento com a atividade de envio de mensagens. O bloco de hardware *Message Handler* será detalhado na Seção 3.4.7.

NEXT_PORT

A máquina de estados do hardware tem seu ciclo de execução para cada canal de detecção, durante este ciclo, ações como cálculos, leituras e escritas na memória, verificações e envio de mensagens podem ser feitas. Depois do processamento de um canal é iniciado o processamento do próximo, se houver outro canal para ser processado. A mudança do canal a ser processado é feita pelo estado **NEXT_PORT**.

Se houver um próximo canal a ser processado, o valor do sinal interno *port_proc_s*, que guarda o valor do canal de processamento atual, é incrementado e o próximo estado é **GET_TS**, para a leitura da memória deste canal.

Não havendo mais canais a serem processados, a máquina avança para o estado **HALT_PROCESSOR**, fazendo com que o processador fique no aguardo por novas amostras de áudio para o processamento.

HALT_PROCESSOR

Devido a frequência de amostragem dos áudios serem $8KHz$, os sinais em detecção tem novas amostras de áudio disponível para processamento a cada $125\mu s$. Com a frequência do clock do processador a $80MHz$, todos os canais são processados muito antes que as novas amostras estejam disponíveis. Neste tempo restante, o *Signal Processor* interrompe seu processamento e libera a memória para que o bloco de hardware *Command Handler*, Seção 3.4.3, tenha a memória disponível para que assim possa escrever parâmetros para um novo processamento.

A máquina de estados do hardware permanece em **HALT_PROCESSOR** até que novas amostras de áudio estejam disponíveis para o processamento, com isto, a máquina vai para o estado **S0**, e assim reinicia todo o ciclo de processamento.

3.4.7 Tratamento das Mensagens

A última interface do *Tone Detector* é a de mensagens, estas representam as respostas aos processamentos iniciados através do *Command Handler*, Seção 3.4.3. No bloco de hardware *Message Handler*, Figura 15, está todo o processo de recebimento das mensagens do *Signal Processor* e o envio destas através da interface de mensagens, ambas ações realizadas por duas máquinas de estados que são, respectivamente, a máquina de estados de recepção e a máquina de estados de transmissão de mensagens.

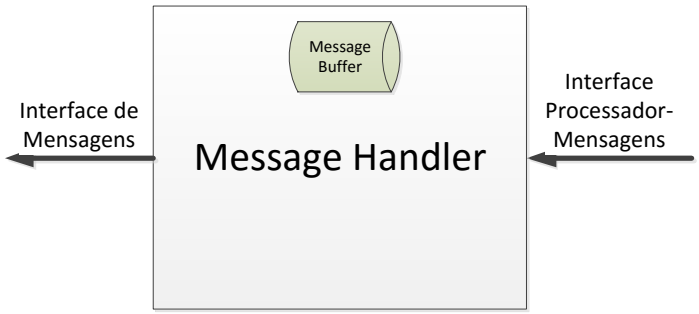


Figura 15 – Message Handler Block

As duas máquinas de estados fazem uso de um recurso comum, um buffer circular do tipo FIFO, para armazenar as mensagens da detecção, para que as mesma possam ser enviadas via interface de mensagem. O acesso ao buffer circular é feito através de dois ponteiros, que são os sinais internos *buf_tail_s* e *buf_head_s*, que respectivamente apontam para o para final e início da fila. As duas máquinas de estados serão explicitadas nos itens, a seguir.

Rx State Machine

A máquina de estado de recepção de mensagens é constituída por cinco estados, que são: **S0**; **WAIT_RX**; **RECEIVE_DATA**; **INC_TAIL**; **WAIT_FREE_SPACE**. O diagrama de máquina de estados é mostrado na Figura 16.

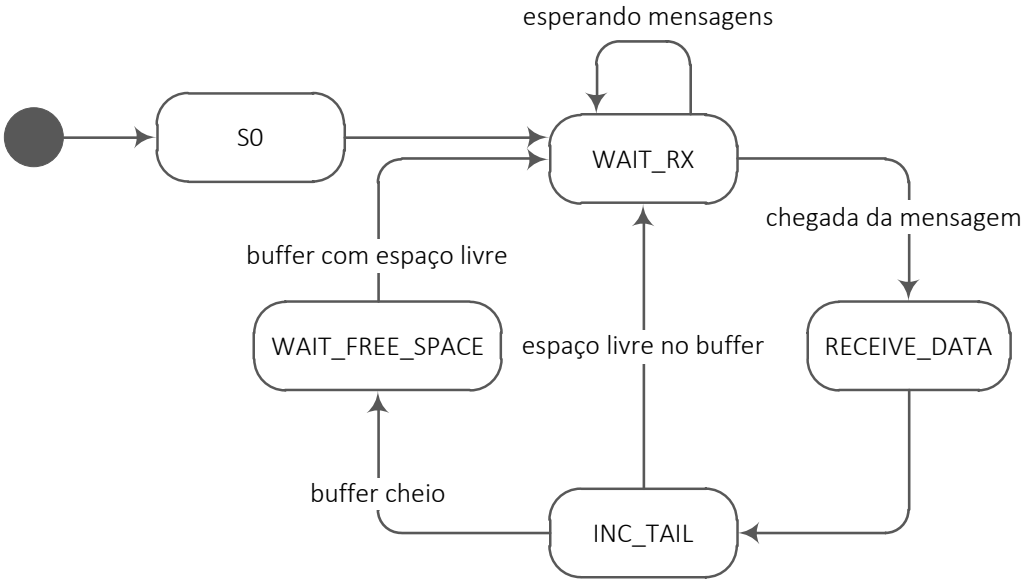


Figura 16 – Rx Message State Machine

No estado inicial, **S0**, são efetuado os resets dos sinais internos relativos a recepção das mensagens, assim como dos campos do buffer interno de mensagens. Após o período de reset, a máquina avança para o estado **WAIT_RX**, e permanece deste modo enquanto o *Signal Processor* não enviar nenhuma mensagem.

Com a chegada da mensagem vinda do *Signal Processor*, a máquina avança para o estado **RECEIVE_DATA**, que salva o conteúdo da mensagem no campo do buffer interno apontado pelo sinal interno *buf_tail_s*, e após isso, a máquina avança para o estado **INC_TAIL**, neste estado, é efetuado o incremento ponteiro *buf_tail_s*. Por se tratar de um buffer circular, um incremento para quando *buf_tail_s* estiver apontando para o final do buffer implica que ele irá apontar para o início do buffer. A forma de incremento do ponteiro está apresentado em 3.9.

$$buf\_tail\_s = \begin{cases} buf\_tail\_s + 1, & \text{se} \quad buf\_tail\_s < MSG\_BUFFER\_SIZE, \\ 0, & \text{alhures.} \end{cases} \tag{3.9}$$

Após incremento, a máquina tem duas possibilidades de avanço de estado, retornar para **WAIT_RX** se houver espaço no buffer, e assim ficar na espera por novas mensagens, ou ir para o estado **WAIT_FREE_SPACE** se não houver mais espaço disponível para armazenar mensagens no buffer interno. A máquina de estados de recepção de mensagens sabe que não tem mais espaço no buffer interno quando, após o incremento do ponteiro *buf_tail_s*, seu valor seja igual ao ponteiro *buf_head_s*, que é o ponteiro que aponta para o início da fila.

Chegando ao estado **WAIT_FREE_SPACE**, permanecerá até que *buf_tail_s* e *buf_head_s* tenham valores diferentes. Enquanto estiver nesse estado, o sinal *MSG_BUF_FULL* permanecerá em nível lógico alto, informando para o *Signal Processor* que não há espaço para receber mensagens. Ao ser liberado espaço no buffer interno, a máquina retorna ao estado **WAIT_RX**.

Tx State Machine

O último passo do processamento de um áudio é transmitir a informação de sucesso, ou não, da detecção para o módulo que estiver controlando o Tone Detector. Se houver mensagem a ser transmitida, esta é enviada ao controlador externo via protocolo *handshake*. A máquina de estados de transmissão de mensagens possue quatro estados, que são: **S0**; **WAIT_MSG**; **SEND_TX**; **INC_HEAD**. O diagrama de máquina de estados está apresentado na Figura 17.

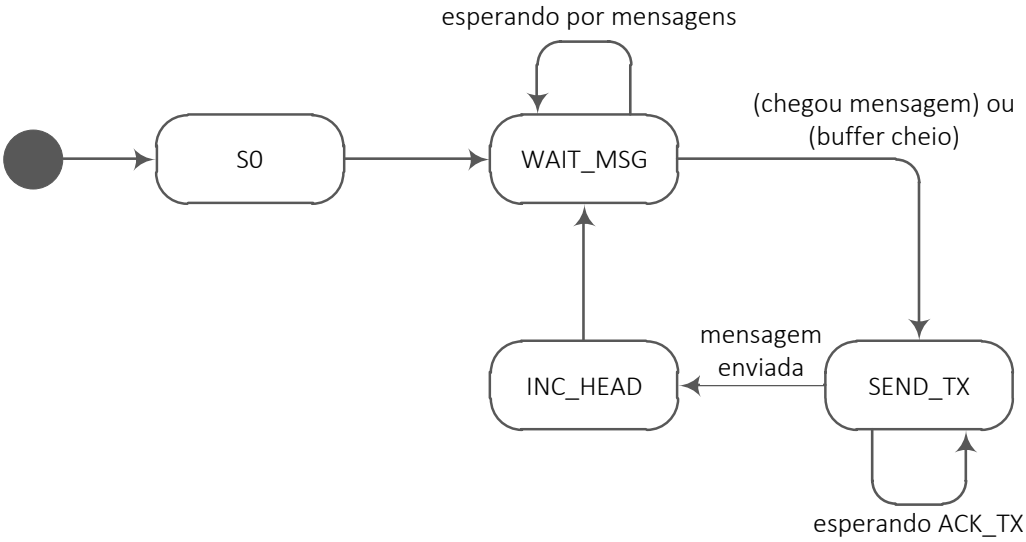


Figura 17 – Tx Message State Machine

O primeiro estado, assim como em todas as máquinas de estados deste trabalho, é o **S0**, o qual faz o reset dos sinais internos relativos a transmissão de mensagens do módulo *Message Handler*. Após período de reset, a máquina avança para o estado **WAIT_MSG**, onde permanece até que tenha alguma mensagem no buffer interno para envio, isto acontece quando *buf_tail_s* e *buf_head_s* possuem valores diferentes, ou quando o sinal *MSG_BUF_FULL* tiver valor lógico alto.

Com a existência de alguma mensagem para ser enviada, a máquina avança para o estado **SEND_TX**. Nesse estado, é enviado um sinal de *RX* ao controlador externo e a mensagem fica disponível para ser lida através do sinal *MSG_TX_DATA*. A máquina permanece nesse estado até receber uma resposta através do sinal *ACK_TX*, informando que o conteúdo da mensagem foi lido.

Com a mensagem lida, a máquina avança para o estado **INC_HEAD**, onde o valor do ponteiro *buf_head_s* é incrementado. A forma de incremento deste ponteiro segue o mesmo protocolo de incremento para o ponteiro *buf_tail_s*, como pode ser visto a seguir em 3.10.

$$buf\_header\_s = \begin{cases} buf\_header\_s + 1, & \text{se} \quad buf\_header\_s < MSG\_BUFFER\_SIZE, \\ 0, & \text{alhures.} \end{cases} \tag{3.10}$$

Com o incremento do ponteiro concluído, a máquina de estados da transmissão de mensagens retorna ao estado **WAIT_MSG**, assim ficando disponível para envio de novas mensagens.

3.4.8 Unidade de Testes

Com o desenvolvimento do *Tone Detector* finalizado, foi necessário criar uma unidade de testes para validar a funcionalidade do hardware proposto. A unidade de testes também está desenvolvida na linguagem VHDL, mas diferente do projeto do *Tone Detector*, utiliza funcionalidades que não são sintetizáveis, como leitura de arquivos. A unidade de testes aqui está descrita de maneira mais funcional, e portanto mais simplificada.

A unidade de teste é composta por três módulos controlados, *frameInserter*, *commandInserter* e *messageReceive*, que respectivamente são responsáveis pela inserção dos comandos, inserção dos áudios, recepção das mensagens, e um módulo controlador, *testControl*, que gerencia os três módulos controlados. A Figura 18 apresenta como a unidade de testes se conecta ao *Tone Detector*.

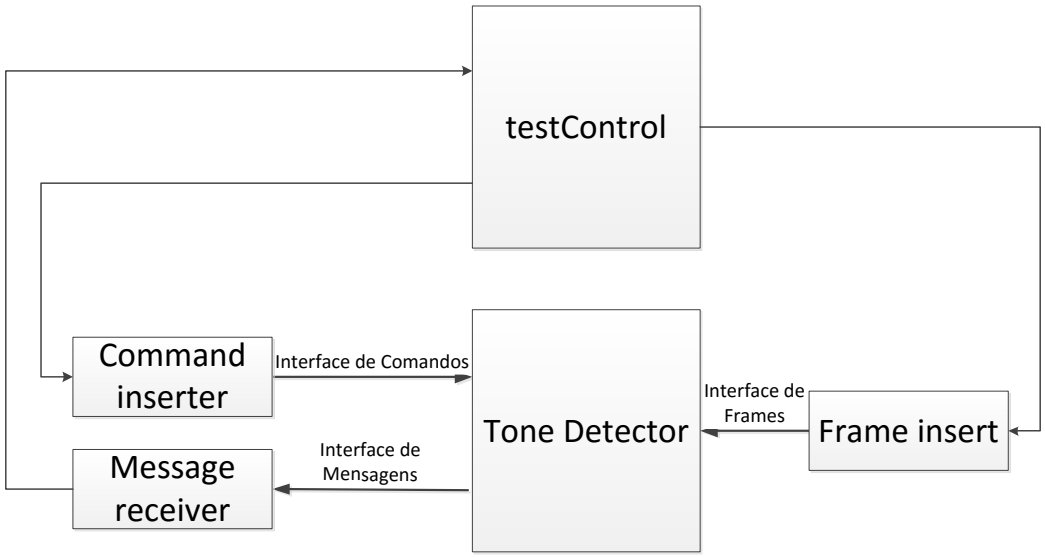


Figura 18 – Test Unity

A unidade de testes tem ciclo de execução orientado para os canais de processamento, ou seja, sua funcionalidade tem como base fazer com que os canais estejam sempre processando, enquanto houver áudios a serem processados.

Os áudios e os comando são gerados com uso de um programa desenvolvido em *C ANSI*, o qual gera sinais com frequências, durações de pulsos, durações de pausas e valor de cadência com valores aleatórios, baseados nesses valores são calculados os parâmetros para cada áudio gerados. Este programa tem como saída os arquivos de áudios e o arquivo "fileParams.bin", que é um arquivos que contém os comandos para cada áudio gerado. Os valores das frequências estão dentro do intervalo $[400Hz, 800Hz]$. Os valores possíveis para as largura de pulsos e pausas estão no intervalo $[120ms, 670ms]$. Dentre os áudios gerados, 80% não tem cadências, ou seja, o sinal não tem repetições, em 10%, a cadência tem valor um e nos outros 10% a o sinal gerado tem cadência dois.

O modelo funcional da unidade de testes está apresentado na Figura 19, e nela estão apresentado os passos que são seguidos pelo módulo controlador, por canal, para a simulação dos processamentos.

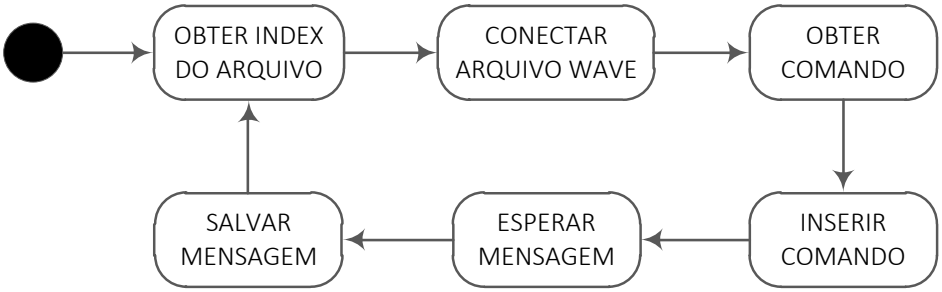


Figura 19 – Test Unity Functional Model

A primeira etapa dos testes, *GET FILE INDEX*, consiste em saber quais canais estão disponíveis para processamento, para isso, existe um array de 8 bits, para qual, cada bit representa se a porta está em uso, ou não, nível lógico alto significa canal disponível, e nível lógico baixo significa canal ocupado.

Com a obtenção da informação de qual arquivo será utilizado, inicia-se a segunda etapa dos testes, a conexão do arquivo ao *Tone Detector*, através do módulo *frameInserter*, que faz a leitura de um arquivos no formato ".wav", descartando apenas seu cabeçalho, e inserindo frames a uma frequência de $8Khz$.

Depois de realizada a conexão do arquivo de áudio ao *Tone Detector*, é utilizado seu index para a leitura do comando dentro do arquivo "fileParams.bin", que contém os parâmetros para cada sinal gerado. Depois de lidos os parâmetros do áudio, é inserido o comando através do módulo *commandInserter*. Esta são, respectivamente as etapas três e quatro.

Após a etapa quatro, ou seja, a inserção do comando relativo ao áudio no canal, é dado início ao processamento. A tarefa da unidade de testes, para este canal, é ficar no aguardo pela mensagem da detecção, que consiste na etapa cinco. Com a chegada desta mensagem, seu conteúdo é salvo em um arquivo juntamente com a informação de index do arquivo e o valor da canal de processamento. Estas ações correspondem, respectivamente às etapas cinco e seis da unidade de testes.

Os processos realizados pela unidade de testes, verificação de canal livre, conexão de áudio, inserção dos comandos e recebimento das mensagens, são executados para cada canal de detecção, e este ciclo é repetido até que todos os áudios gerados sejam processados.

4 RESULTADOS

Neste capítulo serão apresentados três resultados, o primeiro é o da ferramenta de síntese, que contém as informações dos elementos lógicos utilizados, o segundo é uma análise dos tempos de processamento, e por último os resultados das simulações baseado nas mensagens da detecção. Para análise temporal do *Tone Detector*, foram medidos os tempos necessários para os parâmetros da detecção serem escritos pelo *Command Handler*, e os tempos de processamentos dos canais, assumindo o pior de caso de execução.

Para o nosso melhor conhecimento, na literatura não foram encontrados trabalhos semelhantes, que poderiam ser utilizados para efeito de comparação com os resultados da síntese do trabalho proposto.

4.1 Resultados da Síntese

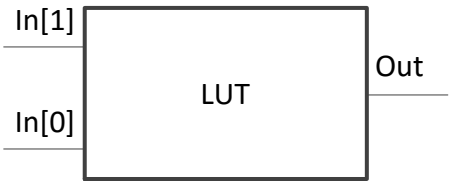
Para a síntese foi utilizada a ferramenta proprietária da *Xilinx*, *ISE* versão 14.7, tendo como hardware alvo, a FPGA modelo *6vlx240tff784-1*. A ferramenta de síntese faz a leitura do código da descrição do hardware, e infere estruturas de hardware correspondentes.

A síntese atingiu para a frequência de clock um valor máximo de $88.402MHz$, resultado esse que para o hardware proposto é suficiente, pois o clock do projeto tem frequência igual $80MHz$. A Tabela 1 mostra os elementos de hardware inferidos pela ferramenta de síntese.

Tabela 1 – Elementos de hardwares inferidos na síntese

Logic elements	**Amount**
MACs	3
Multipliers	8
Adders	14
Subtractors	6
Counters	1
Registers	2504
Comparators	22
Multiplexers	490

Os blocos lógico são constituídos por slices, estes são formados por agrupamentos de **L**ook **U**p **T**ables (**LUT**) e Flip Flops. LUTs são elementos lógicos que fazem a relação de um sinal de saída com um conjunto de entradas, por exemplo, a figura 20, mostra os uso de uma LUT para inferir uma porta lógica *AND*, em que a saída só será verdadeira quando as duas entradas forem verdadeira. A ferramenta de síntese também dá um sumário da utilização de slices da FPGA, conforme pode ser observado na Tabela 2.



In[1:0]	Out
00	0
01	0
10	0
11	1

Figura 20 – LUT Example

Tabela 2 – Sumário de utilização da FPGA

Utilização de Slices Lógicos

	Usado	Disponível	Utilização
Quantidade de Registradores Slices	2173	301440	0,72%
Quantidade de LUTs Slice	1645	150720	1,09%

Distribuição dos Slices Lógicos

	Usado	Disponível	Utilização
Número de Flip Flops não utilizados	1308	3481	37,58%
Número de LUTS não utulizadas	1836	3481	52,74%
Número de pares LUT-FF utilizados	337	3481	9,68%

Em (BHAVANAM P. SIDDAIAH, 2014a) e (BHAVANAM P. SIDDAIAH, 2014b) são sintetizados detectores DTMF. Nestes trabalhos são informados dados como área e potência do hardware sintetizado.

4.2 Análise dos tempos de processamento

Para o *Tone Detector* existem dois tempos que são cruciais para o processamento, o primeiro é o tempo que o *Command Handler* leva para receber o comando e escrevê-lo na memória, e o segundo é o intervalo de tempo em que o *Signal Processor* leva para processar um canal.

Depois de o comando ser recebido, o *Command Handler* faz a requisição para escrever na memória os parâmetros da detecção. Com a autorização para que a escrita seja feita, leva-se dez ciclos de clock para que a ação seja concluída, para o recebimento deste comando leva-se também dez ciclos, totalizando vinte ciclos de clock para o recebimento e escrita na memória, do comando. Para o *Signal Processor*, o pior de caso de execução, ou seja, a situação em que leva-se mais tempo para processar um canal, gasta-se quarenta e cinco ciclos de clock, sendo vinte e seis ciclos para ler a região de memória do canal, cinco ciclos de processamento,

treze para a escrita na memória e um para a mudança para o próximo canal.

Como a frequência do clock é $80MHz$, o perído é $12,5ns$, assim temos que no total, o *Command Handler* gasta $250ns$ para escrever os parâmetros na memória. O *Signal Processor* leva $562,5ns$ por canal, no pior caso de execução. Como a taxa de atualização dos frames é $8KHz$, ou seja, o *Tone Detector* tem uma janela de processamento de $125\mu s$ para processar todos os canais.

Com as informações de tempo de escrita dos parâmetros na memória, e dos tempos de processamento do *Signal Processor*, podemos extrair a informação de quantidade máxima de canais possíveis de serem processados, sobre duas situações distintas: Não havendo escrita de comandos na memória; Escrita de vinte comandos na memória. Em ambas as situações, todos os canais estão processando, e no pior caso de execução.

Na primeira situação, temos que a janela de processamento está totalmente disponível para o *Signal Processor*, ou seja, a quantidade máxima de canais possíveis de processar é:

$$n_{max} = \frac{\textit{Janela de processamento}}{\textit{Tempo de processamento}} = \frac{125\mu s}{562,5ns} = 222 \textit{ canais} \tag{4.1}$$

Na segunda situação, com a escrita de vinte comandos na memória, temos:

$$n_{max} = \frac{\textit{(Janela de processamento)} - 20.\textit{(Tempo de escrita do comando)}}{\textit{Tempo de processamento}} \tag{4.2}$$

$$= \frac{125\mu s - 20.250ns}{562,5ns} \tag{4.3}$$

$$= 213 \textit{ canais} \tag{4.4}$$

De maneira geral, o número máximo de canais possíveis de serem processados é:

$$n_{max} = \frac{\textit{(Janela de processamento)} - n_{comandos}.\textit{(Tempo de escrita por comando)}}{\textit{Tempo de processamento}} \tag{4.5}$$

$$= \frac{125\mu s - n_{comandos}.250ns}{562,5ns} \tag{4.6}$$

Os valores de quantidade máxima de canais possíveis de serem processados se mostram com valores ótimos, tendo em vista a baixa frequência de clock, o que influi diretamente na potência utilizada pelo *Tone Detector*. Tomando como base o DSP da *Texas Instrument* modelo *tms320vc5509a*, que em sua máxima performance, executa uma instrução a cada $5ns$, implicando que para igualar a performance, este DSP teria que fazer todo o processamento em no máximo cento e doze instruções, levando em conta acessos a memória, multiplicações, cálculo do algoritmo de *Goertzel*, etc. O cálculo da quantidade de instruções necessárias é feito baseado no tempo gasto pelo *Signal Processor* para processar um canal, que é $562,5ns$.

O poder de processamento de *Signal Processor* frente ao DSP *tms320vc5509a*, vem do fato de o *Signal Processor* ser projetado para uso específico nesse tipo de processamento, além de ser implementado em hardware, que tráz a vantagem de a maior parte das operações serem efetuadas em paralelo.

4.2.1 Comparação de desempenho temporal

Para métrica de comparação, foi utilizado o *Aplication Note*, da *Silicon Labs*, (SILICON LABS,), no qual foram realizados testes de desempenho em três testes: Projetos de filtros FIR; Algoritmos de Goertzel para detecção DTMF; Algoritmo da FFT. Para comparação usamos os resultados obtidos da utilização do algoritmo de Goertzel para a detecção DTMF.

Sinais DTMF(*Dual-Tone Multi-Frequency*), são sinais compostos pela soma de sinais com baixa frequência e outro com alta frequência, como pode ser visto na Figura 21. Na telefonia são utilizados para sinalizar os dígitos do teclado numérico.

		Alta Frequência			
		1209 Hz	1336 Hz	1477 Hz	1633 Hz
Baixa Frequência	697 Hz	1	2	3	A
	770 Hz	4	5	6	B
	852 Hz	7	8	9	C
	941 Hz	E	0	F	D

Figura 21 – Dígitos DTMF

Na detecção de dígitos DTMF implementada em (SILICON LABS,), assim como nesse trabalho, o algoritmo de Goertzel foi utilizado, mas na detecção de dígitos DTMF, existem oito frequências que têm que ser verificadas, diferente da detecção deste trabalho, que processa duas frequências para cada frame de áudio.

No trabalho da *Silicon Labs*, (SILICON LABS,), foi utilizado um DSP da família C8051F12x, que tem uma frequência máxima de $25MHz$. São apresentados resultados para o processamento das oito frequências presentes na detecção DTMF em diferentes níveis de desempenho do DSP, escolhendo o caso de maior desempenho, que é na utilização de instruções MAC com implementação de noventa e oito MIPS(milhões de instruções por segundo). A comparação foi realizada somente sob o segmento de código responsável pelo processamento.

Para comparação direta com os resultados do documento referenciado, fizemos uma redução no clock do *Tone Detector* para $25MHz$, assim tendo como período $40ns$, e para o processamento das oito frequências, faremos a utilização de quatro canais. No *Tone Detector* são gastos 42 ciclos de clock para o cálculo do primeiro passo do algoritmo de Goertzel, incluindo leitura e escrita na memória, e três ciclos para o cálculo da segunda etapa, a potência do sinal, que inclui a fase de verificação dos resultados também, portanto, são quarenta e cinco

ciclos para processar dois frequências, assim, para oito temos um total de cento e sessenta e oito ciclos para o cálculo de Goertzel e doze para o cálculo da da potência. Os resultados do cálculo do algoritmo de Goertzel, dos dois trabalhos encontram-se na Tabela 3.

Tabela 3 – Comparação dos resultados

	C8051F12x		**Tone Detector**		**% de Redução**	
	Ciclos de clock	μs	Ciclos de clock	μs	Ciclos de Clock	Tempo
Gortzel	1018	10,4	168	6,72	83,49%	35,39%
Potência	1743	17,8	12	0,48	99,31%	97,30%
Goertzel + Potência	2761	28,2	180	7,2	93,48%	74,47%
Tempo total para 200 samples de entrada	205000	2095	33612	1344	83,60%	35,85%

Analisando a Tabela 3 podemos constatar que o *Tone Detector* obteve grande desempenho frente à implementação referenciada. Para um processamento com duzentas amostras por frame, obtivemos uma grande redução nos ciclos de clock necessários para realizar a mesma tarefa, tendo uma redução de $83,60\%$, alcançando uma redução de $751\mu s$, equivalendo a $35,85\%$ em relação à aplicação da *Silicon Labs*.

4.3 Resultado das simulações

As simulações foram realizadas, através do bloco de unidade de testes, descrito na Seção 3.4.8, para isto, são inseridos, por canal, o comando contendo os parâmetros e o áudio relativo a este comando. O resultado das simulações é obtido através da análise das mensagens que o *Tone Detector* retorna, são três tipos, mensagens de detecção, *timeout* de pulsos e *timeourt* de pausas. Para a análise das mensagens, foram gerados áudios com essas três características.

Do universo de áudios gerados, na média 50% correspondem a parcela de sinais que foram gerados com *timeout*, sendo 25% de pulsos e 25% de pausas. Os 50% restantes são sinais em que suas características estão descritas de acordo com os parâmetros passados.

Para fazer a análise foram gerados dois mil áudios, dentre estes novecentos e noventa e quatro correspondem a áudios corretos, quinhentos e doze são áudios com *timeout* de pulso e quatrocentos e noventa e quatro são correspondem a *timeout* de pausa. Para fazer a simulação do processamento de todos os áudios, levou-se $25h20m$, isto para simular um tempo de $6m40s$.

O resultado da análise da simulação do *Tone Detector* nos mostra que foi obtido uma taxa de acerto de 100%, isto é, todos os áudios sem *timeout* foram detectados corretamente, assim como todos o áudios com *timeout* de pulso ou de pausa tiveram esta falha detectada. Essa

taxa de acerto devido ao fato de que o processamento do *Tone Detector* não é probabilístico, ou seja, quando um áudio tem seu processamento iniciado só existe como possibilidade, este ser detectado, ou não.

5 CONCLUSÃO

Neste trabalho foi desenvolvido um core de detecção de sinais telefônicos monotônicos. A plataforma alvo de desenvolvimento é um hardware configurável FPGA, modelo virtex 6 fabricado pela *Xilinx*. O hardware proposto tem duas interfaces de controle, que são a de recebimento de comandos e de envio de mensagens, e uma interface de injeção de áudio, que consiste em oito canais de recepção multiplexados no tempo. Internamente, está implementado um processador de sinais, com arquitetura 16 bits, responsável pelo processamento matemático dos áudios.

Foi utilizado neste trabalho o algoritmo de *Goertzel*, que é a base teórica fundamental para o processamento dos sinais. Este algoritmo foi implementado em hardware, possibilitando um aumento significativo de sua velocidade, pois a maior parte das operações são efetuadas em paralelo.

O desenvolvimento foi realizado utilizando a metodologia *top-down*, assim possibilitando uma maior organização do trabalho. A sequência de desenvolvimento seguiu o fluxo de utilização do hardware proposto, ou seja, iniciou-se pela interface de comandos, por onde passa a informação do áudio que será detectado, seguido pelo desenvolvimento da interface de recebimento de áudio. Tendo os parâmetros e o áudio a ser detectado, deu-se início ao desenvolvimento do detector de sinais, posteriormente sendo feito o desenvolvimento da interface de mensagens, por fim retorna o resultado da detecção.

Neste trabalho foram obtidos três resultados, primeiro da ferramenta de síntese, segundo dos resultados temporais, e o terceiro baseados nas mensagens de respostas. Da ferramenta de síntese obtemos a quantidade de elementos lógicos e slices utilizados, não foi possível fazer uma comparação destes resultados pois não foi encontrado na literatura métricas de comparação. Os resultados temporais mostraram que dentro da janela de processamento, o hardware proposto consegue processar muitos canais além dos oito canais propostos, com uma taxa de acerto de 100%, chegando esse número até duzentos e vinte e dois canais. Da análise das mensagens de resposta pode ser verificado que todos os resultados esperados aconteceram, isso acontece pelo fato de que o processamento é realizado de maneira determinística, só havendo a possibilidade de o áudio ser detectado, ou não.

5.1 Trabalhos Futuros

Como trabalho futuro temos três metas, embarcar o código na FPGA e iniciar a fase de testes no hardware. A segunda meta é ampliar a faixa de frequência em que a detecção acontece, mas para isso teremos que voltar a fase de pré-projeto, pois para essa ampliação terão que ser aumentados a largura, em bits, de alguns sinais internos, influenciando em desempenho, pois esta ação irá influenciar diretamente nos cálculos matemáticos. A terceira meta é injetar

áudios com ruídos, tornando a análise mais próxima da realidade em que os áudios estão submetidos no meio de comunicação.

REFERÊNCIAS

ALI, K. S. Digital circuit design using fpgas. *Computers and Industrial Engineering*, p. 127 – 129, 1996.

BHAVANAM P. SIDDAIAH, P. R. R. S. N. Fpga based efficient dtmf detection using split goertzel algorithm with optimized resource sharing approach. *Wireless and Optical Communications Networks (WOCN), 2014 Eleventh International Conference on*, September 2014.

BHAVANAM P. SIDDAIAH, P. R. R. S. N. Zynq 7000 series fpga based efficient dtmf detection. *Computational Intelligence and Computing Research (ICCIC), 2014 IEEE International Conference on*, p. 1 – 7, December 2014.

COSTA, C. d.; MESQUITA, L.; PINHEIRO, E. *Elementos de Lógica Programável com VHDL e DSP: Teoria e Prática*. São Paulo, Brasil: Érica, 2011. ISBN 978-85-365-0312-7.

LINDH JOHAN STäNER, J. A. L. Experiences with vhdl and fpgas. *Journal of Systems Architecture*, p. 97 – 104, 1996.

OPPENHEIM, A. V. *Discrete-Time Signal Processing*. Upper Saddle River, New Jersey 07458: Prentice Hall, 1998. ISBN 0-13-754920-2.

PEDRONI, V. A. *Eletrônica Digital Moderna e VHDL*. Rio de Janeiro, Brasil: Elsevier, 2010. ISBN 978-85-352-3465-7.

SELMAN, R. P. S. Comparative analysis of methods used in the design of dtmf tone detectors. *IEEE International Conference on Telecommunications and Malaysia International Conference on Communications*, May 2007.

SILICON LABS. *Using Microcontrollers in Digital Signal Processing Applications*. Ug364 (v1.2). [S.l.]. Disponível em: <https://www.silabs.com/Support%20Documents/TechnicalDocs/an219.pdf>.

XILINX. *EDK Concepts, Tools, and Techniques: A Hands-On Guide to Effective Embedded System Design*. Ug683 (v13.4). [S.l.], 2012.

XILINX. *Embedded System Tools Reference Manual: EDK*. Ug111 (v13.4). [S.l.], 2012.

XILINX. *Virtex-6 Family Overview: Product Specification*. Ds150 (v2.4). [S.l.], 2012.

XILINX. *Virtex-6 FPGA Configurable Logic Block: User Guide*. Ug364 (v1.2). [S.l.], 2012.

YUYING, L. Research of dtmf dialing system based on the goertzel algorithm and matlab simulation. *Information Technology and Artificial Intelligence Conference (ITAIC), 2014 IEEE 7th Joint International*, p. 93 – 97, December 2014.

ZáPLATA, M. K. F. Using the goertzel algorithm as a filter. *Radioelektronika (RADIOELEKTRONIKA), 2014 24th International Conference*, p. 1 – 3, April 2014.