Universidade Federal de Juiz de Fora

Programa de Pós-Graduação em

Engenharia Elétrica

Eder Barboza Kapisch

DETECÇÃO E COMPRESSÃO DE DISTÚRBIOS ELÉTRICOS BASEADAS EM PLATAFORMA FPGA

Juiz de Fora

2015

Universidade Federal de Juiz de Fora
Programa de Pós-Graduação em
Engenharia Elétrica

Eder Barboza Kapisch

DETECÇÃO E COMPRESSÃO DE DISTÚRBIOS ELÉTRICOS BASEADAS EM PLATAFORMA FPGA

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Juiz de Fora, área de concentração: Sistemas Eletrônicos, da Faculdade de Engenharia da Universidade Federal de Juiz de Fora como requisito parcial para obtenção do grau de Mestre.

Orientador: Prof. Carlos Augusto Duque, D.Sc.

Coorientador: Prof. Luciano Manhães de Andrade Filho, D.Sc.

Juiz de Fora

2015

Ficha catalográfica elaborada através do programa de geração
automática da Biblioteca Universitária da UFJF,
com os dados fornecidos pelo(a) autor(a)

Kapisch, Eder Barboza.
Detecção e Compressão de Distúrbios Elétricos Baseadas em Plataforma FPGA / Eder Barboza Kapisch. -- 2015.
216 f. : il.

Orientador: Carlos Augusto Duque
Coorientador: Luciano Manhães de Andrade Filho
Dissertação (mestrado acadêmico) - Universidade Federal de Juiz de Fora, Faculdade de Engenharia. Programa de PósGraduação em Engenharia Elétrica, 2015.

1. Compressão de dados. 2. Detecção de distúrbios elétricos. 3. Transformada Wavelet. 4. Processador embarcado. 5. FPGA. I. Duque, Carlos Augusto, orient. II. Filho, Luciano Manhães de Andrade, coorient. III. Título.

Eder Barboza Kapisch

DETECÇÃO E COMPRESSÃO DE DISTÚRBIOS ELÉTRICOS BASEADAS EM PLATAFORMA FPGA

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Juiz de Fora, área de concentração: Sistemas Eletrônicos, da Faculdade de Engenharia da Universidade Federal de Juiz de Fora como requisito parcial para obtenção do grau de Mestre.

Aprovada em 22 de janeiro de 2015

BANCA EXAMINADORA

Prof. Carlos Augusto Duque, D. Sc.
Universidade Federal de Juiz de Fora, UFJF
Orientador

Prof. Luciano Manhães de Andrade Filho, D. Sc.
Universidade Federal de Juiz de Fora, UFJF
Orientador

Rafael Antunes Nóbrega, Ph. D.
Universidade Federal de Juiz de Fora, UFJF

José Gabriel Rodriguez Carneiro Gomes, Ph. D.
Universidade Federal do Rio de Janeiro, UFRJ

AGRADECIMENTOS

Agradeço, em primeiro lugar, a Deus, por ter me dado vida e forças para concluir este trabalho.

À minha querida esposa, Laíse, pelo amor e por sempre me ajudar nas minhas atividades acadêmicas. Por ser paciente e fornecer-me apoio constante e incondicional.

Aos meus pais, Ademir e Raquel, meus irmãos Ademir Júnior e Suélen, pelo carinho e por sempre demostrarem preocupação e interesse no meu bem-estar.

A todos os meus familiares, que de alguma forma me apoiaram.

Aos amigos do LAPTEL e PSCOPE, Leandro Manso, Carlos Henrique, Henrique Monteiro, Alexander Barbosa, pela ajuda nas implementações, dúvidas, conselhos, formatação e pela amizade.

Agradeço aos meus professores e orientadores Carlos Augusto Duque e Luciano Manhães de Andrade Filho, pelos ensinamentos, amizade e confiança depositada em mim durante todo esse tempo.

Ao CNPQ, à Universidade Federal de Juiz de Fora, à Faculdade de Engenharia, ao PPEE e à KRON, por todo o suporte dado para o desenvolvimento deste trabalho.

“É quando nos esquecemos de nós mesmos que fazemos coisas que jamais serão esquecidas.”

Ellen G. White

RESUMO

A presente dissertação apresenta a implementação de um Sistema de Detecção e Compressão de Distúrbios Elétricos (SDCDE), com foco nas implementações baseadas em plataforma FPGA (*Field-Programmable Gate Array*). Inicialmente são abordados os algoritmos de compressão e detecção. Posteriormente são mostradas as sínteses na FPGA e um protótipo desenvolvido para testes. O sistema proposto é voltado para aplicações em Sistemas Elétricos de Potência (SEPs) e prevê a aquisição e o armazenamento dos distúrbios comumente encontrados nesse campo. A partir dos dados armazenados, é possível reconstruir inteiramente o sinal registrado, para possíveis análises de oscilográfia. O processo de compressão passa por três estágios: detecção de novidade, compressão com perdas, utilizando a Transformada *Wavelet* Discreta (DWT), e a Compressão em termos de bit. Esses três níveis de compressão permitem uma otimização do espaço de memória utilizado e garantem que longos períodos de registros possam ser armazenados em um cartão de memória. A abordagem das sínteses em FPGA visa avaliar, dentre outros fatores, o consumo de recursos de *hardware* utilizado, através da implementação de um processador embarcado, criado e idealizado para aplicações de Processamento Digital de Sinais (DSP). A partir do protótipo desenvolvido, alguns resultados de sínteses e estudos de casos com testes executados em ambientes reais, são apresentados.

Palavras-chaves: Compressão de dados, Detecção de distúrbios elétricos, Transformada *Wavelet*, Processador embarcado, FPGA.

ABSTRACT

This dissertation presents the implementation of a System of Detection and Compression of Electrical Disturbances (SDCDE), focusing on implementations based on FPGA platform (Field-Programmable Gate Array). Initially are discussed compression and detection algorithms. Subsequently the synthesis in FPGA and a prototype that was developed for testing are shown. The proposed system is aimed at applications in Electric Power Systems (SEPs) and provides for the acquisition and storage of the disturbances commonly found in this field. From the data stored, the recorded signal can be fully reconstructed for possible oscillographic analysis. The compression process involves three stages: novelty detection, lossy compression, using the Discrete Wavelet Transform (DWT), and a bit-level compression. These three levels of compression allow an optimization of used memory space and they ensure that long periods of records can be stored on a memory card. The approach of the synthesis on FPGA aims to evaluate, among other factors, the usage of hardware resources, through the implementation of an embedded processor, created and designed for digital signal processing applications. From the prototype developed, some results of synthesis and case studies with tests performed in real environments are presented.

Keywords: Data Compression, Detection of electrical disturbances, Wavelet Transform, Embedded processor, FPGA

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

Atualmente, existe uma grande preocupação envolvendo problemas de qualidade de energia elétrica, proveniente tanto dos agentes profissionais do mercado de energia, que prezam pela qualidade de seu produto, quanto dos pesquisadores, que se empenham em desenvolver novas técnicas e equipamentos a fim de contornar os problemas (Bukata & Li, 2011), (Mazumdar, Harley, Lambert, & Venayagamoorthy, 2005). Tais problemas surgem devido a diversos fatores, dentre os quais pode-se destacar:

- Uso massivo de cargas não lineares acopladas ao sistema elétrico (Bakar, 2008).
- Equipamentos baseados em eletrônica utilizados nas residências, centros comerciais e indústrias.
- Proliferação do uso de geração distribuída (GD) e geração dispersa (Gd) no sistema de potência (Yadav & Srivastava, 2014), (Naik, Khatod, & Sharma, 2015).

Dessa forma, o monitoramento dos parâmetros dos sistemas de potência em tempo real, juntamente com a análise *off-line* dos mesmos, têm se tornado cada vez mais importante (Tcheou, et al., 2014).

Em diversas aplicações a aquisição contínua de dados, bem como seu armazenamento são necessários, tanto para fins de monitoramento quanto para a análise dos mesmos. Além disso, o pós-processamento desses dados pode revelar informações previamente não observadas, permitindo o aprimoramento do sistema, a solução de problemas, a otimização de algoritmos, dentre outros benefícios.

Entretanto, o armazenamento contínuo dos dados de sinais elétricos não é uma simples tarefa, devido à grande quantidade de dados a ser registrada e, posteriormente, transferida (LAI, 2014). Além disso, atualmente não existem muitos equipamentos voltados para o armazenamento contínuo de sinais amostrados a altas taxas (Maass, et al., 2013), ao invés disso, objetivam apenas a captura de pequenos trechos do sinal no qual ocorreu uma falha ou distúrbio (Moreto & Rolim, 2011) e (Qiong & Zhao-Hui, 2011).

A pesquisa na área de aquisição contínua de dados a altas taxas de amostragem, é um campo desafiador. Trabalhos têm sido realizados nessa área, e para a implementação dos mesmos

utiliza-se plataformas de desenvolvimento, dentre as quais está cada vez mais presente a FPGA (Friederich, et al., 2011).

Desse modo, a motivação para a utilização de um sistema detector e compressor é embasada em dois principais pilares: a necessidade de constante monitoramento dos parâmetros elétricos e a grande quantidade de massa de dados provenientes do sistema.

O presente trabalho apresenta um sistema que propõe ser capaz de recuperar inteiramente o sinal amostrado a uma alta taxa. Entretanto, o mesmo não registra e armazena todo o sinal de forma contínua. Ao invés disso, é capaz de detectar os distúrbios presentes no sinal e salvar apenas os trechos ao redor desses distúrbios, e além deles, informações sobre a frequência estimada do trecho onde não foi detectado nenhum distúrbio, a fim de reconstruir completamente o sinal. Essas informações são utilizadas para produzir o sinal entre dois distúrbios, recuperando o sinal elétrico inteiramente, mesmo quando a frequência seja variante no tempo.

A fim de possibilitar o armazenamento de dados provenientes de grandes períodos de monitoramento, são aplicadas técnicas de compactação sobre os trechos armazenados do sinal.

O Sistema de Detecção e Compressão de Distúrbios Elétricos (SDCDE), mostrado neste trabalho, possui algumas características que devem ser consideradas:

- Reconstrução do sinal armazenado com nível de perda controlado: o usuário do sistema pode parametrizá-lo, a fim de adaptá-lo às características dos sinais existentes no ambiente de medição.

- Flexibilidade de operação: a parametrização é flexível, podendo ser realizada através de um dispositivo móvel, com sistema operacional Android®. A partir deste dispositivo também podem ser enviados comandos de abertura e fechamento de arquivos, dentre outras configurações.

- Taxa de amostragem: os sinais são amostrados a altas taxas e com alta resolução. Isso permite uma reconstrução completa do sinal incluindo a presença de harmônicos de altas frequências e distúrbios de baixa frequência. A taxa utilizada é de 7680 Hz, representando 128 pontos a cada ciclo de 60 Hz. A resolução do conversor AD utilizado é de 16 bits.

- Otimização do espaço de memória utilizado: devido aos algoritmos de compactação utilizados, é possível obter um armazenamento de longos períodos de gravação em um cartão SD ou *pen drive*. O sistema suporta dispositivos de armazenamento de arquivos de até 32 Gbytes de capacidade.

- Núcleo básico de processamento: contém algoritmos de estimação de parâmetros da rede, detecção e compressão de distúrbios para análise da qualidade de energia. Utiliza-se o conceito de processador embarcado a fim de minimizar o consumo de lógica dentro do Dispositivo Lógico Programável (DLP) utilizado, ou seja, a FPGA.

- *Hardware* reconfigurável: a utilização de um *hardware* reconfigurável e universal permite não só a validação das pesquisas, como também a redução do tempo entre a pesquisa básica e aplicada à disponibilização de um produto comercial contendo as inovações das pesquisas.

De posse dessas ideias, essa dissertação mostra a implementação de um Sistema de Detecção e Compressão de Distúrbios Elétricos (SDCDE). É dado maior enfoque nas sínteses em FPGA, apesar de o sistema completo fazer uso de outras plataformas. Sendo assim, no Capítulo 2 é apresentada uma revisão bibliográfica, na qual são vistos tópicos que envolvem Teoria da Informação, compactação de dados, técnicas de estimação de parâmetros elétricos e técnicas de detecção. No Capítulo 3 é abordada a FPGA, sua arquitetura básica, projeto e programação. Nesse mesmo capítulo também é visto um pouco de uma Linguagem de Descrição de *Hardware* (HDL), a linguagem Verilog, utilizada para a criação de projetos e para a síntese da FPGA. O Capítulo 4 aborda as implementações do SDCDE na plataforma FPGA e um protótipo desenvolvido. Alguns resultados e análises são feitos no Capítulo 5 e, por fim, no Capítulo 6 são tiradas algumas conclusões gerais.

2. REVISÃO BIBLIOGRÁFICA

2.1. Introdução

Com o advento do processamento de dados e, juntamente com ele, o processamento digital de sinais, novas áreas de desenvolvimento acadêmico foram abertas, gerando diferentes e mais especializados campos de pesquisa. Em meados da década de 40, principalmente no seu fim, a área de processamento de sinais foi ganhando força e consolidando-se como disciplina nas universidades (NEBEKER, 1998).

Desde o surgimento dessa ampla e desafiadora área, os seus objetivos têm girado em torno de dois principais resultados: (*i*) extrair informação do sinal trabalhado e (*ii*) modificar o sinal para algum fim específico. Muitas vezes o segundo é necessário para que o primeiro seja efetivo.

Seja qual for a finalidade para a qual as técnicas de processamento de sinais forem empregadas, resulta-se em dados. Estes, para serem devidamente interpretados, precisam ser processados, o que geralmente requer um alto custo computacional. Quanto maior a quantidade de informação a ser extraída, maior o custo envolvido e, além disso, mais recursos de *hardware* como memórias, são necessários. Sendo assim, tange-se em um tópico importante, que vem a ser a compactação dos dados. Nesse âmbito, a Teoria da Informação se mostra de fundamental importância para se entender e solucionar problemas relacionados à compactação de dados.

Em 1948, Claude Shannon, conhecido como o “Pai da Teoria da Informação” publicou um dos seus maiores trabalhos: “*A mathematical theory of communication*” (SHANNON, A mathematical theory of communication, 1948) permitindo que várias outras áreas científicas usufruíssem dos princípios teóricos desenvolvidos pela Teoria da Informação (SHANNON, The Bandwagon, 1956). Esta se tornou precursora para a linha de pesquisa que envolve criação de códigos para fontes a fim de compactar a informação.

Inseridos nesse cenário, pesquisadores foram, ao longo dos anos, desenvolvendo técnicas de processamento de sinais e, dentro delas, foram desenvolvidos métodos de compactação cada vez mais eficazes e específicos, com a finalidade de serem obtidas altas taxas de compressão (FUSCO, 1990), (WYLIE, 1995), (CHAO, 2001). Atualmente existem diversas técnicas de compactação de dados, não só para sinais provenientes de sistemas elétricos, mas

principalmente para sinais de outra natureza, como sinais de áudio e imagem, dentre outros (SARAVANAN, 2013), (HAIBING, 2014), (WANG, XIANG, & JINGMING, 2014).

Como visto no Capítulo1, a compactação ou a compressão de sinais se mostra uma prática de fundamental importância para a manipulação, armazenamento e análise adequados dos registros de dados provenientes dos parâmetros elétricos, haja visto que a massa de dados resultante desses registros é extremamente grande – *Big Data* (LAI, 2014).

Considerando-se as tendências do sistema elétrico de se tornar cada vez mais inteligente e interconectado, pode-se prever que os desafios vindouros, com a implantação de um sistema adaptável, terão como um dos quesitos principais para sua funcionalidade e garantia da qualidade de energia, a compactação de dados (YIN, SHARMA, GORTON, & AKYOL, 2013). Além disso, outros fatores, como a inserção de medidores inteligentes com mais indicadores registrados em suas memórias, e ainda, a transmissão dos dados, que deve ser flexível, escalável e possuir um custo a ser levado em conta, fortalecem a razão para a utilização de técnicas de compactação de dados.

Este capítulo trata brevemente de algumas técnicas existentes para a compactação de dados, em especial as provenientes de registros de sinais do sistema elétrico, dando foco aos métodos utilizados para esta dissertação.

Sendo assim, na Seção 2.2 serão abordadas técnicas de compactação de sinais. Na Seção 2.3 serão vistos dois métodos de estimação de frequência. Na Seção 2.4, duas técnicas de detecção de distúrbios elétricos são mostradas e, finalmente, na Seção 2.5 serão feitas conclusões sobre o capítulo.

2.2. Técnicas de compactação de dados

A fim de se compactar a informação proveniente dos dados do sistema elétrico, ao longo dos anos, técnicas de compactação foram sendo desenvolvidas e aperfeiçoadas. Nesta seção será visto um pouco sobre compactação de dados de uma forma geral.

A seguir será feita uma breve revisão dos conceitos básicos envolvendo Teoria da Informação e então serão introduzidos os tópicos referentes à compressão.

2.2.1. Conceitos de Teoria da Informação (TI)

A Teoria da Informação é uma ramificação da Matemática e possui amplo alcance de seus conceitos (informação, entropia, redundância) (FINAMORE & RIBEIRO, 2013), interligando diversas áreas de conhecimento (COVER & THOMAS, 2006). Ela abrange várias ferramentas que estabelecem limitantes que ajudam no entendimento e solução de problemas de comunicação (compressão, transmissão e processamento da informação).

- **Fonte de informação.**

A fonte de informação (ou simplesmente fonte), é um elemento capaz de emitir informação toda vez que é usado. A cada uso, pode-se dizer que ocorre um evento. O evento pode ser a emissão de um símbolo (0 ou 1 para sistemas binários de um único dígito, por exemplo) ou um conjunto de símbolos formando uma palavra, isso depende do tipo da fonte.

- **Modelagem de uma fonte**

Pode-se modelar uma fonte de informação, probabilisticamente, como um conjunto de elementos a_i que podem ser emitidos pela fonte, formando um alfabeto $\mathcal{A}$ e associado a este alfabeto, um conjunto de probabilidades $\mathcal{P}$, que pode ser representado por um vetor de probabilidades $\mathbf{p}$, onde cada elemento p_i deste vetor é o valor da probabilidade do correspondente símbolo ser emitido pela fonte, ou seja, de um evento *E* ocorrer. Assim, uma fonte de informação pode ser modelada pelo conjunto $\mathcal{S} = (\mathcal{A}, \mathcal{P})$. Considerando um alfabeto contável e com um número finito *j* de elementos, diz-se que a cardinalidade de $\mathcal{A}$, simbolizada por $|\mathcal{A}|$ vale *j*. Assim, o índice *i*, quando associado à letra *a*, representa a posição do elemento dentro do alfabeto da fonte, ou seja, *i* pertence a um conjunto dos números naturais e vale, no máximo, *j*–1.

Cada símbolo emitido pela fonte pode ter sua representatividade temporal definida por u_n, onde o índice *n*, quando associado à letra *u*, pertence aos inteiros e representa um instante de tempo qualquer considerado. Como esse símbolo pode assumir um determinado valor de acordo com uma probabilidade, ele pode ser representado por uma variável aleatória U_n em cada instante *n*. A Figura 1, juntamente com o conjunto de expressões (2.1), resumem o que foi enunciado.

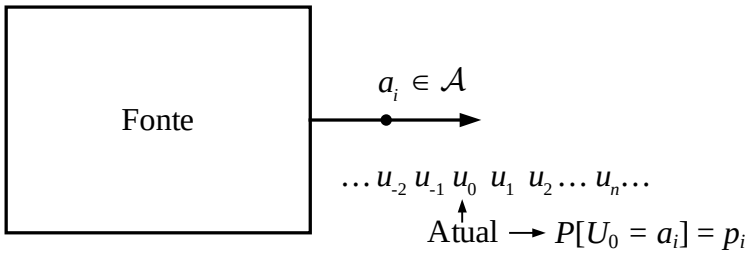


Figura 1: Fonte de informação.

$$\begin{gathered}\text{alfabeto: } \mathcal{A} = \{a_0, a_1, \cdots, a_{j-1}\} \rightarrow |\mathcal{A}| = j \\ \text{elementos do alfabeto: } a_i \in \mathcal{A};\ i \in \mathbb{Z}^+ \,|\, 0 \le i \le j-1 \rightarrow [0:j] \\ \text{probabilidades: } \forall a_i \subset \mathcal{A}\ \exists\ p_i \,|\, p_i = P[E] = P\left[\{\mathrm{U}_n = a_i\}\right] = P\left[\{a_i\}\right] \\ \mathcal{P} = \underline{P} = \{p_0, p_1, \cdots, p_{j-1}\} \\ \text{cardinalidade: } |\mathcal{A}| = |\mathcal{P}|\end{gathered} \tag{2.1}$$

- **Fonte DMS**

Uma fonte discreta na qual a probabilidade de um determinado símbolo ocorrer não depende em nada de um símbolo já ocorrido e que também, a ocorrência do elemento atual não influencia no símbolo seguinte, pode ser considerada uma fonte sem memória, ou DMS (*Discrete Memoryless Source*). Nela, não há nenhuma correlação entre um elemento emitido e o elemento subsequente. Em outras palavras, pode-se dizer que a probabilidade de que ocorra um determinado símbolo $u_n = a_i$ no tempo n (representado por a_{i_n}), dado que já ocorreu o elemento $u_{i-1} = a_{i_{n-1}}$ qualquer, do alfabeto $\mathcal{A}$, é o próprio valor da probabilidade de a_i. As equações em (2.2) representam esse tipo de fonte.

$$\mathcal{P} \rightarrow \begin{cases} P\left[\left\{a_{i_n}\right\}\right] = p_{j_n} \\ P\left[\left\{a_{i_n}\right\} \middle| \left\{a_{i_{n-1}}\right\}\right] = P\left[\left\{a_{i_n}\right\}\right] = p_{j_n} \text{ (condição sem memória)} \end{cases} \tag{2.2}$$

Pode-se estabelecer medidas de informação para fontes de informação. Uma das mais fundamentais é a entropia. À uma fonte DMS, com um vetor de probabilidades $\mathbf{p} = (p_0, p_1, p_2, ..., p_{j-1})$, está associada a medida de informação definida pela equação (2.3).

$$H(\mathbf{p}) = -\sum_{i=0}^{i=j-1} p_i \log_2(p_i) \quad \text{(função entropia)} \tag{2.3}$$

Esta medida é denominada de entropia da fonte. Para as probabilidades com valor nulo, convenciona-se o logaritmo dessa probabilidade como nulo também. Se um elemento possui probabilidade de ocorrência igual a 1, consequentemente todos os outros terão probabilidade nula, então a entropia vale 0.

A entropia é o limitante inferior no que diz respeito à quantidade de bits necessária para representar os símbolos emitidos pela fonte. Nesse sentido, a unidade para a entropia é bits por símbolo da fonte (b/sf). A demonstração para este teorema pode ser vista em (YEUNG, 2007).

- **Codificação**

A compressão da informação é nada mais do que um código aplicado à informação da fonte. A cada símbolo da fonte u_i é atribuído um código $\mathbf{c}_i$, que possui uma largura l_i em número de bits.

- **Codificadores para fonte**

O codificador é o elemento responsável pela codificação. Na sua forma mais simples, ele mapeia as variáveis aleatórias U_i, vindas da fonte e acopladas em sua entrada, em outras variáveis aleatórias de códigos $\mathbf{C}_i$, com um ou mais bits. Em outros termos, ele atribui palavras código às palavras de informação. A entrada do codificador pode assumir qualquer valor do alfabeto $\mathcal{A}$, e sua saída pertencerá ao alfabeto $\mathcal{B} \subset \mathcal{B}^*$, cuja definição está em (2.4). Codificadores de fontes, essencialmente, executam algoritmos de compressão como forma de processar a informação vinda da fonte.

$$\mathcal{B}^1=\{1,0\};\ \mathcal{B}^2=\{00,01,10,11\};\ \mathcal{B}^3=\{000,001,010,011,100,101,110,111\}$$
$$\mathcal{B}^n=\{\underbrace{000...0}_{n\text{ bits}},000...1,00...10,\cdots,111...1\};$$
$$\mathcal{B}^\infty=\{\underbrace{00...00}_{\infty\text{ bits}},000...01,000...10,\cdots,111...11\};\ \boxed{\mathcal{B}^*=\bigcup_{n=1}^{\infty}\mathcal{B}^n} \tag{2.4}$$

A Figura 2 representa um esquema de codificação simples.

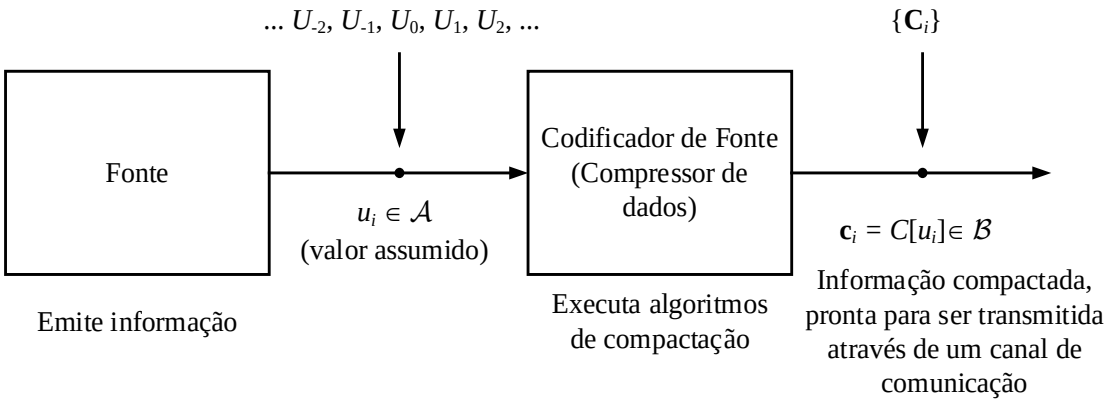


Figura 2: Diagrama de representação para processo de codificação de fonte.

2.2.2. Classificação de códigos de compressão

Algoritmos de compressão podem ser classificados, dentre outras, em duas principais categorias: Quanto ao tamanho das entradas e saídas e com base na perda de dados.

- **Quanto ao tamanho das entradas e saídas do codificador**

Se, durante o processo de codificação, o codificador suporta um tamanho variável de palavras de informação em sua entrada, adquirindo, a cada iteração, um número possivelmente diferente de símbolos u_i, mas em sua saída, são geradas palavras código $\mathbf{c}_i$ de largura l_i fixa de bits, diz-se que é um codificador variável-fixo. Dessa forma, tem-se quatro classes: fixo-fixo, fixo-variável, variável-variável e variável-fixo. A Tabela 1, mostra exemplos de algoritmos dentro dessas classes.

Tabela 1: Tipos de algoritmo, quanto ao tamanho das entradas e saídas.

Técnica	Tamanho do agrupamento entrada → saída		
CS&Q	fixo	→	fixo
Huffman	fixo	→	variável
Codificação Aritmética	variável	→	variável
run-lenth, LZW	variável	→	fixo

- **Quanto à perda de dados**

Uma das principais categorias, na qual tem-se maior interesse quando se trata de compressão, está relacionada à perda de dados.

Todas as técnicas descritas a seguir são apresentadas de forma resumida, a partir de (SMITH, 2003).

Existem códigos com perda (*lossy*) e sem perda (*lossless*). Em técnicas de compressão sem perda, os dados recuperados após a descompressão são idênticos aos originais. Esse tipo de compressão é indispensável para vários tipos aplicações, como em códigos de programas executáveis e textos. Em outros casos é permitida a perda de dados, como sinais de imagens e sons. Nesse caso, a compactação é muito mais eficaz. A Tabela 2 mostra outros exemplos de técnicas:

Tabela 2: Códigos com perda e sem perda

Códigos sem perda	Com perda
Códigos *run-length*	JPEG
Huffman	MPEG
Delta	MP3
LZW	Compressão por *wavelets*[1]

Códigos de compressão do tipo CS&Q (*Coarser Sampling and/or quantization*), trabalham de forma a descartar certas amostras do sinal adquirido, como um processo de *downsampling*, e

[1] Apesar da Transformada *Wavelet* ser uma ferramenta de análise de sinais, compreendendo aplicações muito mais amplas do que as realizadas apenas em técnicas de compressão com perdas, essa utilização foi classificada dessa forma devido à presença dessa abordagem neste trabalho. Esse fato não limita a sua utilização apenas para os fins citados.

quantizar as amostras restantes, promovendo um alto grau de compactação, mas inserindo perda de informação.

Compressão com códigos do tipo *run-length* são bem eficientes para sequências onde um mesmo caractere se repete muitas vezes seguidas. Nela, basicamente se substitui o elemento repetido *n* vezes seguidas, pelo próprio elemento e o número *n* em sequência.

Compressões com códigos de Huffman atribuem palavras código com menor número de bits aos símbolos mais usados e, aos símbolos menos usados, códigos maiores. Num arquivo contendo textos, por exemplo, os símbolos mais presentes são espaços, logo em seguida vêm as letras minúsculas. Dentre as letras, ainda há aquelas que se repetem mais vezes. Na língua portuguesa, por exemplo, tem-se a letra "a", com maior frequência relativa (QUARESMA, 2008).

A compressão delta trabalha com a diferença entre as amostras numa mesma sequência, comparando o primeiro valor com os demais. Ela é mais eficaz em sinais com formato mais suave, como ondas sonoras.

Compactação do tipo JPEG e MPEG são mais sofisticados, eles são utilizados, respectivamente, para compactação de imagens e vídeos, com ou sem áudio. Mais detalhes desses métodos podem ser encontrados em (SMITH, 2003).

Compactação MP3 é utilizada para arquivos de áudio. Nela, os componentes de frequência não perceptíveis à maioria dos seres humanos são retirados, restando apenas os componentes da faixa audível.

A seguir serão abordados mais profundamente dois dos algoritmos utilizados para este trabalho, a saber: compressão LZW e compressão por *wavelets*.

2.2.3. Compactação sem perda - LZW

Esse tipo de compactação foi nomeado dessa forma devido aos seus idealizadores, A. Lempel e J. Ziv (LEMPEL & ZIV, 1977). Posteriormente houve modificações por Terry A. Welch, a partir de então denominou-se o algoritmo como Lempel-Ziv-Welch (LZW). Uma das principais diferenças entre o algoritmo LZW e seu antecessor é que o dicionário, na compactação LZW,

não inicia-se vazio, mas com todos os caracteres individuais possíveis, ou raízes (ROMANO, 2001).

A técnica desenvolvida é bem versátil, universal e é utilizada atualmente por vários aplicativos computacionais comerciais. Sua eficácia é mais notável em arquivos contendo redundâncias repetitivas. Ele é capaz de identificar trechos que contenham familiaridade com trechos antecedentes. Sua filosofia é baseada em elementos como dicionário e átomos, ou palavras, contidas nesse dicionário.

Conforme o processo de compactação ocorre, seu dicionário é incrementado com palavras formadas pelos trechos familiares, compostos por mais de um símbolo u_i da fonte. Quando um novo trecho semelhante a alguma palavra, já presente no dicionário, é percebido, a representação codificada é feita usando apenas o endereço do dicionário, correspondente àquela palavra, que representa mais de um símbolo u_i da fonte. Assim, é feita a compressão.

Exemplo 2.1

A Tabela 3 mostra um exemplo de dicionário, já pronto, com 4096 posições, contendo as raízes para o conjunto de elementos do código ASCII e outras palavras, que foram formadas com o decorrer da compressão, preenchendo todo o resto do dicionário.

No sistema ASCII tem-se 256 códigos para representar os caracteres, utilizando portanto, 8 bits ou 1 byte de largura. Como a tabela possui 4096 posições, são necessários 12 bits de endereçamento.

A construção do dicionário é feita durante a compressão. Após completada a construção do mesmo, a linha de dados do arquivo pode continuar sendo compactada, trocando-se os trechos familiares pelos correspondentes códigos criados e guardados no dicionário. Esse processo é mostrado na Figura 3.

Através dessa figura é possível ver que, para representar a informação, que antes usava 152 bits para armazenar os 19 símbolos da tabela ASCII, após a codificação, passou a utilizar 144 bits, ou em outras palavras, 1 byte a menos. Obviamente, considerando textos, e também dicionários maiores, a compactação pode ser muito mais expressiva.

Tabela 3: Exemplo de dicionário para o algoritmo Lempel-Ziv-Welch.

	Palavras código		Informação		Significado
	(12 bits)	Dec.	(em bytes)	Decimal	
Raízes do código ASCII	000000000000	0	00000000	0	*Null* - NUL
	000000000001	1	00000001	1	*Start of Heading* - SOH
	000000000010	2	00000010	2	*Start of Text* - STX
	000000000011	3	00000011	3	*End of Text* - ETX
	000000000100	4	00000100	4	*End of Transmission* - EOT
	⋮	⋮	⋮	⋮	⋮
	000011111101	253	11111101	253	²
	000011111110	254	11111110	254	■
	000011111111	255	11111111	255	
Códigos únicos criados	000100000000	256	[01100001 01100101]	[97 101]	ae
	000100000001	257	[01100101 01101001]	[101 105]	ei
	000100000010	258	[01101001 01100001]	[105 97]	ia
	000100000011	259	[01100001 01101111]	[97 111]	ao
	⋮	⋮	⋮	⋮	⋮
	111111111101	4093	[01101111 01101110]	[111 110]	on
	111111111110	4094	[01110100 01100101]	[116 101]	te
	111111111111	4095	[01110001 01110101 01100101]	[113 117 101]	que

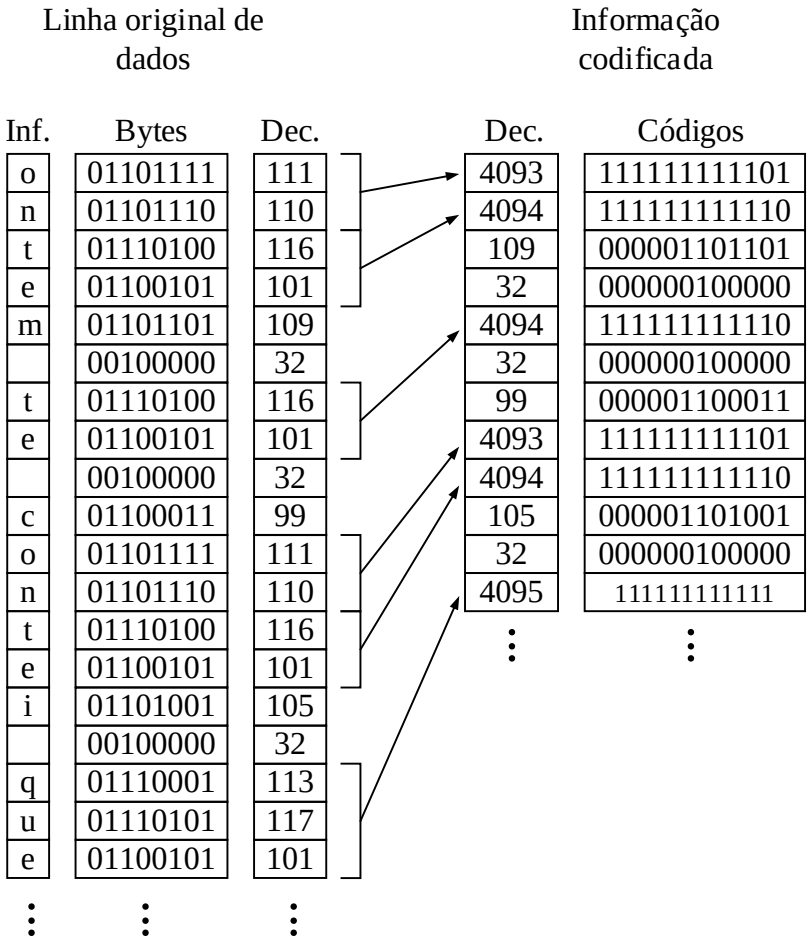
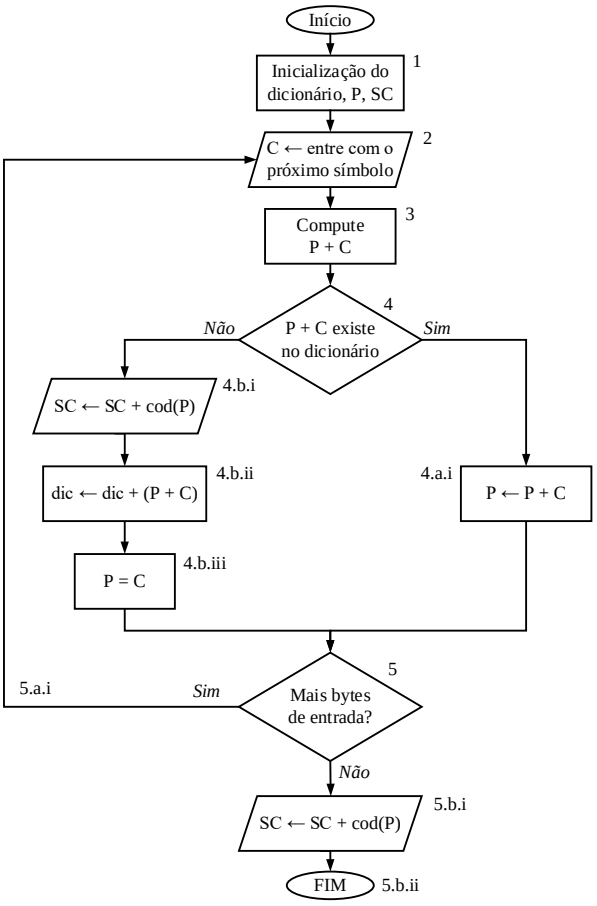


Figura 3: Exemplo de informação codificada.

O processo de compressão ou codificação do algoritmo LZW pode ser visualizado através do diagrama de blocos na Figura 4, juntamente com um pseudocódigo indicando os passos do diagrama. Já o processo de descompressão de um arquivo codificado pode ser visto na Figura 5, juntamente como seu correspondente pseudocódigo. Ambos os diagramas estão relacionados ao sistema da Figura 6.



Pseudocódigo:

1. No início, o dicionário contém todas as raízes possíveis: caracteres individuais (Códigos ASCII, 0 → 255).
P é inicializado vazio (P = Ø); Sequência Codificada é inicializada vazia. SC = Ø;
2. C ← Próximo caractere da sequência de entrada;
3. Compute P + C = [P,C];
4. P + C existe no dicionário?
   a. se *sim*,
      i. P ← P + C;
   b. se *não*,
      i. Coloque a palavra código correspondente a P na sequência codificada ;
      ii. Adicione a string P + C ao dicionário;
      iii. P ← C;
5. Existem mais caracteres na sequência de entrada?
   a. se *sim*,
      i. Volte ao passo 2;
   b. se *não*,
      i. Coloque a palavra código correspondente a P na sequência codificada;
      ii. FIM.

Figura 4: Diagrama de blocos e pseudocódigo representando o processo de compactação LZW.

Nos diagramas, algumas simbologias são interpretadas como se segue:

C: é um caractere;

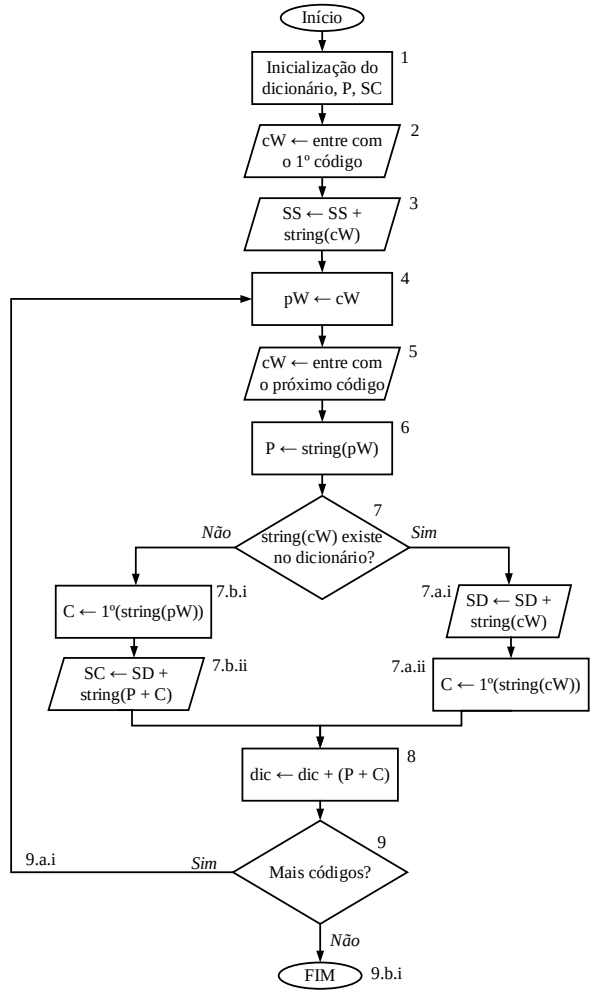
P: *string* que representa um prefixo;

cW: palavra código;

pW: palavra código que representa um prefixo;

P + C = concatenação de P com C = [P,C], ambos *strings*;

string(X): *string* correspondente à palavra código X.



Pseudocódigo:
1. No início, o dicionário contém todas as raízes possíveis: caracteres individuais (Códigos ASCII, 0 → 255).
P é inicializado vazio (P = Ø); Sequência de Saída é inicializada vazia. SS = Ø;
2. cW ← Primeira palavra código as Sequência Codificada: 1º(SC), sempre é uma raiz;
3. Coloque string(cW) na Sequência de Saída;
4. pW recebe o valor e cW;
5. cW recebe a próxima palavra código;
6. P recebe a string(pW);
7. A string(cW) existe no dicionário?
   a. se *sim*,
      i. Coloque a string(cW) na Sequência de Saída;
      ii. C recebe o primeiro caractere da string(cW);
   b. se *não*,
      i. C recebe o primeiro caractere da string(pW);
      ii. Coloque string: P + C na Sequência de Saída;
8. Adicione P + C ao dicionário;
9. Existem mais palavras codigo?
   a. se *sim*,
      i. Volte ao passo 4;
   b. se *não*,
      i. FIM.

Figura 5: Diagrama de blocos e pseudocódigo representando o processo de descompactação LZW.

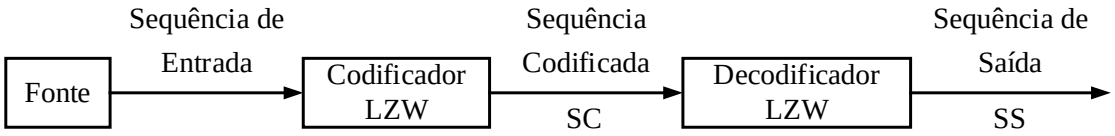


Figura 6: Sistema de compressão e descompressão LZW.

Para este trabalho, foi utilizado o algoritmo LZW proposto em (NANDI & MANDAL, 2012), onde buscou-se otimizá-lo em termos de quantidade necessária de memória.

A plataforma de desenvolvimento utilizada para a implementação de todos os circuitos digitais deste trabalho foi a FPGA (*Field-Programmable Gate Array*), que é o Dispositivo Lógico Programável (DLP) para o qual o foco deste trabalho está voltado.

2.2.4. Compactação com perda - *wavelets*

Wavelet é uma forma de onda de duração limitada e que possui valor médio nulo, e às vezes, formato irregular e assimétrico. As *wavelets* são apropriadas para descrever e analisar pulsos, anomalias e outros eventos dentro do sinal (FUGAL, 2009).

A transformada *Wavelet* possui todo um arcabouço teórico complexo por base. Nesta subseção, a abordagem deste tema será tratada com o objetivo apenas de inserir esse tópico no contexto da compactação.

A utilização da *Wavelet* se dá principalmente em aplicações onde é necessária a análise de sinais não-estacionários[2]. Um sinal com distúrbio, por exemplo, é um sinal não-estacionário, o seu espectro de frequência na hora do distúrbio é diferente do espectro fora dele, portanto é variante com o tempo.

Se um determinado sinal possuir mais de uma frequência e for analisado utilizando a transformada de Fourier tradicional, ela mostrará as frequências existentes em todo o sinal, porém ela não mostrará a variabilidade temporal destas frequências, caso existam.

O princípio da transformada *Wavelet* consiste em uma decomposição hierárquica de um sinal de entrada em uma série de sinais sucessivos com resolução mais baixa, provendo um meio efetivo para a análise do sinal em várias escalas e resoluções. Ela permite a visualização de altas frequências em janelas curtas e de baixas frequências em janelas longas. Desta forma, as características de um sinal não-estacionário podem ser melhor monitoradas (HUANG, YANG, & HUANG, 2002).

Graças à capacidade de decompor os sinais, tanto no domínio da frequência quanto no domínio do tempo, as funções *wavelet* são ferramentas poderosas de processamento de sinais, muito aplicadas na compressão de dados, eliminação de ruído, separação de componentes no sinal, identificação de singularidades, distúrbios, detecção de auto-semelhança, dentre outras aplicações. De fato a Transformada *Wavelet* Discreta (DWT) tem sido amplamente usada para propósitos de compactação, devido à sua habilidade de representar um sinal esparsamente (LIVANI & EVRENOSOGLU, 2014), (ZHANG, LI, & HU, 2011) e (COSTA, 2014).

[2] Sinais não estacionários possuem momentos estatísticos, média, variância, e etc., que podem variar em quaisquer segmentos tomados deste sinal (BOLZAN, 2006).

A *wavelet* utilizada para o desenvolvimento deste trabalho foi a do tipo Daubechies 3[3]. Uma representação da forma de onda desta *wavelet* pode ser observada na Figura 7.

A implementação desta transformada pode ser realizada através de uma cascata de filtros passa-altas e passa-baixas com suas saídas decimadas por um fator 2, conectados conforme mostrado na Figura 8. Cada filtro é do tipo FIR de quinta ordem, representado pela equação de diferenças (2.5), onde $y[n]$ é a saída do filtro, $x[n]$ é a entrada e i é o atraso das amostras do sinal de entrada. Os coeficientes dos filtros passa-altas e passa-baixas podem ser observados na Tabela 4 e tiveram de ser quantizados para uma implementação direta em FPGA (KAPISCH, SILVA, MARTINS, FILHO, & DUQUE, Implementação em FPGA de Transformada Wavelet para Compactaçõ de Sinais Elétricos de Sistemas de Potência Utilizando Processador Embarcado, 2014).

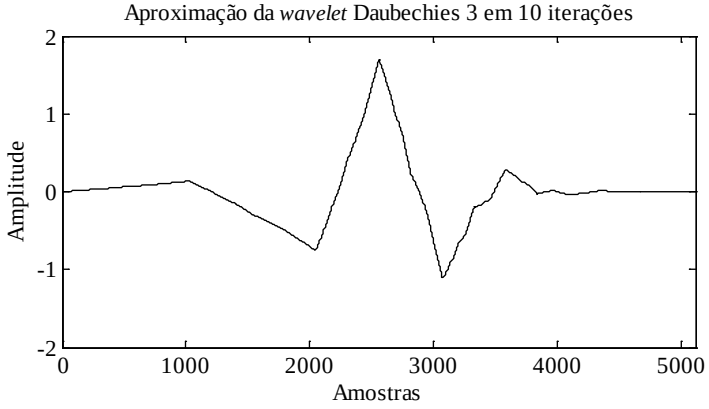


Figura 7: *Wavelet* do tipo Daubechies 3.

[3] Ingrid Daubechies (Houthalen-Helchteren, 17 de agosto de 1954), física e matemática belga.

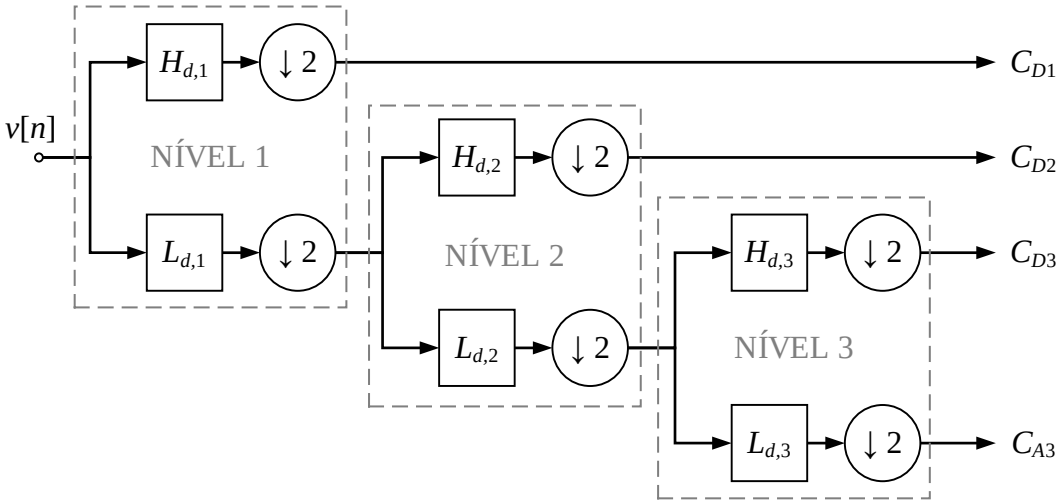


Figura 8: Implementação da transformada *Wavelet* para compressão com perdas.

$$y[n] = \sum_{i=0}^{5} x[n-i]a_i \qquad (2.5)$$

Tabela 4: Coeficientes dos filtros da *wavelet* Daubechies 3.

		Valores em ponto flutuante	Valores em ponto fixo
passa-altas	a_0	–0,3327	–10900
	a_1	0,8069	26440
	a_2	–0,4599	–15069
	a_3	–0,1350	–4424
	a_4	0,0857	2799
	a_5	0,0352	1154
passa-baixas	a_0	–0,0352	–1154
	a_1	–0,0857	–2799
	a_2	–0,1350	–4424
	a_3	0,4599	15069
	a_4	0,8069	26440
	a_5	0,3327	10900

A Figura 9 mostra a representação simplificada das faixas de corte e passagem dos filtros do esquema apresentado, em relação à frequência de amostragem F_s do sistema.

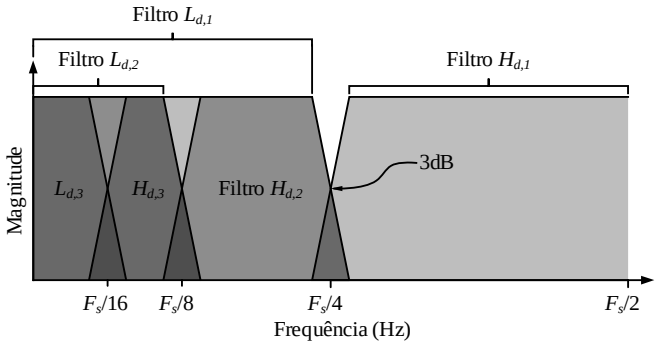


Figura 9: Representação da resposta em frequência dos filtros da árvore de filtros da *wavelet*.

O processo começa com a entrada do sinal $v[n]$ no primeiro nível da Figura 8. Depois, ele é decomposto nos coeficientes de detalhe (C_{D1}), contendo componentes de alta frequência, e nos coeficientes de aproximação (C_{A1}), contendo as baixas frequências. C_{D1} representa uma saída do processo, enquanto C_{A1} é dividido novamente com filtros iguais aos que existem no primeiro nível, resultando em C_{D2} e C_{A2}. C_{D2} se torna outra saída e C_{A2} é subdividido, por fim, em C_{A3} e C_{D3}.

Os quatro coeficientes finais utilizados para a etapa de compressão são C_{D1}, C_{D2}, C_{D3} e C_{A3}. Nesta ordem, esses coeficientes revelam os componentes de mais alta frequência à mais baixa frequência do sinal original e a partir deles é possível compactar o sinal de entrada.

A compactação se dá através da aplicação de limiares aos coeficientes de detalhe. Todos os valores abaixo desse limiar são substituídos por zero. Para se reconstruir o sinal de entrada, basta aplicar a estrutura de síntese, composta por dois pares de filtros, geralmente iguais aos da estrutura de análise ou decomposição da Figura 8, com os devidos atrasos, *upsamplings* e somas. Apesar desse processo modificar o sinal, essa distorção pode ser desprezível, utilizando-se um correto patamar de corte, resultando numa reconstrução quase ideal e ao mesmo tempo facilitando a compactação LZW, uma vez que o sinal poderá conter muitos zeros consecutivos.

Exemplo 2.2

A Figura 10 mostra um exemplo de decomposição *Wavelet* para um sistema de aquisição com frequência de amostragem de 7,68 kHz, ou 128 pontos por ciclo da fundamental, que é 60 Hz. A *wavelet* é a Daubechies 3. O sinal de entrada foi submetido a um distúrbio transitório de alta frequência com decaimento exponencial.

Os coeficientes da *wavelet* estão intactos, ou seja, ainda não foi aplicado nenhum limiar.

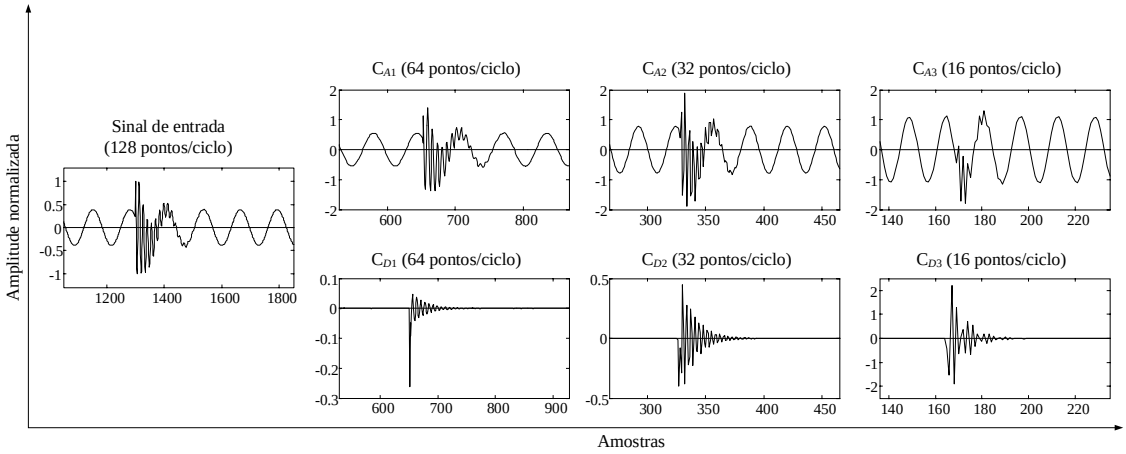


Figura 10: Exemplo de decomposição *Wavelet*.

Como dito anteriormente, a *wavelet* do tipo Daubechies 3 foi a escolhida para o sistema proposto neste trabalho. O motivo reside no fato de que ela possui fácil implementação e boa separação das frequências. Foram realizados testes com outras famílias, mas, dentro dos critérios adotados, está se mostrou mais apropriada para o projeto.

2.2.5. Fatores de importância na escolha do método de compactação adotado

A área de compactação de sinais evoluiu de tal forma que para cada tipo de sinal, existe uma gama de possibilidades para se realizar a compactação. Portanto, para se discriminar o método mais adequado, devem ser levados em conta alguns fatores, que envolvem até mesmo a plataforma de implementação, a qual para este trabalho foi a FPGA. Dentre esses fatores, destacam-se:

1. Filosofia adotada: como um dos mais importantes fatores a serem mencionados está a tolerância à perda de dados. Dependendo da análise proposta, a reconstrução pode permitir alguma perda de informação (*lossy compression*) (PARSEH, ACEVEDO, KANSANEN, MOLINAS, & RAMSTAD, 2012). Em outros casos os dados devem permanecer intactos após a descompressão (*lossless compresion*) (KIEFFER & YANG, 1998), numa aplicação onde existe a exigência de uma reconstrução perfeita. Caso seja permitida alguma perda, deve-se quantificar o nível tolerável dessa perda, de modo que

não prejudique a análise após a reconstrução. Uma vez decidida qual a filosofia adotada, algumas técnicas são mais apropriadas que outras.

2. Complexidade do(s) algoritmo(s) adotado(s): para um sistema completo de compactação de dados, podem estar presentes mais de um, ou até mesmo vários algoritmos interconectados sendo executados simultaneamente. Dependendo da complexidade do sistema a partir do método proposto, pode não ser possível a implementação em uma plataforma FPGA com pouco recurso disponível, necessitando de uma adaptação do algoritmo ou então a troca da plataforma.

3. Poder de processamento do dispositivo a ser utilizado: um fator extremamente importante a ser considerado para a escolha do método de compactação é a capacidade de processamento da plataforma a ser utilizada para a implementação. Esse item está intimamente ligado com o anterior. Algoritmos de compactação extremamente complexos, que necessitem de grande disponibilidade de operações e recursos para serem implementados, exigem que a FPGA, por exemplo, disponha de um número elevado de blocos lógicos, blocos DSP de multiplicação, alta frequência de operação, muitos blocos de memória (RAM), dentre outros fatores.

4. Tempo de processamento para compactação: a aplicação determina quanto tempo há disponível para o processamento de compactação dos dados. Se for uma aplicação *on-line*, por exemplo, necessita-se de uma implementação que seja executada rapidamente, antes da próxima amostra do sinal de entrada ser entregue. Aplicações *off-line* permitem mais tempo de processamento, permitindo algoritmos mais complexos e mais lentos.

5. Tempo de processamento para descompactação: do mesmo modo que o processo de compressão, a metodologia adotada para a descompressão ou descompactação necessita obedecer a exigência de tempo, caso exista.

6. Preço da plataforma de desenvolvimento existente no mercado: esse também é um dos fatores de importância a serem considerados ao se projetar uma implementação de *hardware* e influencia diretamente no método de compactação a ser escolhido. Na literatura podem ser encontrados casos em que o principal foco foi a otimização do uso de espaço lógico dentro do DLP a fim de possibilitar a utilização de dispositivos com um preço acessível de mercado, visando o fator custo-benefício (BETTA, FERRIGNO, & LARACCA, 2013).

O trabalho posposto nesta dissertação permeou por esses fatores a fim de projetar um protótipo funcional de um sistema de detecção e compressão de distúrbios elétricos em FPGA, que pudesse abranger as características necessárias de um dispositivo inteligente, preparado para as exigências que a evolução do sistema elétrico impõe, sem possuir um alto custo total de implementação.

2.3. Técnicas de estimação da frequência fundamental

Um importante parâmetro de um sistema elétrico de potência é a sua frequência fundamental, devido ao fato de que ela é geralmente usada para indicar o estado de operação do sistema. A precisão e acurácia com a qual a frequência fundamental é estimada, permite, dentre outros resultados, principalmente um bom monitoramento de outros parâmetros para qualidade de energia (RIBEIRO, DUQUE, SILVEIRA, & CERQUEIRA, 2014).

Construir um bom estimador de frequência, que seja robusto às variações de amplitude, causadas por distúrbios, é um desafio para a pesquisa de processamento de sinais. Uma estimação correta da frequência é de fundamental importância para o sistema de compactação proposto por este trabalho.

Nesta subseção serão abordados dois métodos de estimação de frequência: *phase-locked loop* (PLL) e o método baseado no cruzamento por zero.

2.3.1. *Phase-locked loop* - PLL

Esse método baseia-se na comparação da forma de onda senoidal que se deseja saber a frequência, com uma onda senoidal ideal conhecida.

O método PLL utilizado para testes neste trabalho é apresentado em (SILVA, NOVOCHADLO, & MODESTO, 2008) e (CIOBOTARU, TEODORESCU, & BLAABJERG, 2006). Um diagrama de blocos representando seu funcionamento pode ser visualizado através da Figura 11 (KAPISCH, et al., An electrical signal disturbance detector and compressor based on FPGA platform, 2014).

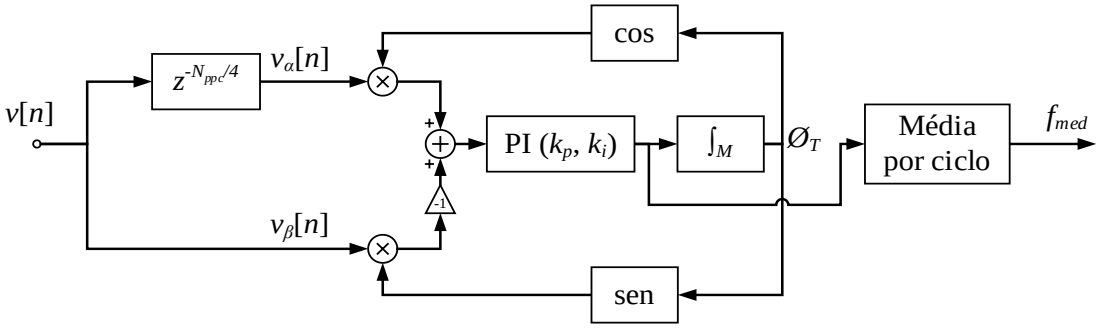


Figura 11: Diagrama de blocos do algoritmo de estimação de frequência, PLL.

Neste método, a frequência instantânea do sinal de entrada $v[n]$ é calculada, e com esse resultado calcula-se a frequência média por ciclo.

O processo começa atrasando-se a entrada $v[n]$ por $N_{ppc}/4$ amostras, onde N_{ppc} é o número de pontos por ciclo do sinal amostrado. Esse atraso gera $v_\alpha[n]$, defasado de 90º em relação ao sinal de entrada $v[n]$ (Figura 12), enquanto que $v_\beta[n]$ é o próprio sinal de entrada. $v_\alpha[n]$ e $v_\beta[n]$ são multiplicados, respectivamente, pelo cosseno e pelo seno da fase total estimada $\varnothing_T$. O produto envolvendo $v_\beta[n]$ é multiplicado por um ganho unitário negativo e em seguida, é somado ao produto envolvendo $v_\alpha[n]$. A soma é aplicada a um controlador Proporcional Integral (PI), de onde sai a frequência instantânea estimada. A partir da saída desse controlador é retirada a fase total $\varnothing_T$, da qual são extraídos o seno e o cosseno, através de uma aproximação por série de Taylor truncada em 10 termos.

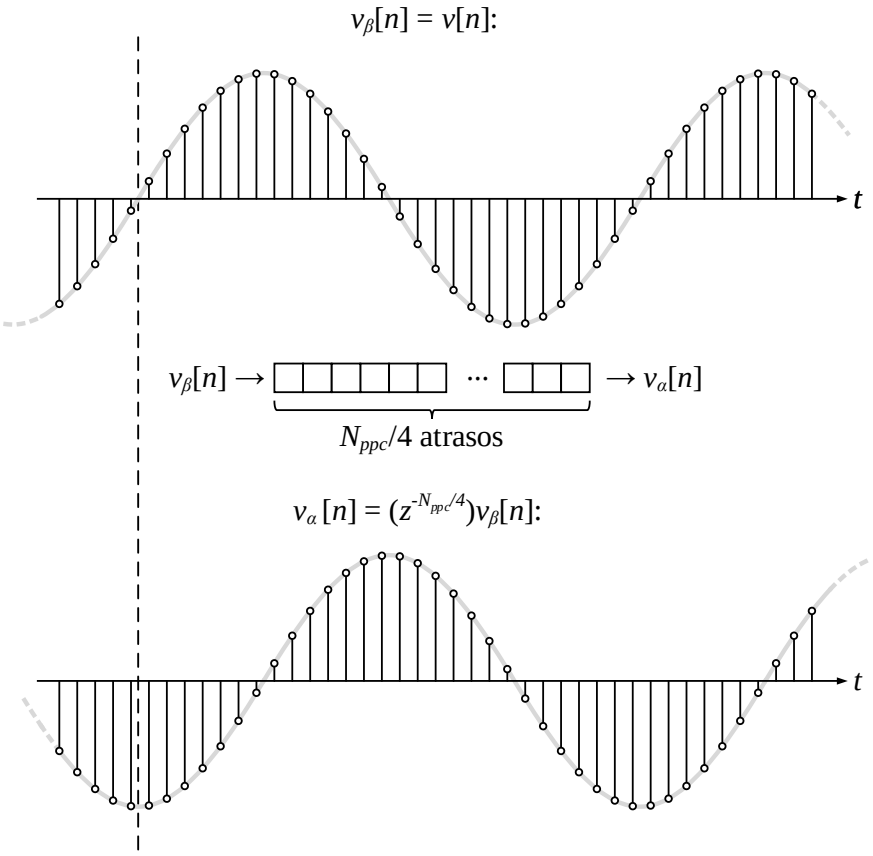


Figura 12: Defasagem de 90º de $v_\alpha[n]$ em relação a $v_\beta[n]$.

Uma vantagem da utilização desse método reside no fato de ele ser robusto à presença de ruídos no sinal. Além disso, obtém-se uma melhora significativa ao se colocar um estágio de pré-filtragem aplicado ao sinal de entrada. Em contrapartida, uma desvantagem é o tempo que o método leva para convergir ao valor estimado de frequência e, caso a aplicação exija velocidade de convergência, esse fator pode ser reprovativo. Esse tempo pode ser otimizado através da manipulação das constantes k_p e k_i presentes no controlador PI, de modo a acelerar a convergência da estimação e diminuir a oscilação presente na mesma, em regime permanente.

Outro fator que pode ser considerado negativo é a presença de realimentação no algoritmo, o que proporciona maior vulnerabilidade a instabilidades, caso o sinal de entrada não seja devidamente condicionado, por exemplo, em termos de nível de *off-set*.

Exemplo 2.3

Para um sinal de entrada v_1, contendo um transitório de alta frequência, foi utilizado o algoritmo para estimação da frequência fundamental descrito anteriormente. A Figura 13 a) mostra o sinal

de entrada v_1, o qual possui frequência fundamental de 60 Hz. A Figura 13 b) mostra a estimação da frequência fundamental para o sinal v_1 utilizando o método PLL.

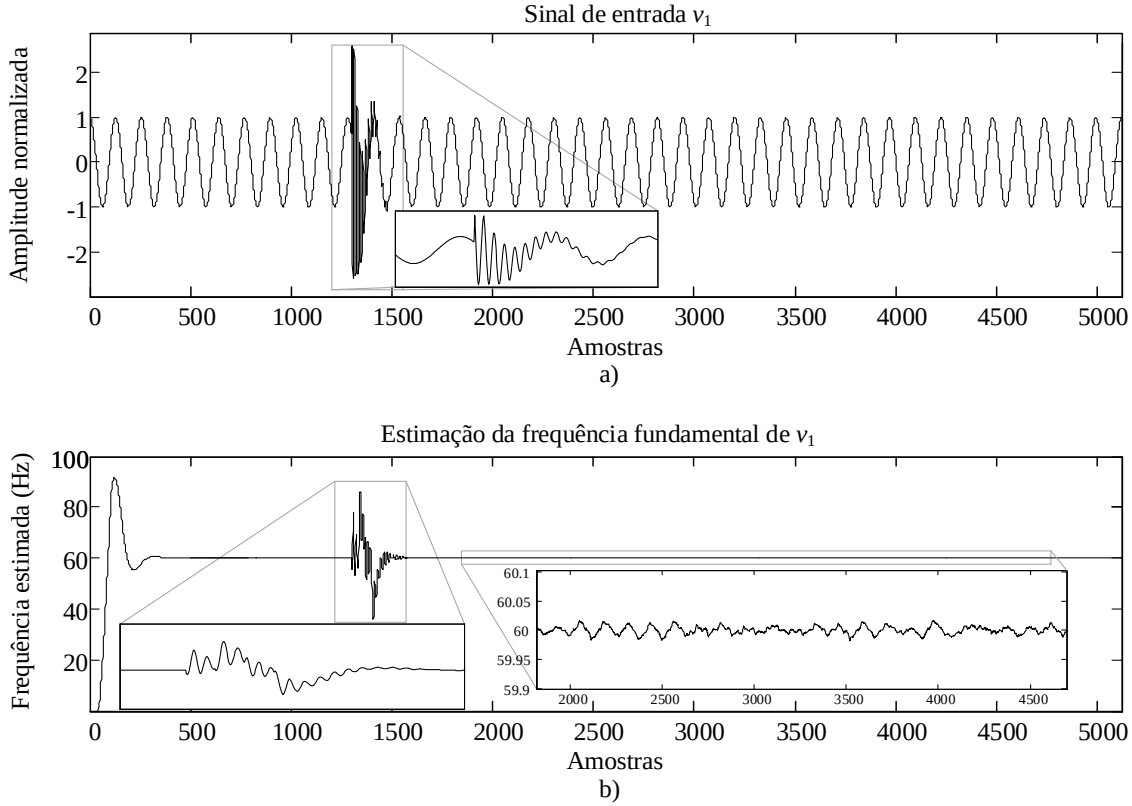


Figura 13: Estimação da frequência fundamental de um sinal com transitório utilizando o método PLL. a) Sinal de entrada v_1. b) Estimação da frequência fundamental de v_1.

A seguir, outro sinal v_2, contendo um aumento na amplitude (*Swell*), tem sua frequência estimada pelo mesmo método (Figura 14).

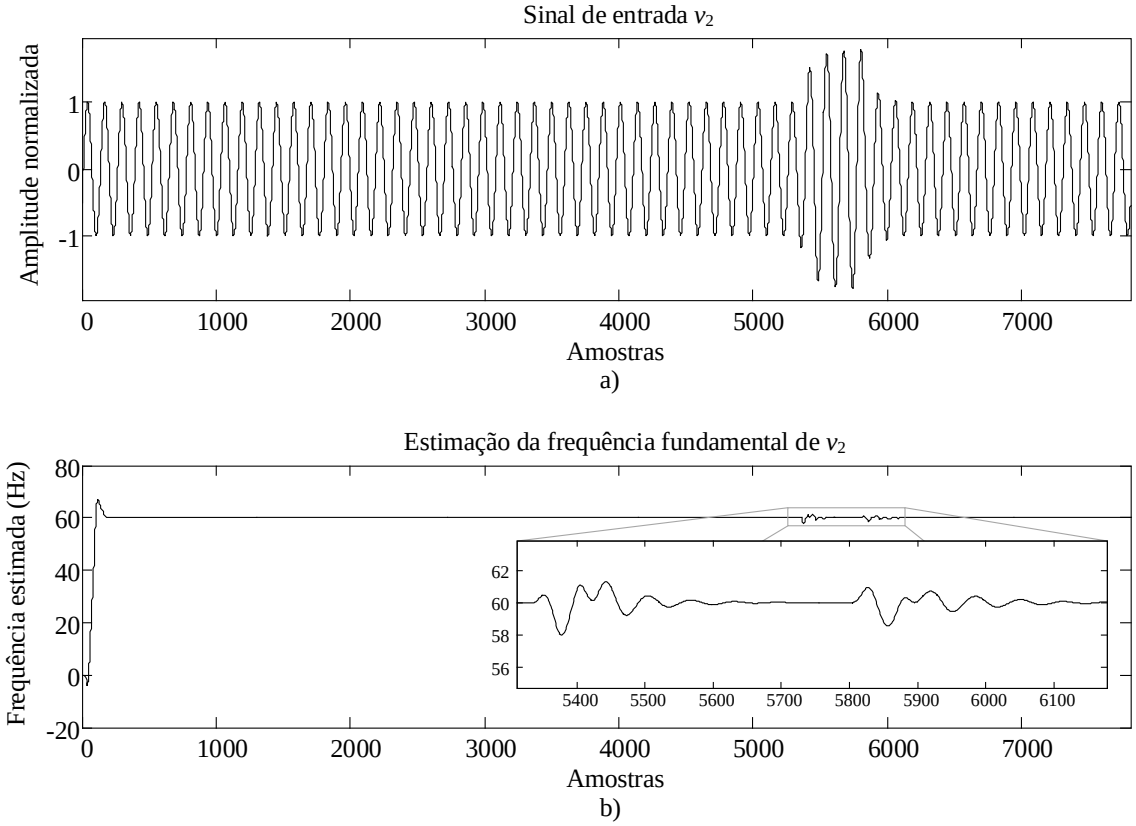


Figura 14: Estimação da frequência fundamental de um sinal com elevação de amplitude utilizando o método PLL. a) Sinal de entrada v_2. b) Estimação da frequência fundamental de v_2.

É importante observar o tempo de convergência do método de estimação, que alcançou estabilidade após a 380ª amostra, correspondendo a aproximadamente 3 ciclos, para o sinal com transitório. O método também apresenta grande sensibilidade à variação de amplitude e, além disso, mesmo depois do estado permanente ser atingido, existe uma pequena oscilação.

2.3.2. Cruzamento por zero (*Zero Crossing*)

Esse método é tradicional e um dos mais simples de se implementar, pois baseia-se na contagem de amostras do sinal de entrada entre duas passagens por zero na mesma direção. Sob algumas modificações, pode ser bem atrativo, com respeito à precisão e ao tempo de convergência.

O cruzamento por zero ocorre quando o sinal passa de um valor positivo para um negativo, ou vice-versa. Portanto, o primeiro passo é encontrar dois ou mais cruzamentos na mesma direção, como mostrado na Figura 15.

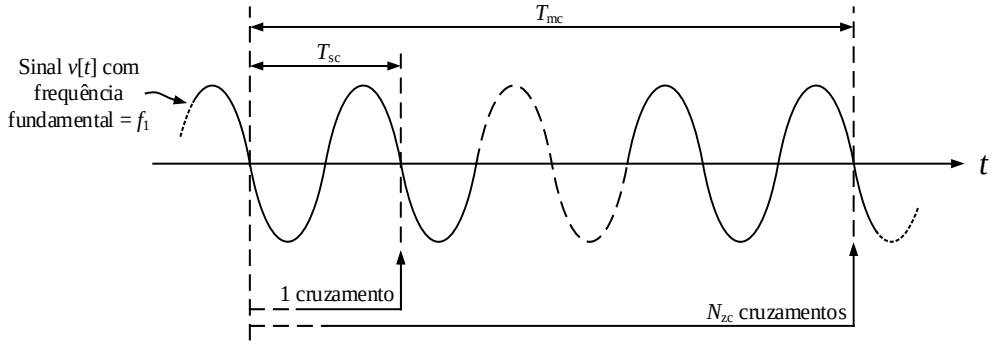


Figura 15: Tempos entre cruzamentos por zero na mesma direção.

Uma estratégia para isso é realizar a multiplicação entre duas amostras consecutivas, sempre que uma amostra nova chegar. Assim, quando o resultado dessa multiplicação der um valor negativo, sabe-se que nesse momento, o cruzamento ocorreu entre a amostra atual e a anterior. Desse modo pode-se estimar o tempo entre dois cruzamentos, ou entre vários.

De posse de T_{sc} ou T_{mc}, uma aproximação $\hat{f}_1$, da frequência fundamental real f_1, do sinal $v[t]$ pode ser calculada em (2.6) ou (2.7), onde T_{sc} é o tempo entre dois cruzamentos consecutivos na mesma direção, e T_{mc} é o tempo entre N_{zc} cruzamentos na mesma direção. Esses períodos são estimados pela contagem dos períodos de amostragem T_s entre os cruzamentos, e sempre será um número inteiro destes.

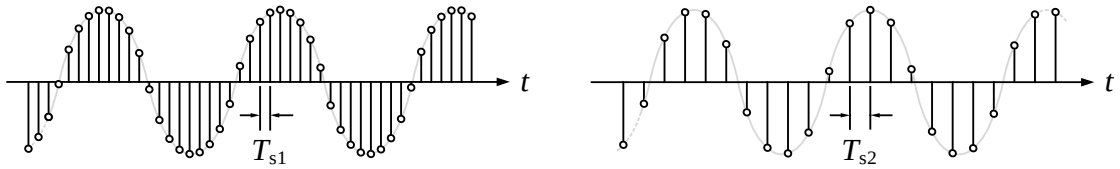
$$\hat{f}_1 = \frac{1}{T_{sc}} \tag{2.6}$$

$$\hat{f}_1 = \frac{N_{zc}}{T_{mc}} \tag{2.7}$$

O primeiro cálculo fornece uma resposta mais rápida, pois, passado um ciclo, já se tem uma estimação. Em contrapartida, o segundo cálculo pode ser mais preciso, pois espera-se vários cruzamentos e faz-se uma média entre eles. Porém, a resposta apresenta um atraso significativo.

Um fato que vale a pena ressaltar é que, em ambos os cálculos, a precisão depende da frequência de amostragem F_s do sinal $v[t]$. Quanto maior for F_s, menor será T_s e, consequentemente, ocasionará um menor erro na medida do instante em que a onda cruza o eixo zero. Isso pode

ser visto claramente através da Figura 16, onde, de a) provém uma melhor estimação que de b), pois T_{s1} é menor que T_{s2}, considerando o mesmo sinal amostrado.



a) b)

Figura 16: Diferentes taxas de amostragens promovem diferentes estimações da frequência fundamental para o mesmo sinal no método por cruzamento pelo zero.

Uma modificação, com um incremento na metodologia, pode ajudar a contornar este problema. Ela consiste em realizar uma interpolação linear entre as amostras anterior e a posterior ao cruzamento. A Figura 17 mostra uma pequena porção de um sinal de frequência fundamental f_1, na qual ocorre o cruzamento, cuja região de transição está em destaque. As amostras $v_c[n–1]$ e $v_c[n]$ são as amostras antes e após o cruzamento, respectivamente. N_a e N_b são as estimações das distâncias relativas entre as amostras e o instante exato do cruzamento (N_a e $N_b < 1$; $N_a + N_b = 1$). Utilizando-se uma interpolação linear, esses valores podem ser estimados, por geometria básica, como mostra as Equações (2.8) e (2.9).

Considerando-se N como o número de amostras entre dois cruzamentos consecutivos, como mostra a Figura 18, e a partir de (2.6), $\widehat{f}_1$ pode ser estimada por (2.10), ainda sem o uso da interpolação.

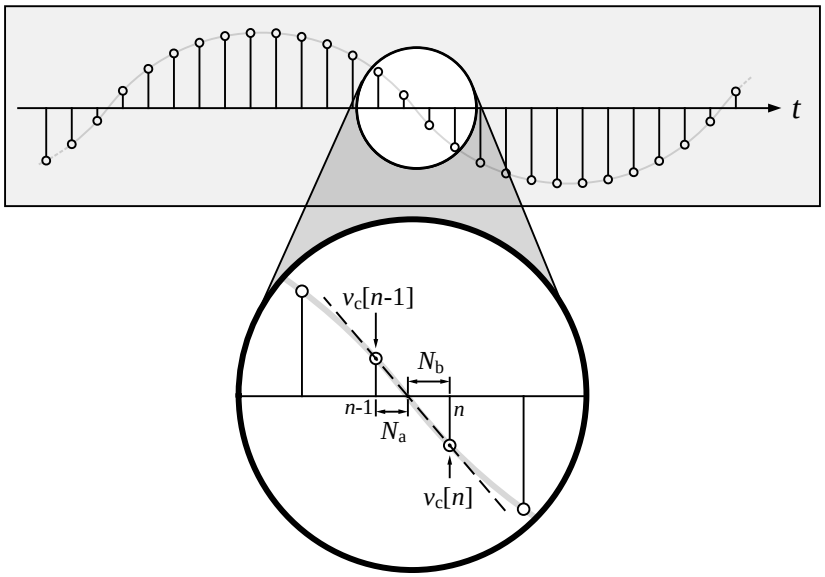


Figura 17: Amostras no cruzamento por zero: interpolação linear

$$N_a = \frac{v[n-1]}{v[n-1]-v[n]} \tag{2.8}$$

$$N_b = \frac{v[n]}{v[n]-v[n-1]} \tag{2.9}$$

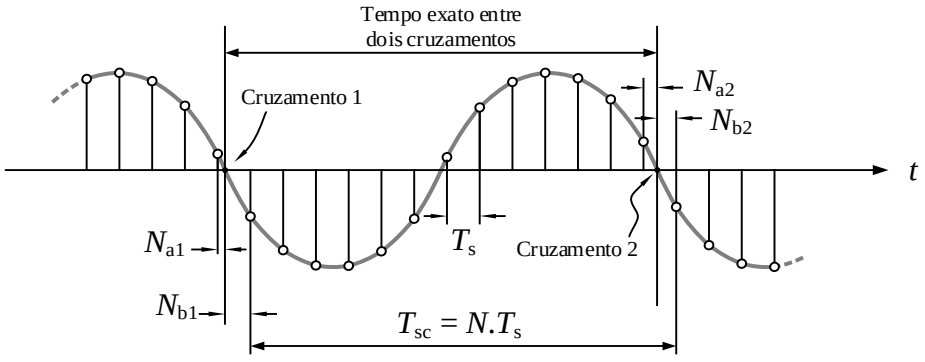


Figura 18: Estimação do período entre dois cruzamentos por zero sem o uso da interpolação.

$$\hat{f}_1 = \frac{1}{T_{sc}} = \frac{1}{N \cdot T_s} = \frac{F_s}{N} \tag{2.10}$$

O intervalo T_{sc} pode ser corrigido, adicionando-se N_{b1} e subtraindo-se N_{b2} antes da multiplicação por T_s. A Figura 19 e a Equação (2.11) mostram como essa correção melhora a estimação da frequência fundamental.

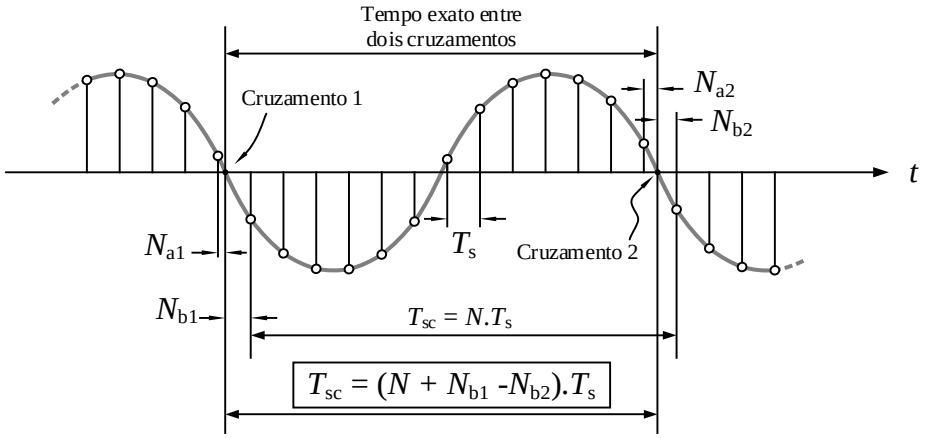


Figura 19: Correção da estimação do período entre dois cruzamentos por zero.

$$\hat{f}_1 = \frac{1}{T_{sc}} = \frac{1}{(N + N_{b1} - N_{b2}) \cdot T_s} = \frac{F_s}{N + N_{b1} - N_{b2}} \tag{2.11}$$

O exemplo a seguir faz uma comparação entre os métodos de estimação da frequência para um mesmo sinal.

Exemplo 2.4

Um sinal v_3, contendo uma queda na amplitude (*Sag*), tem sua frequência fundamental estimada pelos dois métodos apresentados anteriormente. A Figura 20 mostra o sinal v_3, juntamente com as respectivas estimações.

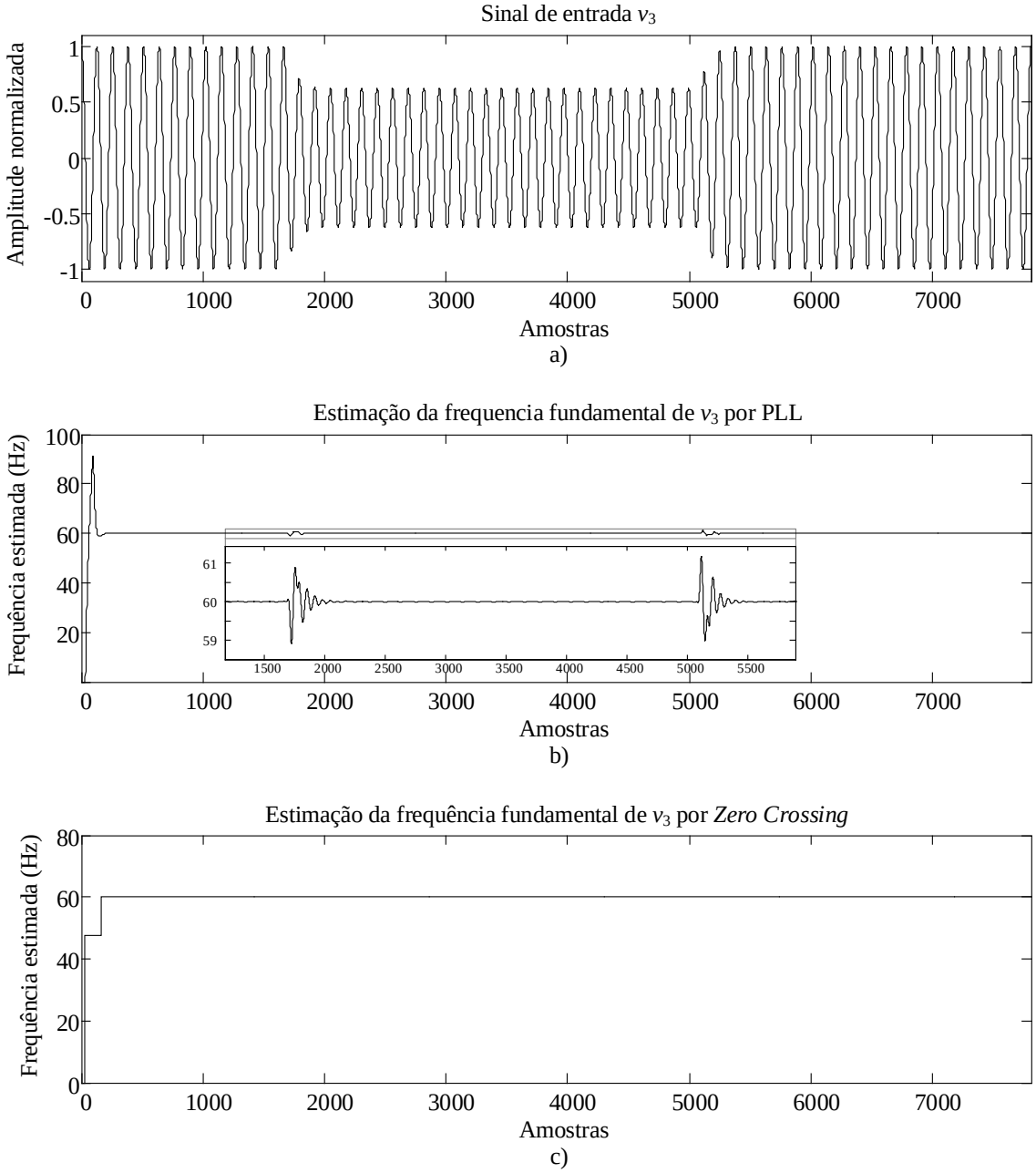


Figura 20: Estimação da frequência fundamental de um sinal contendo um *Sag*. a) Sinal de entrada v_3. b) Estimação da frequência fundamental de v_3 por PLL. c) Estimação da frequência fundamental de v_3 por *Zero Crossing*.

É possível ver que a estimação proveniente do método *Zero Crossing* possui variação mais abrupta no começo, uma vez que o valor estimado é atualizado apenas quando há um novo cruzamento por zero. Através da superposição dos dois gráficos de estimação é possível observar as principais diferenças entre os métodos (Figura 21).

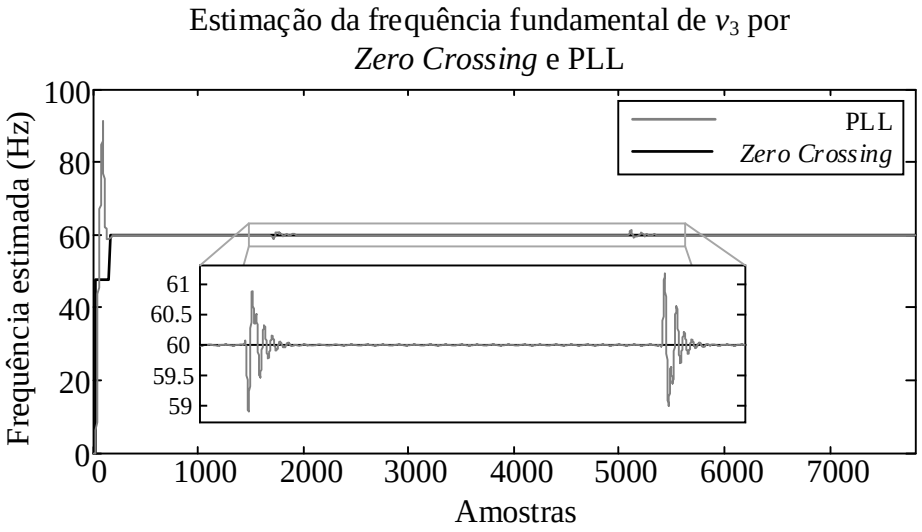


Figura 21: Comparação entre os métodos de estimação de frequência fundamental abordados.

Para a utilização do algoritmo de estimação da frequência pelo cruzamento por zero, necessita-se de acoplar um filtro passa-baixas antes de estimar o cruzamento. Isso se torna necessário a fim de eliminar componentes indesejadas, como harmônicos de alta frequência, que podem prejudicar a estimação. A Figura 22 mostra um possível erro na estimação do intervalo entre dois cruzamentos, caso não seja aplicado o filtro, e o sinal possua harmônicos com amplitudes razoáveis.

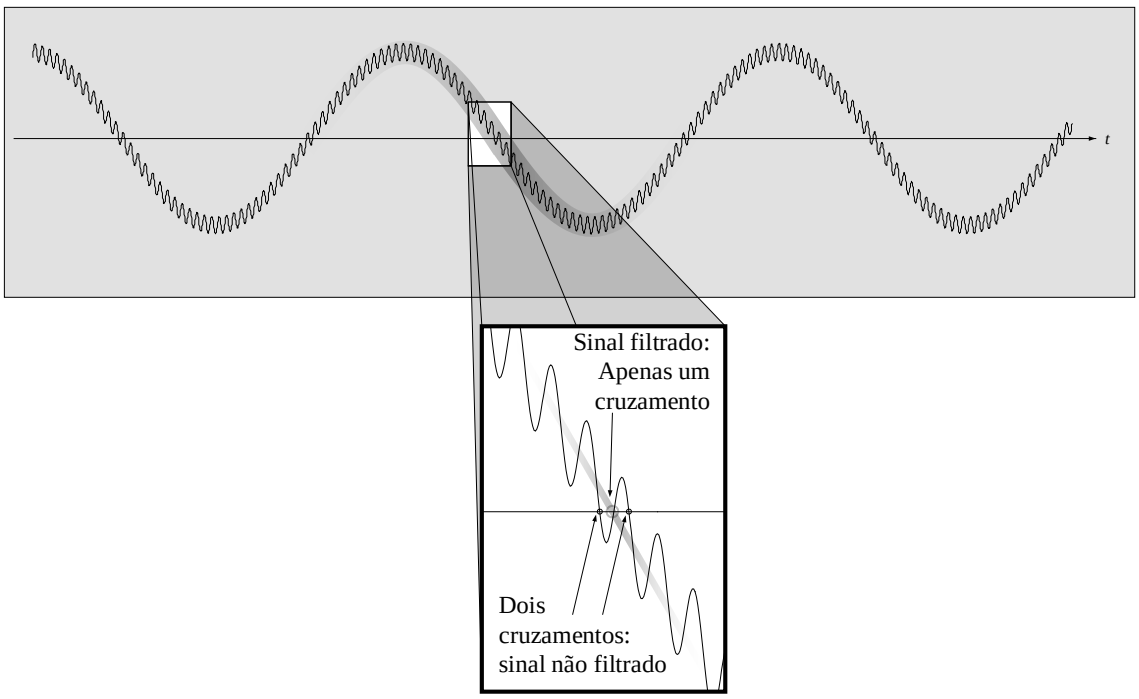


Figura 22: Diferença entre utilização de sinal com aplicação de filtro passa-baixas e sinal sem filtragem, para estimação da frequência fundamental por *Zero Crossing*.

O Exemplo 2.5 mostra um caso onde a aplicação do filtro é imprescindível no uso do método por *Zero Crossing*, comparando com o uso do algoritmo PLL.

Exemplo 2.5

O mesmo sinal v_1 do Exemplo 2.3, contendo um transitório de alta frequência, foi utilizado para comparar os métodos de estimação de frequência fundamental abordados. A Figura 23 mostra essa comparação.

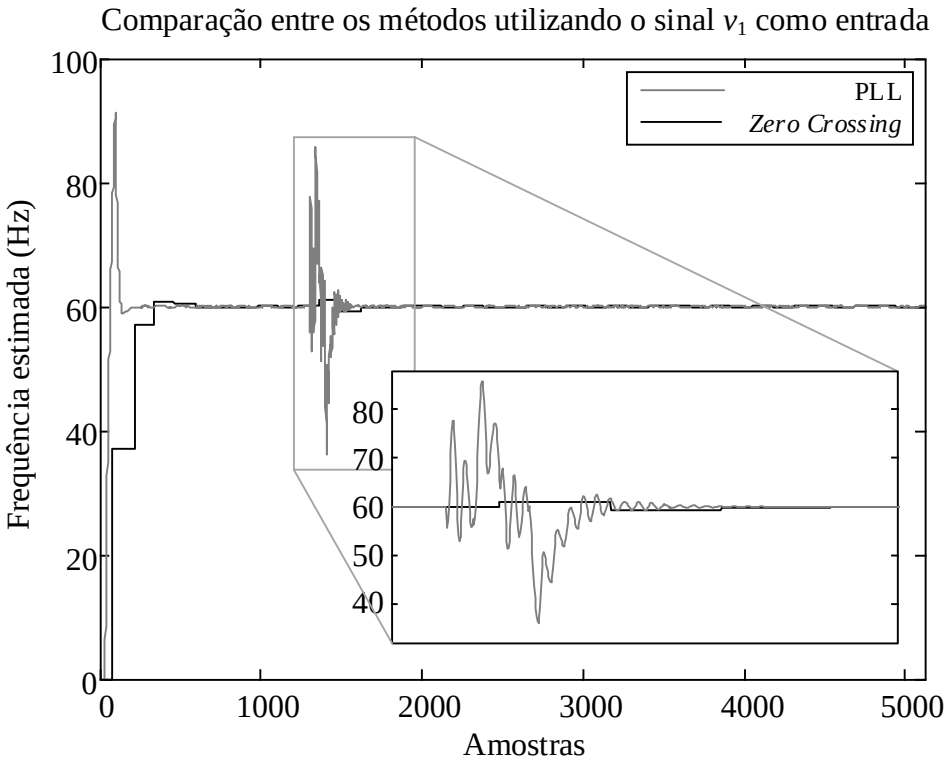


Figura 23: Estimação da frequência fundamental de v_1 utilizando *Zero Crossing* com a aplicação de estágio de pré-filtragem em comparação com o método por PLL.

Pode-se pensar, então, em se utilizar o mesmo filtro passa-baixas, presente no método por *Zero Crossing*, para se tentar obter uma estimação mais estável para o método por PLL. O resultado para tal proposta pode ser visto através da Figura 24.

Pode ser observado que, com a introdução do filtro, obteve-se melhora na estabilidade e na oscilação da estimação pelo PLL. Mas, ainda assim, o método por *Zero Crossing* apresenta um resultado melhor, em termos de sensibilidade à variação de amplitude.

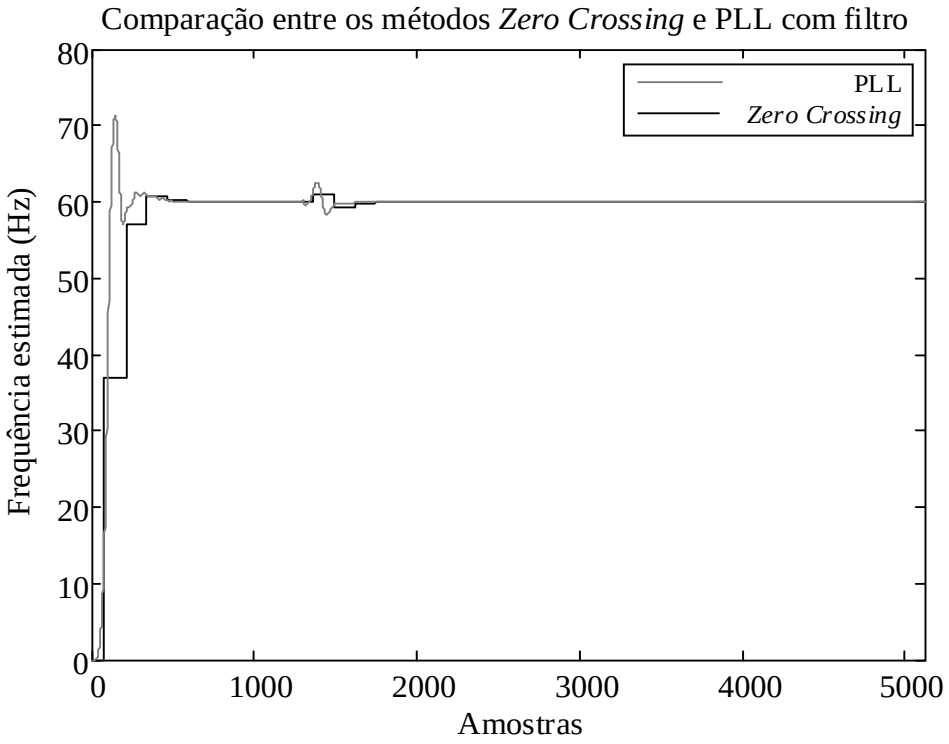


Figura 24: Estimação da frequência fundamental de v_1 utilizando *Zero Crossing* e PLL, ambos com a aplicação de estágio de pré-filtragem.

Após vários testes comparando-se a utilização do algoritmo PLL para estimação da frequência com o método do cruzamento por zero, optou-se pelo uso do segundo, visto que o primeiro apresentava pontos de instabilidades computacionais no decorrer do funcionamento do *hardware* implementado e também, um tempo maior de convergência. Esse comportamento foi justificado devido à presença de realimentação no algoritmo, sendo mais suscetível a instabilidades. Já o segundo método, apresentou-se mais simples de implementar, mais robusto, mais estável e mais rápido.

2.4. Técnicas de detecção de distúrbios elétricos

O conceito para detecção de distúrbios, utilizado neste trabalho, é o conceito de novidade. Como o nome sugere, ela ocorre quando algo novo acontece. A novidade deve estar relacionada a um distúrbio, de qualquer tipo, e quando é identificada, o trecho onde ela ocorreu deve ser armazenado, para posterior reconstrução. Para se identificar uma novidade, é necessário comparar um *frame* anterior com o atual. Por *frame* entende-se o seccionamento ou a divisão do sinal em várias partes, para serem comparadas umas com as outras. Esse conceito de “dividir

para conquistar” pode ser visto em (DUQUE, RIBEIRO, RAMOS, & SZCZUPAK, 2005) e proporciona bons resultados em termos de detecção de novidade, quando aplicado a sinais provenientes de sistemas elétricos de potência.

Após se escolher a divisão do sinal em *frames*, é importante decidir qual o tamanho do *frame*. Para o trabalho proposto, optou-se por utilizar um *frame* com o tamanho de quatro ciclos da frequência fundamental (60 Hz), ou seja, com uma duração de aproximadamente 66,7 ms (Figura 25). Considerando uma taxa de amostragem de 7,68 kHz, por exemplo, cada *frame* possuirá 512 amostras.

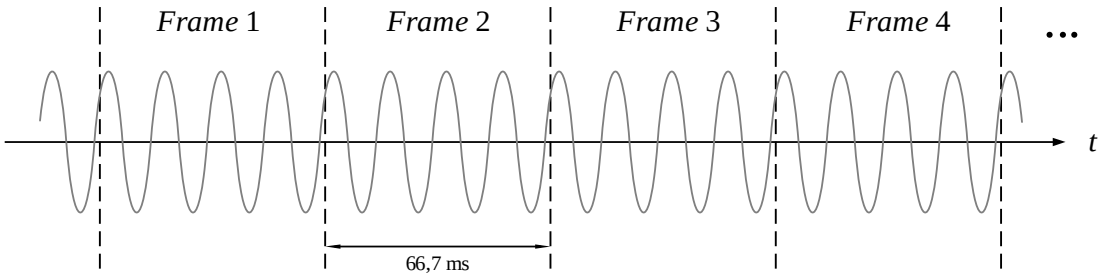


Figura 25: Divisão do sinal em *frames*.

Uma vez definido o tamanho do *frame*, o próximo passo é definir qual parâmetro, inerente ao *frame*, deve ser escolhido de forma a realizar a comparação entre os *frames*, e assim detectar a novidade. Duas propostas foram estudadas: a realização do cálculo do erro médio quadrático entre as amostras dos *frames* e a diferença das energias entre os *frames*.

2.4.1. Erro médio quadrático

Esta técnica é simples e baseia-se no erro médio quadrático entre cada amostra dos *frames* que se deseja comparar. Todas as amostras do primeiro *frame* devem ser armazenadas para que, quando chegar o próximo *frame*, as respectivas amostras sejam subtraídas na ordem correta, uma a uma. As diferenças são então elevadas ao quadrado e acumuladas. O resultado final é então comparado com um limiar. Caso seja maior, é constatada uma novidade e o *frame* novo precisa ser armazenado. Caso contrário, o *frame* pode ser descartado. O *frame* armazenado mais recentemente é considerado como *frame* de referência.

A Figura 26 mostra dois *frames* consecutivos, a partir dos quais se deseja calcular o erro médio quadrático ($EMQ_{1,2}$, na equação (2.12)).

De forma genérica, se for desejado fazer o mesmo cálculo entre dois *frames* A e B quaisquer (Figura 27), mesmo que estes não sejam consecutivos, contendo N_{af} amostras cada um, segue-se o processo descrito pela equação (2.13).

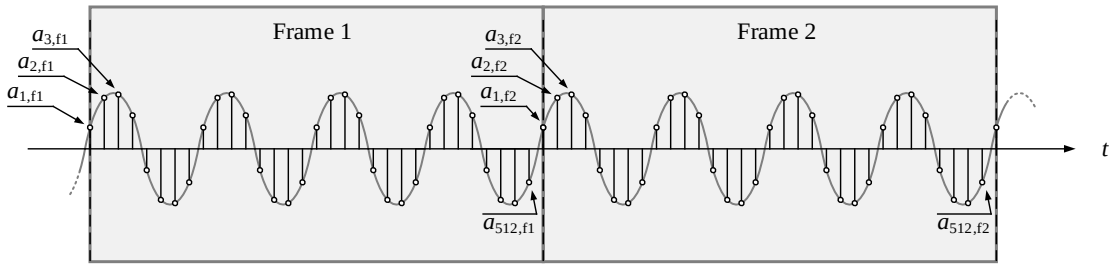


Figura 26: Comparação entre dois *frames* consecutivos a fim de detectar uma novidade.

$$\begin{aligned} \mathrm{EMQ}_{1,2} &= (a_{1,\mathrm{f2}} - a_{1,\mathrm{f1}})^2 + (a_{2,\mathrm{f2}} - a_{2,\mathrm{f1}})^2 + (a_{3,\mathrm{f2}} - a_{3,\mathrm{f1}})^2 + \cdots + (a_{512,\mathrm{f2}} - a_{512,\mathrm{f1}})^2 \\ &= \sum_{i=1}^{i=512} (a_{i,\mathrm{f2}} - a_{i,\mathrm{f1}})^2 \end{aligned} \qquad (2.12)$$

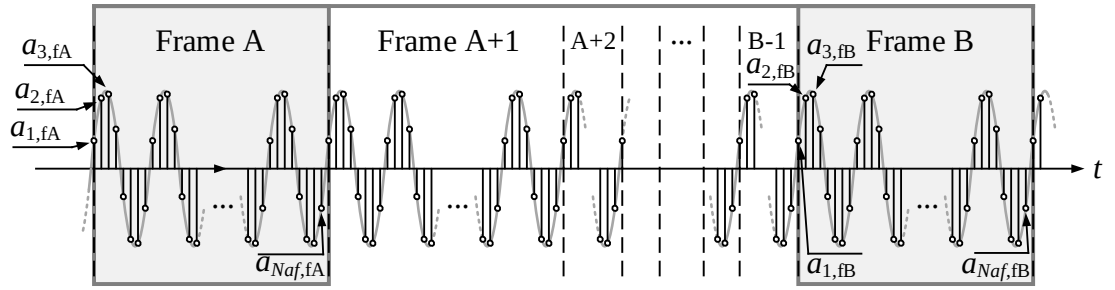


Figura 27: Comparação entre dois *frames* quaisquer a fim de detectar uma novidade.

$$\mathrm{EMQ}_{\mathrm{A,B}} = \sum_{i=1}^{i=N_{af}} (a_{i,\mathrm{fB}} - a_{i,\mathrm{fA}})^2 \qquad (2.13)$$

2.4.2. Diferença de energias

Nesse método, as amostras de um mesmo *frame* são elevadas ao quadrado e somadas, resultando na energia do *frame* (E_A). O mesmo é feito para o outro *frame* (E_B). As energias de cada *frame* são então subtraídas ($E_{A,B}$), é tirado o módulo dessa diferença e o valor resultante é comparado a um limiar. Caso seja maior, o *frame* é considerado como contendo novidade. Caso contrário, ele pode ser descartado. A Equação (2.14) mostra o cálculo da diferença de energias entre dois *frames* quaisquer A e B.

$$E_A = \sum_{i=1}^{i=N_{af}} (a_{i,fA})^2;\ E_B = \sum_{i=1}^{i=N_{af}} (a_{i,fB})^2;\ |E_{A,B}| = |E_B - E_A| \qquad (2.14)$$

A principal diferença entre esse método e o anterior, em termos de memória ocupada no DLP, é que esse não necessita de um *Buffer* para armazenar todas as amostras do *frame* de referência, mas apenas um registrador contendo a soma dos quadrados de um *frame* inteiro.

2.5. Conclusões do capítulo

Neste capítulo foram vistas algumas técnicas de compactação de dados. Métodos de compactação mais complexos podem ser desenvolvidos, utilizando-se mais de uma técnica de compressão, além de outras não vistas aqui. Também foram vistas técnicas estimação da frequência fundamental e de detecção de distúrbios elétricos.

O método de detecção adotado para o trabalho, foi o de comparação entre energias. Escolheu-se este método, pois, de acordo com a metodologia proposta, a detecção desejada neste estágio não deveria ser acionada por desvios de frequência, e o método do erro médio quadrático não se mostrou imune a esse tipo de evento. Além disso, esse método foi mais sensível à presença de ruídos, indicando novidades, quando, na verdade o sinal estava ligeiramente ruidoso. Em contrapartida, o método por diferenças de energia se mostrou mais imune, tanto à variação de frequência, quanto à presença de ruído.

Finalmente, vale ressaltar que a sensibilidade do sistema de detecção é de acordo com o nível do limiar a ser comparado. Caso seja desejado que o sistema indique novidades em todos os *frames*, basta utilizar um limiar nulo.

3. FPGA

Em conformidade com o que foi descrito no Capítulo 2, o Dispositivo Lógico Programável (DLP) utilizado para este trabalho foi a FPGA (*Field-Programmable Gate Array*). Este capítulo tem por objetivo, mostrar um pouco sobre este dispositivo, passando pelos seus elementos básicos, sua linguagem, sua programação e implementação.

Sendo assim, na Seção 3.1 é introduzido este tema, na Seção 3.2 é apresentada a arquitetura básica da FPGA, na Seção 3.3 é visto um pouco sobre o projeto e a programação da mesma. Já na Seção 3.4, é mostrada uma Linguagem Descrição de *Hardware* (HDL), com a qual foram desenvolvidos os circuitos digitais desta dissertação. A Seção 3.5 mostra formas de implementação destes circuitos em FPGA. Por fim, a Seção 3.6 traz conclusões a respeito do capítulo.

3.1. Introdução

A FPGA é um dispositivo lógico programável baseado em memória RAM (*Random Access Memory*). Como o nome sugere, este dispositivo é, resumidamente, um conjunto de portas lógicas programáveis em campo (Meyer-Baese, 2013). Ele é disposto em forma de matriz bidimensional, cujas posições são preenchidas por elementos lógicos (LE – *Logic Elements*), que formam a unidade básica de lógica programável. A FPGA é, portanto, em senso amplo, uma matriz de elementos lógicos (Figura 28), conectados uns nos outros. Além dos elementos lógicos, existem outros elementos participantes, como os blocos de entrada e saída, blocos de memória dedicada, malha de interconexão, dentre outros.

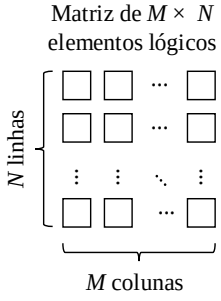


Figura 28: Representação da matriz de elementos lógicos numa FPGA.

O termo “programável em campo” denota que a FPGA, além de poder ser programada por uma quantidade ilimitada de vezes, pode ter o seu *firmware* atualizado, mesmo que o *chip* faça parte de um produto que já esteja instalado em campo. E mais do que isso, esse dispositivo é capaz de possuir um *hardware* dinamicamente reconfigurável (*on-the-fly*) (Bhandari & Pujari, 2010). Em outras palavras, isso significa que o mesmo *chip* pode progredir para diferentes topologias e arquiteturas de circuitos digitais, durante o funcionamento, uma vez projetado para isso. Inclusive, este fator é um forte apelo de *marketing* por parte das empresas fabricantes, embora não seja, ainda, a característica mais explorada deste poderoso dispositivo.

As primeiras FPGAs, comercialmente disponíveis, surgiram em meados da década de 80, lançadas pela XILINX® (Joost & Salomon , 2005). O primeiro *chip*, XC2064, possuía apenas 64 blocos lógicos configuráveis, cada um com duas tabelas verdades de três entradas (MORAES & Calazans, Parte 2 - Introdução a FPGAs e Prototipação de Hardware, 2014). Com o passar do tempo, outros fabricantes foram adquirindo *know how* nessa área, dentre os quais pode-se destacar a ALTERA®. As FPGAs das famílias Cyclone e Stratix, ambas da ALTERA®, foram introduzidas em 2002. Hoje, o mercado no ramo de FPGA está dividido, principalmente, entre essas duas grandes companhias, que são as principais fabricantes de *chips* e *kits* de desenvolvimento, principalmente para projetos de pesquisa (Ayodele, Inyang, & Kehinde, 2015), (Yildiz, Cesur, Kayaer, Tavsanoglu, & Alpay, 2014) e (WANG, XIANG, & JINGMING, 2014). Outras marcas como LATTICE® e Microsemi®, podem, também, ser lembradas.

O gráfico apresentado na Figura 29 mostra a divisão desse mercado, através das receitas das principais empresas no ano de 2010.

Com o passar dos anos, as FPGAs vieram ganhando força e presença cada vez mais crescentes em produtos comerciais (PUTNAM, et al., 2014). Em decorrência desse fato, os investimentos em novas tecnologias, por parte dos fabricantes, têm se tornado mais intensos, a fim de agregar mais recursos às FPGAs. Atualmente, elas possuem outros elementos internos, tais como blocos de memória SRAM dedicados e configuráveis, *transceivers* de alta velocidade, blocos de gerenciamento de *clock*, como PLLs (*Phase-Locked Loops* – ALTERA®) e DCMs (*Digital Clock Manager* – XILINX®), elementos multiplicadores, dentre outros, que ampliam o horizonte de desenvolvimento. Dessa forma, os *chips* modernos estão mais robustos, com maior capacidade, e permitem que os níveis de implementação de *hardware* sejam levados a patamares cada vez mais elevados (HAMAD, AL-QUTAYRI, SALAH, SCHINIANAKIS, &

STOURAITIS, 2015). Além disso, pesquisadores têm se empenhado para ampliar os alcances do uso das FPGAs (GREAVES & SINGH, 2008), (ROSE, et al., 2012).

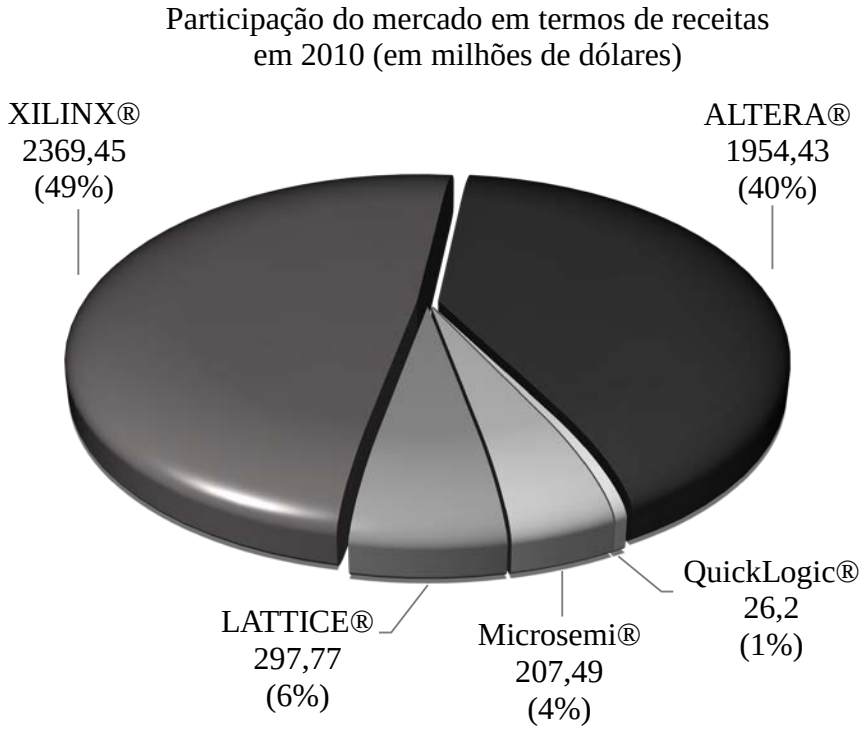


Figura 29: Divisão do mercado de FPGAs no ano de 2010 (Johnson, 2011).

Dentro do âmbito da importância que a FPGA possui atualmente, alguns fatores que justificam tal posição podem ser considerados, dentre os quais pode-se destacar:

- Alta reconfigurabilidade: a estrutura interna da FPGA, constituída de uma arquitetura altamente granularizada, proporciona uma grande flexibilidade em termos de lógica digital customizada, abrangendo todo e qualquer tipo de circuito digital que se deseja implementar. Assim, a FPGA é uma plataforma de *hardware* reconfigurável e altamente modelável.

- Alta densidade de lógica: existem *chips* de FPGA contendo, desde milhares de elementos lógicos (Cyclone IV, Device Handbook, 2014), até milhões deles num mesmo encapsulamento (STRATIX V, 2015), (ALTERA®, Stratix 10 FPGAs and SoCs: Delivering the Unimaginable, 2015). Quanto maior for a densidade, maior a granularidade do *chip*, podendo assim, admitir implementações mais ricas e complexas.

- Alta velocidade de operação: atualmente, existem FPGAs que suportam *clocks* que podem atingir taxas com ordem de Giga-hertz (Virtex-6, 2012).

- Capacidade de paralelismo: implementação de circuitos digitais independentes, sendo utilizados simultaneamente e, até mesmo, com diferentes taxas de *clock*.

- *Time to Market*: devido à capacidade de uma rápida implementação do *hardware* projetado, realizações de testes, simulações e identificação de erros para devidas correções, diminui-se o tempo despendido para o desenvolvimento e finalização de um produto comercial.

- Facilidade de sintetização de *hardware*: é possível utilizar linguagens de alto nível de abstração para se desenvolver projetos de descrição de *hardware* em FPGA, com o auxílio de programas sintetizadores. Dentre elas, destacam-se VHDL e Verilog. Essa flexibilidade auxilia também, no projeto de ASICs (*Application Specific Integrated Circuits*).

Algumas desvantagens da FPGAs, quando comparadas a outros processadores com os DSPs, são:

- Custo: apesar de *Chips* e *Kits* de desenvolvimento estarem sendo fabricados e disponibilizados comercialmente com um preço de aquisição cada vez mais acessível, as FPGAs ainda possuem maior custo.

- Complexidade de implementação: o tempo despendido para se implementar um algoritmo em um processador é muito menor do que na FPGA, devido principalmente, à possibilidade de realização de processo de *debug*.

Por causa de sua grande flexibilidade, FPGAs podem ser utilizadas tanto para a implementação de um circuito digital dedicado, como para embarcar processadores customizados, incluindo os circuitos de apoio e de interface (SILVA C. E., 2011). Tudo isso, obviamente, dependendo de capacidade da FPGA.

A FPGA pode ser utilizada a fim de implementar qualquer função lógica que um Circuito Integrado de Aplicação Específica (ASIC) pode realizar. Porém, a capacidade de possibilitar a atualização de suas funcionalidades, mesmo depois de inserida num sistema pronto e funcional, oferece vantagens para diversas aplicações, inclusive para a compactação de sinais.

Enfim, a utilização da FPGA pode ser encontrada na literatura em diversas aplicações. Em sistemas de processamento de sinais (Du, Luo, & Wang, 2011), sistemas de medição e monitoramento de parâmetros de qualidade de energia (Femine, Gallo, Landi, & Luiso, 2009), (Ferrigno, Landi, & Laracca, 2008), registradores de falta em linhas de transmissão (Qiong & Zhao-Hui, 2011), detectores de distúrbios elétricos (Choong, Reaz, Sulaiman, & Mohd-Yasin, 2005), classificadores de eventos (Finker, del Campo, Echanobe, & Martínez , 2014), e além destes, outros trabalhos envolvendo outras áreas acadêmicas e científicas (Marinkovic, Gillette, & Ning, 2005), (Wang, Yan, Xu, Wang, & Hsu, 2015). Esses exemplos mostram um pouco da vasta aplicabilidade deste poderoso dispositivo.

3.2. Arquitetura básica

As FPGAs possuem três elementos básicos: blocos Lógicos Configuráveis (CLBs), interconexões programáveis e blocos de entrada e saída (IOBs). A Figura 30 mostra uma representação destes três principais elementos.

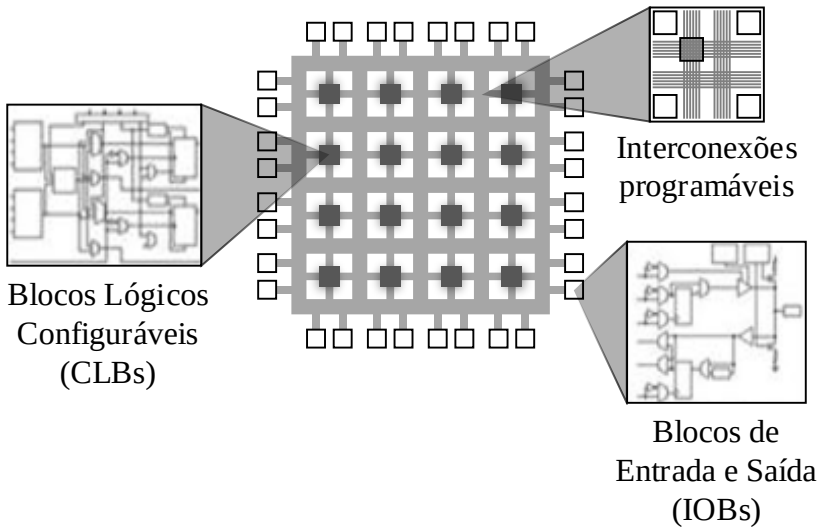


Figura 30: Elementos básicos de uma FPGA.

3.2.1. CLB (*configurable logic blocks*)

Os Blocos Lógicos Configuráveis (CLBs), são responsáveis pela implementação de toda a lógica presente num projeto de *hardware* em FPGA. Esses blocos estão divididos em três

subgrupos: os elementos lógicos (LEs), os blocos de DSP (multiplicadores embarcados) e blocos de memória RAM dedicada.

- **Elementos lógicos (LE)**

O elemento lógico é capaz de implementar uma pequena lógica combinacional, com a opção de se registrar a saída. A lógica é implementada através de uma tabela verdade (*Look-up Table* - LUT), e o registro opcional, por um *Flip-Flop* do tipo D, presente após a implementação da LUT.

O *Flip-Flop* possui alguns controles auxiliares, como *enable*, *clear* e *reset* síncronos ou assíncronos. Além disso, o *clock* do *Flip-Flop* vem de uma entrada dedicada, proveniente da rede de *clock* do *chip*, que por sua vez está conectada à PLL utilizada para geração de *clock*.

O elemento lógico é otimizado para a arquitetura *pipeline* (Sun, Wirthlin, & Neuendorffer, 2007), na qual, o caminho percorrido pelo fluxo de dados é sempre seccionado em várias partes, por registradores, a fim de possibilitar mais altas taxas de *clock* de operação.

A Figura 31 mostra uma representação simplificada de um elemento lógico presente numa FPGA pertencente à família Cyclone IV da ALTERA®. Na figura, podem ser vistas as partes combinacional (LUT) e a parte registrada (registrador tipo D) do LE, que podem ser usadas de forma independente. A LUT apresentada na figura, possui 4 entradas. As entradas e saídas de *Carry* e da cadeia de registradores dos LEs, permitem a implementações de lógicas e de registros maiores do que as de apenas um LE. Além disso, existe a possibilidade de realimentação do registrador para a LUT dentro do próprio LE. Esses recursos foram criados a fim de proporcionar um melhor desempenho do circuito e otimização de *fitting*. A saída do LE se comunica com a malha de interconexões configuráveis.

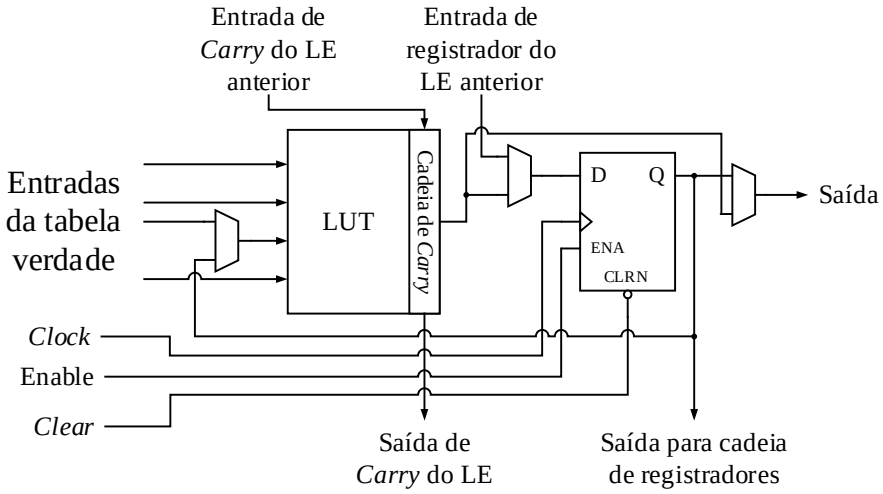


Figura 31: Representação simplificada de um Elemento Lógico (LE).

Dentro da FPGA, os LEs são agrupados em conjuntos de 16 LEs, na forma de coluna. Cada conjunto é um Bloco de Arranjo Lógico - LAB (*Logic Array Block*). Cada LAB possui alguns recursos, como sinais de controle, cadeia de *Carry*, cadeia de registradores e interconexões locais. Os LABs servem para otimizar a combinação de LEs em sínteses de lógicas mais complexas. Um esquema de um LAB e suas conexões locais são mostrados posteriormente nas Figuras 36 e 37.

- **Blocos DSP**

Esses blocos são voltados para aplicações que envolvem Processamento Digital de Sinais (DSP), nas quais estão presentes muitas operações de multiplicação, soma, subtração e acumulação. A Figura 32 mostra um bloco DSP multiplicador do mesmo *chip* cujo LE foi mostrado anteriormente. Esse bloco está representado com a configuração que admite duas entradas de até 18 bits cada. Nela é possível ver os sinais de controles, as entradas, a saída e o núcleo de multiplicação, com opções de registro. As entradas de dados A e B podem, ou não, ser sinalizadas, de forma independente uma da outra, conforme os controles “signa” e “signb”.

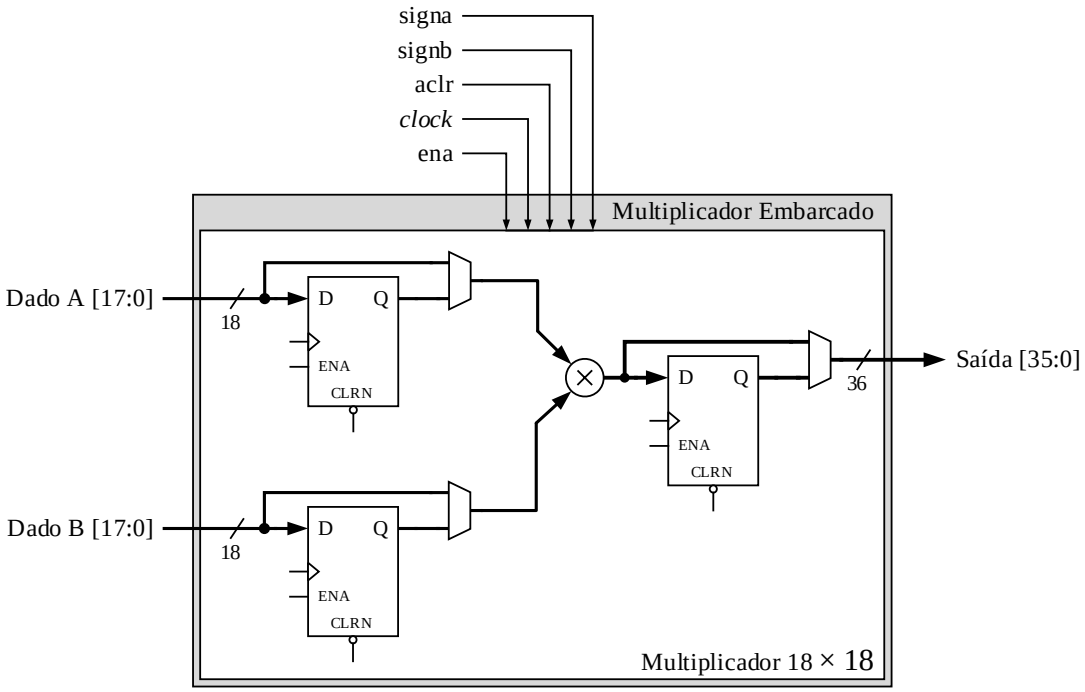


Figura 32: Bloco DSP.

Para multiplicações de palavras de dados com larguras maiores do que suportam as entradas, mais de um bloco DSP é conectado em cascata. Não há restrição quanto à largura da palavra que se deseja multiplicar, obedecendo, obviamente, a capacidade do dispositivo. Porém, quanto maior for a largura, mais lento será o processo final de multiplicação (Cyclone IV, Device Handbook, 2010).

Os elementos DSP estão dispostos também em colunas, entre os LABs, de forma a obterem melhor integração com o circuito (Figura 33).

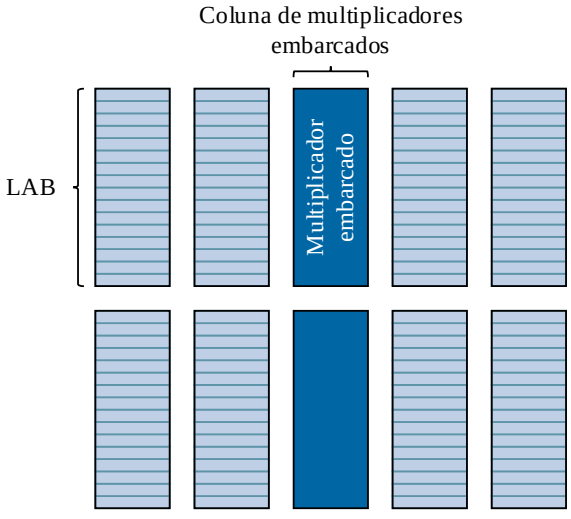


Figura 33: Representação da coluna de multiplicadores entre os elementos lógicos na FPGA.

- **Blocos de memória**

A estrutura de memória embarcada também consiste em colunas formadas por blocos de memória entre os LABs. Eles possuem registradores de entrada que sincronizam a escritas e registradores de saída para projetos em *pipeline*, a fim de melhorar a performance do sistema (Cyclone II, 2008).

Os blocos de memória podem ser configurados de várias formas a fim de cobrir diversas funções de memória, como memória RAM, *shift registers*, memória ROM e FIFO *buffers*.

Exemplo 3.1

As memórias contidas nas FPGAs da família Cyclone IV, por exemplo, são divididas em blocos de 9 Kbits (M9K)[4] e, além de poderem operar com largura de barramento configurável, possuem diversos modos de operação:

- Memória *single port*;
- Memória *dual port* simples;
- *True dual port*;
- *Shift register*[*];
- ROM – com conteúdo inicial programável (*memory initialization file* .mif);
- FIFO[*].

Operações de *shift register* são utilizadas, principalmente, para aplicações de Processamento Digital de Sinais (DSP), como implementação de filtros FIR. Tais aplicações, exigem o armazenamento local de dados, tradicionalmente realizados com *Flip-Flops*, o que, rapidamente, consome muitas células lógicas da FPGA para grandes registradores de deslocamento. Uma forma mais eficiente de implementação é utilizando as memórias embarcadas.

---

[4] Especificamente 8192 bits de memória por bloco (9216 bits por bloco incluindo os bits de paridade, para fins de detecção de erros de leitura/escrita).

[*] Essas configurações utilizam lógica adicional da FPGA.

A Figura 34 mostra o modo de operação *shift register*, numa FPGA da família Cyclone IV, para um registrador de deslocamento de tamanho $w \times m \times n$, onde w é a largura da palavra a ser deslocada, m é o comprimento de cada *tap*, e n é o número de *taps*.

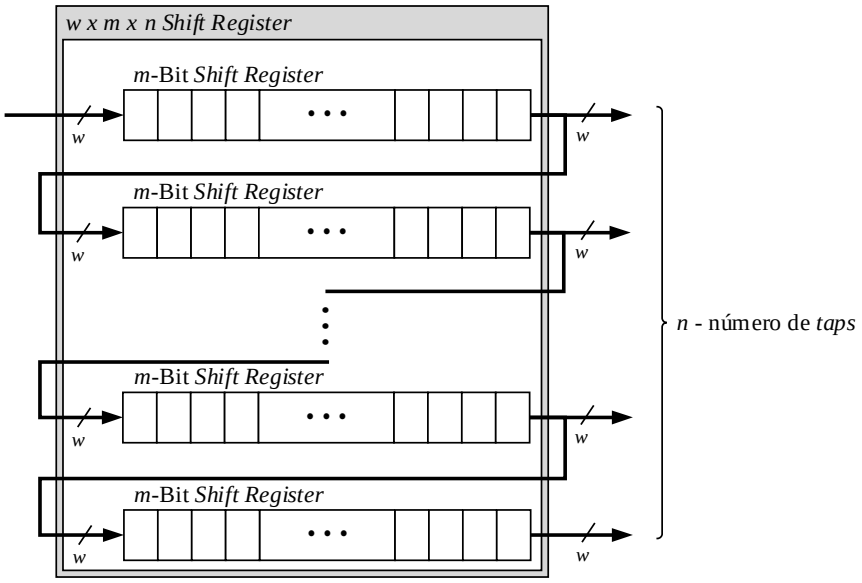


Figura 34: Representação da memória da FPGA configurada como *shift register*.

3.2.2. Elementos de entradas e saída (IOEs)

Os elementos de entrada e saída (*Input/Output Elements* - IOEs) são responsáveis pela comunicação e interface entre os CLBs e o mundo externo. Eles possuem alta flexibilidade, incluindo diferentes tipos de lógica, inclusive lógica diferencial. Além disso, possuem recursos de *pull-up*, *pull-down* e *tri-state*. Esses recursos foram desenvolvidos a fim de se reduzir o uso de componentes externos, como resistores de *pull-up* e diodos (Cyclone IV, Device Handbook, 2014). Os IOEs podem ser utilizados como pinos de entrada, pinos de saída e pinos bidirecionais. Cada elemento possui um *buffer* bidirecional, registradores de entrada (*input register*), registradores de saída (*output register*) e registradores de *enable* para saída (OE *register*). De acordo com a família a qual a FPGA pertença, o número desses registradores pode variar. Os IOEs são agrupados em blocos de entrada e saída (IOBs).

A Figura 35 mostra um diagrama simplificado de um IOE.

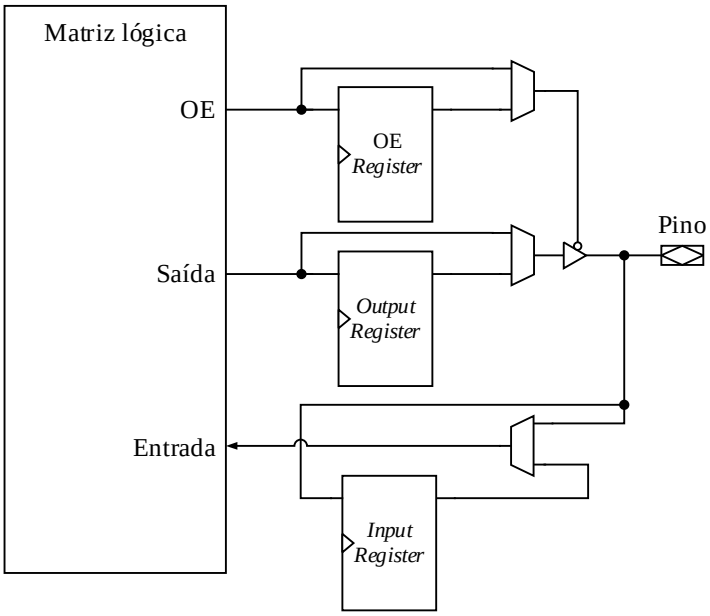


Figura 35: Representação simplificada de um bloco de entrada e saída.

3.2.3. Malha de interconexões programáveis

As interconexões programáveis permitem que os elementos da FPGA sejam conectados entre si a fim de implementar a lógica desejada. Por ela passa a rede de distribuição de *clock* e todos os sinais de controle necessários para o funcionamento do sistema. A malha está disposta em forma de linhas e colunas de interconexão. Existem interconexões globais e locais. Cada LE se comunica a outro LE do mesmo LAB por meio de interconexões locais. Para comunicações de elementos mais distantes, existem as interconexões globais. Dessa forma, busca-se a otimização de performance, flexibilidade e eficiência.

A Figura 36 mostra a representação de uma parte da malha de interconexões, na qual dois LABs vizinhos estão em evidência. Nela, as setas negras mostram as possibilidades de interconexão existentes.

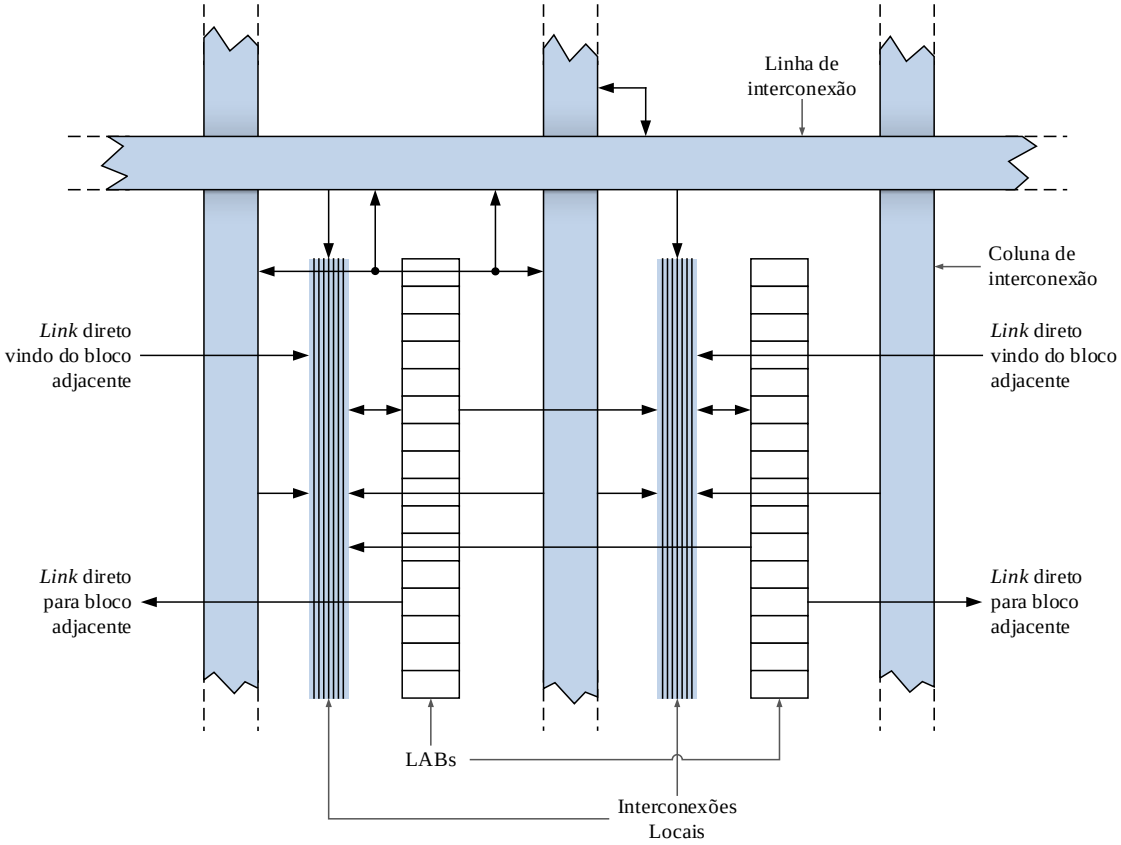


Figura 36: Representação da malha de interconexão com dois LABs vizinhos.

As interconexões locais são alimentadas pelas linhas e colunas de interconexão, bem como pelas saídas dos LEs do mesmo LAB. Outros elementos vizinhos, como PLLs, blocos de memória, multiplicadores embarcados e os LABs mais próximos, podem, também, ter acesso à interconexão local através de um *link* direto. Esses *links* diretos minimizam a utilização das linhas e colunas de interconexão, a fim de melhorar a performance e a flexibilidade.

A Figura 37 mostra uma interconexão local com o acesso aos links diretos pelos elementos próximos ao LAB em destaque. As setas negras mostram as possibilidades de conexão.

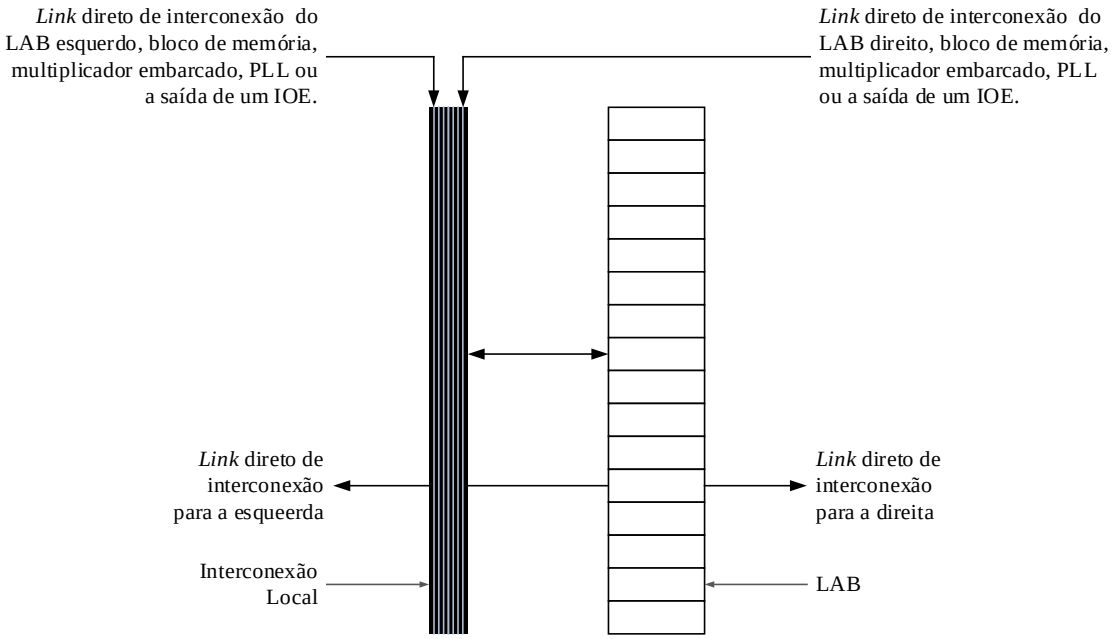


Figura 37: Interconexão local com os *links* diretos.

O exemplo a seguir mostra duas representações dos elementos básicos, vistos até aqui, em dois *chips* de FPGA da ALTERA®.

Exemplo 3.2

Um exemplo da disposição dos elementos básicos da arquitetura de uma FPGA pode ser visto através da Figura 38. Ela representa a estrutura interna simplificada da arquitetura de um *chip* da família Cyclone II, da ALTERA®. Nela aparecem os IOEs, os multiplicadores, os blocos de memória, e também são apresentadas as PLLs, responsáveis pela geração de *clocks* do sistema (Cyclone II, 2008).

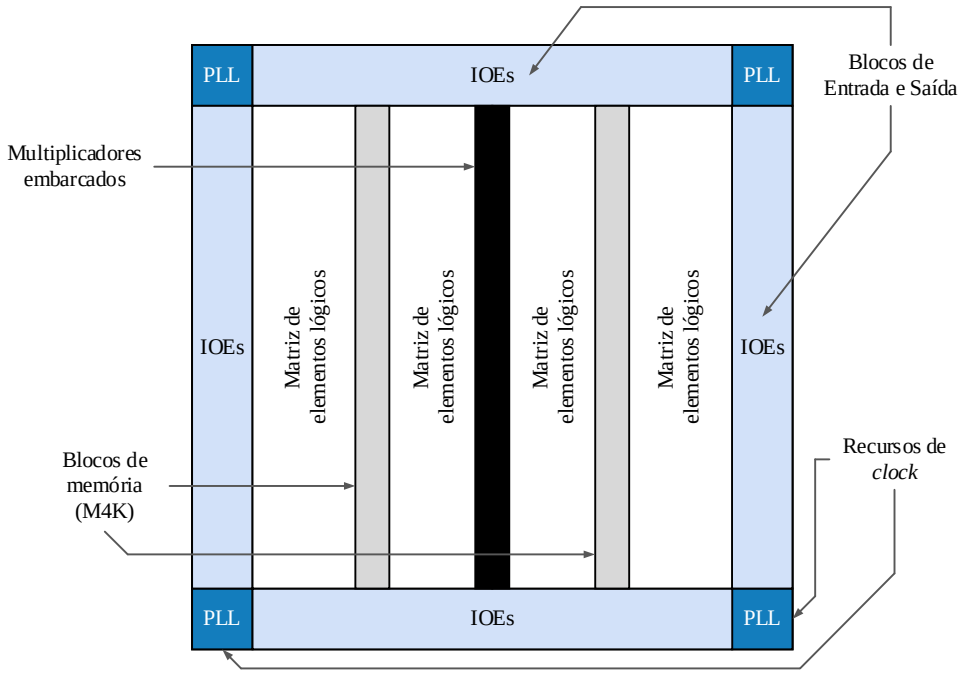


Figura 38: Representação simplificada dos elementos básicos num *chip* Cyclone II EP2C20, da ALTERA®.

Outro exemplo descrevendo mais detalhes da distribuição dos elementos dentro da FPGA pode ser observado através da Figura 39, que mostra a arquitetura da família Cyclone IV da ALTERA® (Cyclone IV, Device Handbook, 2014). Nela também podem ser vistos outros elementos, como os *transceivers* para comunicação serial em alta velocidade e elementos de interface com memória externa e com barramento PCI *Express*.

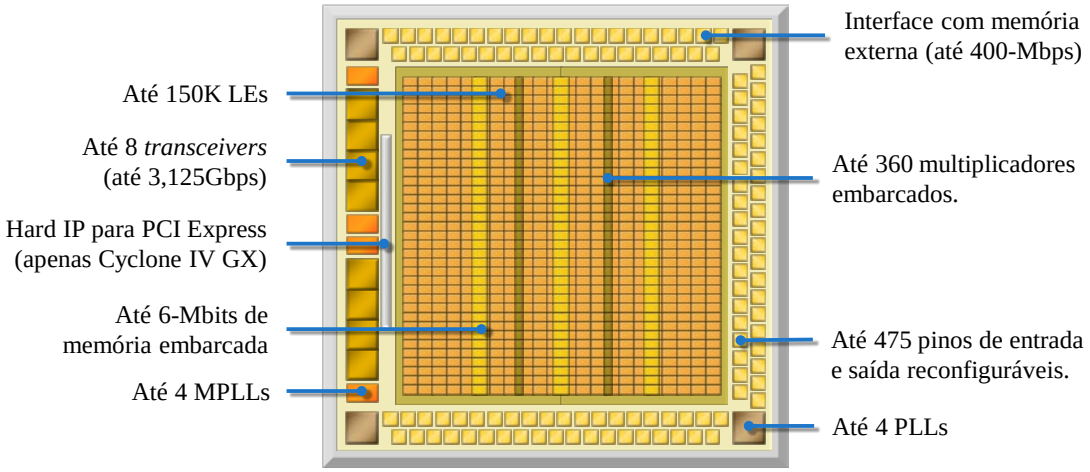


Figura 39: Representação simplificada dos elementos básicos num *chip* da família Cyclone IV, da ALTERA®.

3.3. Projeto e síntese de circuitos em FPGAs

Num projeto de *hardware* em FPGA, existem algumas etapas a serem seguidas, a fim de se obter maior garantia de sucesso ao final da implementação. Essas etapas são procedimentos básicos, que ajudam na identificação de erros e na correção dos mesmos (*debug process*). Eles se estendem desde a concepção da ideia do projeto, passam pela criação do código de descrição, simulações e, finalmente, chegam à programação do *chip* propriamente dita, juntamente com os testes. A Figura 40 mostra um diagrama de blocos, no qual um fluxograma apresenta as etapas a serem seguidas (ALTERA®, My First FPGA Design Tutorial, 2008).

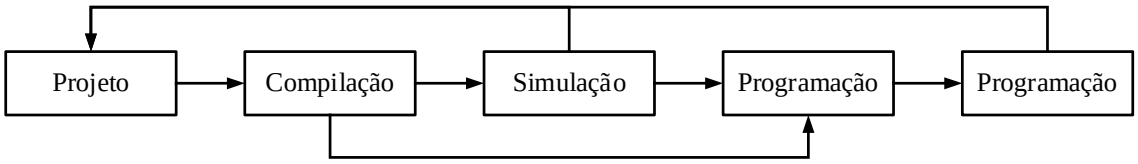


Figura 40: Fluxo de projeto de *hardware* em FPGA.

A etapa inicial é o desenvolvimento de cada bloco de *hardware*. Isso pode ser feito através de um arquivo esquemático, um diagrama de estados, ou através de uma Linguagem de Descrição de *Hardware* (HDL). A linguagem adotada para o projeto deste trabalho foi a linguagem Verilog.

Após terem sido criados os módulos em Verilog, a próxima etapa é a compilação. Nessa etapa o compilador irá gerar um arquivo com todas as informações necessárias para sintetizar o *hardware* desejado.

O próximo passo é o processo de simulação funcional, este é o período onde um simulador é utilizado para a execução do projeto, para confirmar as saídas com diversas entradas de teste. Caso haja algum erro, volta-se aos códigos em Verilog e corrige-se o problema.

Feita a compilação, um *bitstream* é criado em uma etapa denominada *fitting*, ou seja, esta etapa determina quais são os dados binários que deverão ser carregados à FPGA para fazer com que o *chip* execute o projeto.

Uma vez verificado e corrigido, o *hardware* já está pronto para ser programado, não necessitando de novas simulações. Por fim, testa-se o *hardware* em ambientes reais para

verificar se ele se comporta como o esperado. Caso ainda exista algum erro, este deve ser corrigido no projeto. Nova compilação deve ser feita, seguida de nova programação e novos testes de comprovação devem ser executados.

Geralmente, as empresas que desenvolvem os DLPs, fornecem também programas capazes de gerar e carregar o *bitstream* no *hardware*, utilizando as linguagens padronizadas de descrição de *hardware* (Fernandes, Conceitos básicos de um FPGA, 2014). O *software* utilizado para este trabalho foi o Quartus II®, da ALTERA®.

A programação da FPGA é volátil, por isso é perdida sempre que desenergizada, e portanto, necessita de um circuito externo de *booting*, de forma a carregar todo o *firmware* para dentro do *chip*. A comunicação e a transferência dos dados de um projeto de *hardware* para o *chip*, no *kit* de desenvolvimento, podem ser realizadas através de uma interface USB entre um computador e a placa que contém a FPGA.

No caso de funcionamento *stand-alone*, usa-se uma memória não volátil externa.

3.4. Verilog – HDL e conceitos básicos

Uma Linguagem de Descrição de *Hardware* (HDL) se difere de uma linguagem de programação de *software*, não apenas por detalhes de sintaxe, mas sim, e principalmente, no que diz respeito ao conceito que está por trás de cada uma delas.

Uma linguagem de programação de *software* é utilizada na elaboração de um conjunto de instruções que serão compiladas a um nível de máquina para a execução de um processador genérico. Nela descreve-se a sequência de operações ou instruções que o processador deverá executar.

Em HDL, apesar de serem usadas estruturas parecidas com as de linguagem de programação de *software*, o intuito é simular e sintetizar um *hardware*. Entende-se por sintetizar um *hardware*, como a elaboração de um conjunto de instruções que serão interpretadas a fim de construir um *hardware* dedicado, que responda aos estímulos de forma esperada.

Uma comparação entre esses dois tipos de linguagem pode ser vista pela Figura 41.

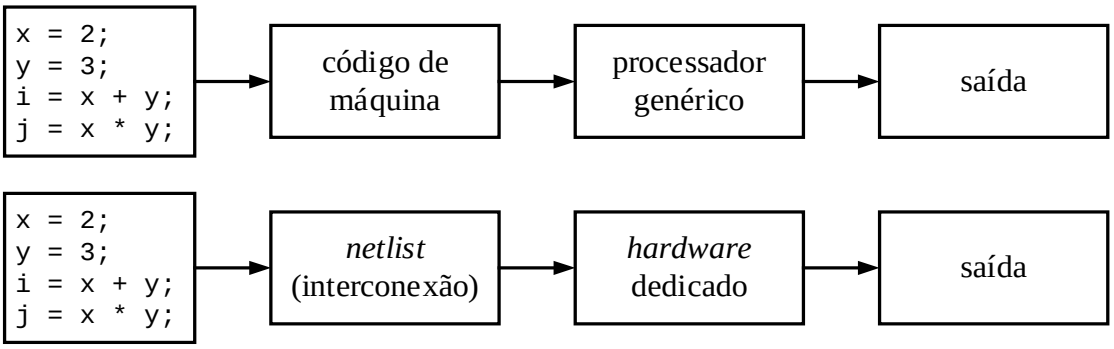


Figura 41: Comparação entre as linguagens de programação de *software* e de descrição de *hardware*.

Atualmente, existem duas principais linguagens de descrição de *hardware*: VHDL e Verilog. Elas se destacam, dentre outros motivos, devido à complexidade do *hardware* que são capazes de descrever.

3.4.1. Breve histórico – VHDL e Verilog

A linguagem VHDL teve início no contexto do programa americano "*Very High Speed Integrated Circuits*" (VHSIC), iniciado em 1980, daí o seu nome (VHSIC *Hardware Description Language*). O desenvolvimento dessa linguagem foi motivado pela necessidade de um padrão para o intercâmbio de informações entre fornecedores de equipamentos para o Departamento de Defesa dos Estados Unidos (Fernandes, 1. Conceitos básicos sobre VHDL, 2014). Ela foi padronizada pelo IEEE em 1987 (*Institute of Electrical and Electronics Engineers*), e revisada nos anos de 1993, 2000, 2002 e 2004 (MORAES & Calazans, Parte 1 – Introdução à Simulação em VHDL, 2013). Uma das vantagens dessa linguagem, é que ela permite a criação de estruturas personalizadas e suporta tipos de dados mais abstratos, criados pelo próprio desenvolvedor. Porém, uma desvantagem é que o *hardware* gerado pode ser menos otimizado.

A linguagem Verilog, teve origem nos anos de 1983 e 1984. Ela foi criada por Prabhu Goel e Phil Moorby, quando trabalhavam para *Automated Integrated Design Systems* (renomeada para *Gateway Design Automation* em 1985), a qual foi, posteriormente, adquirida pela *Cadence Design Systems*. A linguagem foi inventada, inicialmente, com propósitos de simulação e se tornou uma linguagem livre a partir de 1990 (OVI – *Open Verilog International*). Foi padronizada pelo IEEE em 1995 (McNamara, 2012) e revisada nos anos de 2002, 2003, 2005,

2009 e 2013. Ela é mais simples e intuitiva para síntese de circuitos digitais e por se assemelhar mais com a linguagem C. Esses foram os motivos pelos quais ela foi adotada neste trabalho.

Uma abordagem da sintaxe de Verilog, com suas estruturas principais estruturas de linguagem, pode ser vista no Apêndice A.

3.5. Implementações em FPGA

Devido à flexibilidade com a qual a FPGA permite na síntese de circuitos, uma determinada tarefa pode ser executada de diversas maneiras, umas mais eficientes que outras, dependendo dos fatores que são levados em consideração. Algumas abordagens diferentes foram aplicadas durante o desenvolvimento deste trabalho.

Para demonstrar algumas das maneiras de implementação que foram utilizadas e comparadas nesse período, juntamente com suas vantagens e desvantagens, as Subseções 3.5.1 a 3.5.4 visam explorar essas abordagens. Nelas, são implementados diferentes circuitos, mas que executam a mesma tarefa. Assim, a forma com a qual é executado o processo, possui importantes características que serão analisadas.

3.5.1. *Hardware* para comparação de implementações: Filtro passa-altas do bloco de decomposição em *wavelet*

O *hardware* escolhido para ser feita uma comparação em termos de implementação foi o filtro passa-altas do bloco de decomposição em *wavelet* citado na Subseção 2.2.4. Seus coeficientes em ponto flutuante podem ser retirados da Tabela 4 e estão reescritos na Tabela 5. Uma representação do seu diagrama de polos e zeros pode ser vista através da Figura 42, e a resposta em frequência do mesmo, juntamente com a resposta em fase, podem ser vistas pela Figura 43.

Tabela 5: Coeficientes do filtro passa-altas da *wavelet* Daubechies 3.

	Valores em ponto flutuante			
passa-altas	a_0	–0,3327	a_3	–0,1350
	a_1	0,8069	a_4	0,0857
	a_2	–0,4599	a_5	0,0352

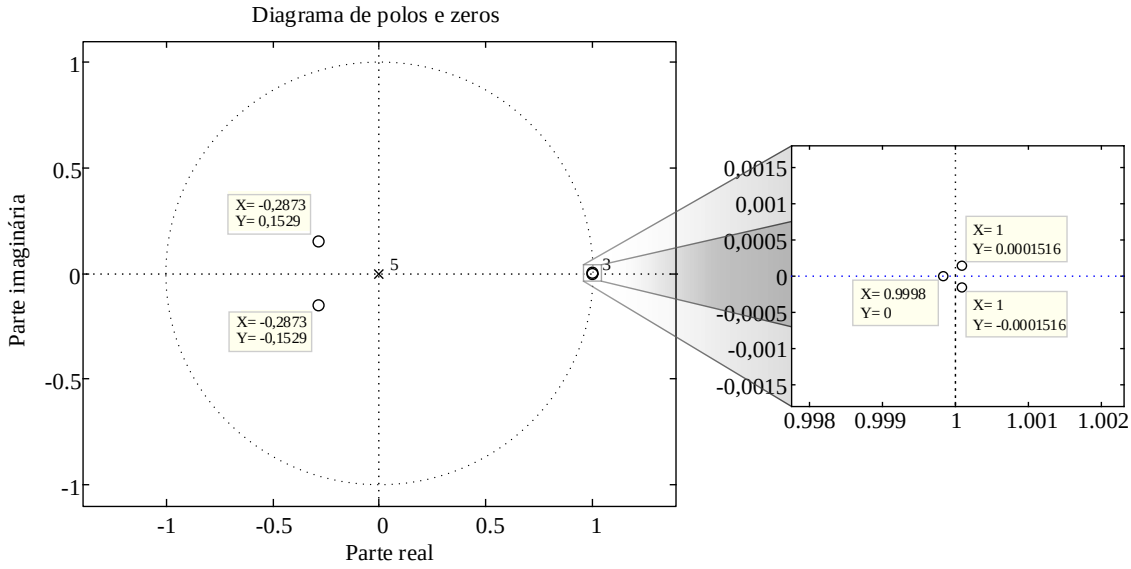


Figura 42: Diagrama de polos e zeros do filtro passa-altas quantizado da *wavelet* Daubechies 3.

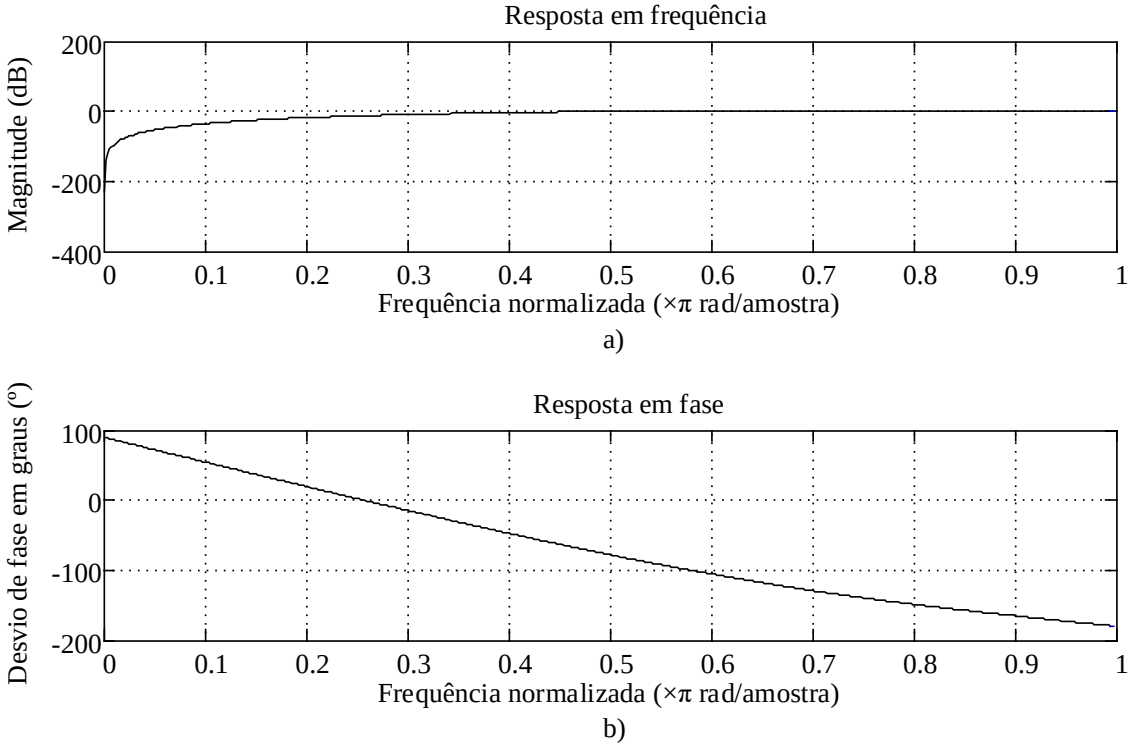


Figura 43: a) Resposta em frequência. b) Resposta em fase.

3.5.2. Implementação direta – Método paralelo

Para uma implementação direta do filtro passa-altas, necessita-se quantizar os coeficientes do filtro. Optou-se por quantizá-los no formato Q_{15}, que utiliza 16 bits, e depois normalizá-los para números inteiros, para que a FPGA fosse capaz de realizar as operações diretamente. Para que

esse processo seja feito, necessita-se primeiramente realizar a verificação contida na Equação (3.1), na qual diz que os módulos de todos os coeficientes a_i devem ser menores do que 1 (normalizados). Caso algum coeficiente se encontre fora dessa verificação, a primeira tarefa consiste em dividi-los por um fator 2^{n^*}, obtendo-se os coeficientes normalizados $a_{i,norm}$.

$$|a_i| < 1, \text{ se não} \rightarrow \frac{a_i}{2^{n^*}} = a_{i,norm} \tag{3.1}$$

Esse fator deve ser a menor potência de 2 possível, que seja maior do que qualquer um dos módulos, conforme mostra a Equação (3.2).

$$\begin{aligned} a_{i,norm} &= \frac{a_i}{2^{n^*}}, \text{ onde } n^* \text{ satisfaz a:} \\ n^* = \lceil n \rceil &= \min\{n \in \mathbb{Z} \,|\, \max(|\, a_i \,|) < 2^n\} \end{aligned} \tag{3.2}$$

Após esse procedimento, multiplica-se os valores por um fator de escala, que para o formato Q_{15} é 2^{15}. Caso o número ainda continue com casas decimais, arredonda-se (Equação (3.3), onde $\lfloor x \rfloor$ representa a função arredondamento). Tem-se então os coeficientes quantizados $a_{i,q}$.

$$a_{i,q} = \lfloor a_{i,norm} \times 2^{15} \rfloor \tag{3.3}$$

Considere que o filtro mostrado na Figura 44 tenha os seus coeficientes quantizados. É fundamental que a entrada $x[n]$ seja também normalizada e que $|x(n)| \leq 1 \; \forall \; n$. Caso contrário, a entrada deve ser escalada (Duque, 2004).

Como os valores dos coeficientes do filtro passa-altas da *wavelet* Daubechies 3, já se encontram de acordo com (3.1), bastou multiplicá-los por 2^{15} e arredondar o resultado, gerando a segunda coluna da Tabela 4 (reescrita na Tabela 6). As entradas do *hardware* de decomposição em *wavelet* do sistema proposto por este trabalho, são provenientes de canais convertidos por um AD com resolução de 16 bits, e portanto, já estão quantizadas.

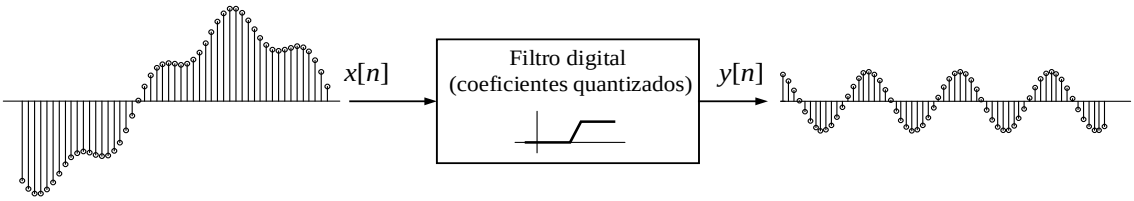


Figura 44: Entrada $x(n)$ num filtro quantizado deve também ser quantizada.

As Figuras 45 e 46 representam o filtro passa-altas da *wavelet* Daubechies 3 com seus coeficientes quantizados no formato Q_{15}.

Tabela 6: Coeficientes quantizados

		Valores em ponto fixo
passa-altas	a_0	–10900
	a_1	26440
	a_2	–15069
	a_3	–4424
	a_4	2799
	a_5	1154

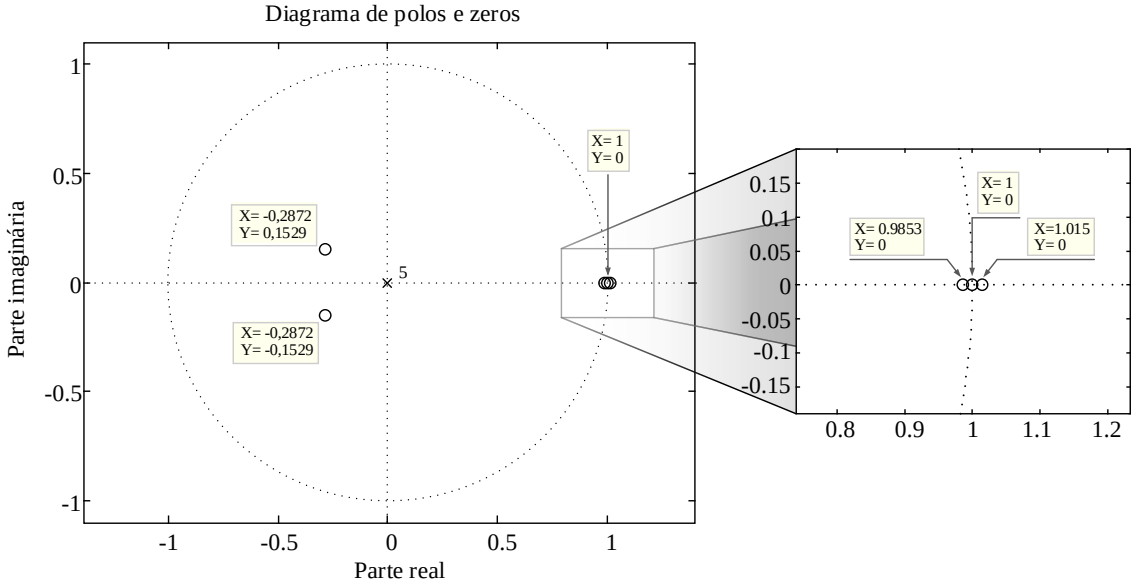


Figura 45: Diagrama de polos e zeros do filtro passa-altas quantizado da *wavelet* Daubechies 3.

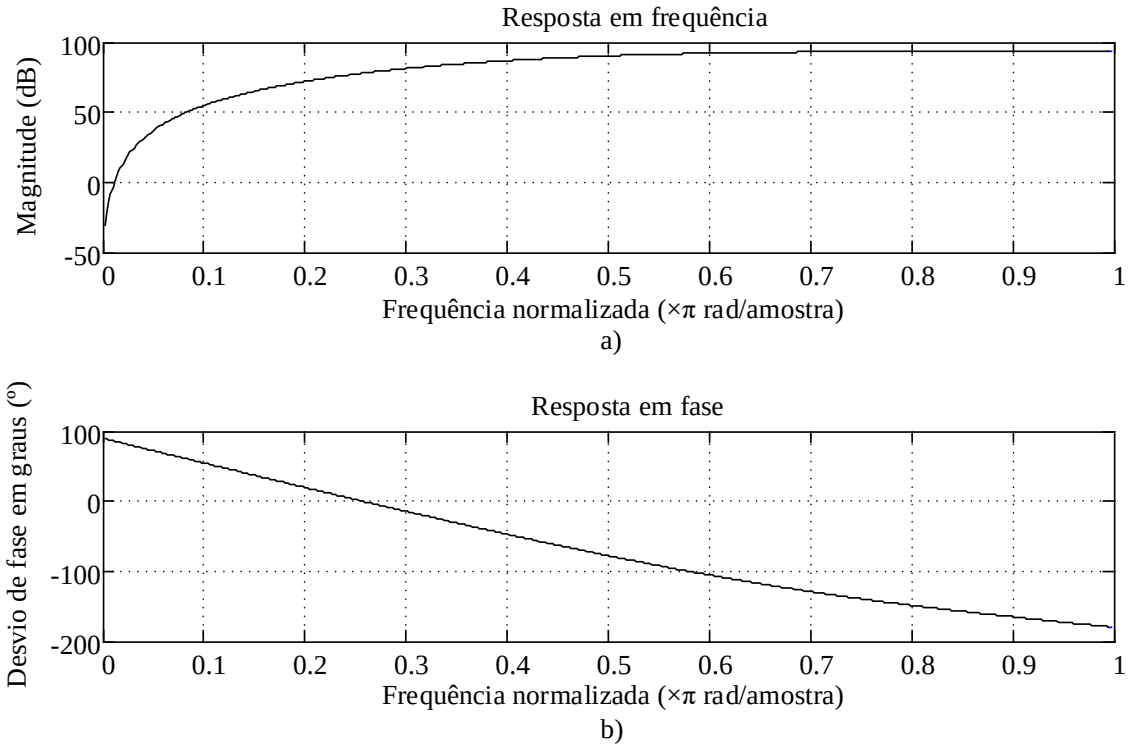


Figura 46: a) Resposta em frequência do filtro quantizado. b) Resposta em fase.

Analisando o filtro antes e depois da quantização, percebe-se que existem algumas dessemelhanças, principalmente no que diz respeito ao diagrama de polos e zeros e à reposta em frequência. Não é apenas um ganho que é inserido ao filtro, mas ele é ligeiramente modificado. Mesmo multiplicando-se resultado da filtragem pelo ganho inverso ao inserido, a resposta conterá um erro, o erro proveniente da quantização. Portanto, já se pode ser identificada uma desvantagem da implementação direta em ponto fixo: perde-se em precisão, uma vez que os arredondamentos trazem consigo, erros numéricos. Porém, quanto maior for o número de bits utilizado, menor será esse erro. Uma implementação direta em ponto flutuante é ainda possível, porém, consumiria muita lógica da FPGA, tornando-se impraticável.

- **Estruturas diretas do filtro FIR**

Um filtro FIR, causal, de ordem N é caracterizado por $N + 1$ coeficientes e, em geral, requer $N + 1$ multiplicadores e N somadores de duas entradas. Estruturas nas quais os ganhos utilizados são os próprios coeficientes da função de transferência, são denominadas de estruturas diretas (Mitra, 2011). A função de transferência no domínio z do filtro FIR passa-altas em questão, está expressa na Equação (3.4), que é um polinômio em z^{-1} de grau $N = 5$.

$$H(z) = \sum_{0}^{5} h[i] \cdot z^{-i} = \sum_{0}^{5} a_i \cdot z^{-i} \quad (3.4)$$

No domínio do tempo, a relação de entrada e saída desse filtro FIR foi apresentada na Equação de diferenças (2.5) e está na sua forma expandida em (3.5). Nela, o número de blocos atrasadores necessários para a implementação é igual a ordem da equação.

$$y[n] = a_0 x[n] + a_1 x[n-1] + a_2 x[n-2] + a_3 x[n-3] + a_4 x[n-4] + a_5 x[n-5] \quad (3.5)$$

Uma estrutura de um filtro digital é dita canônica, se o número de *delays* ou atrasos, na representação em diagrama de blocos é igual a ordem de sua equação de diferenças (Mitra, 2011). A representação direta transversal (*tapped delay line*) da Equação (3.5) pode ser vista na Figura 47. A implementação dessa forma requer cinco atrasadores, portanto, ela é uma estrutura canônica.

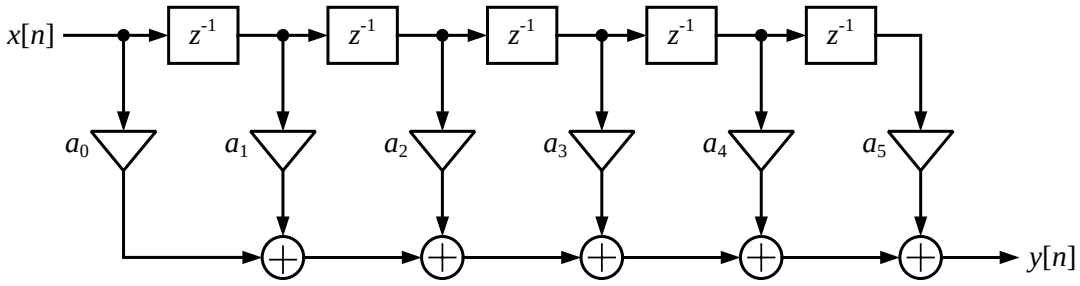


Figura 47: Estrutura direta transversal do filtro passa-altas da *wavelet* Daubechies 3.

Uma forma alternativa de implementação, que também é canônica, é a forma transposta do filtro. Para se chegar a ela basta serem seguidos os seguintes passos (Mitra, 2011):

- Troca-se a direção de todos os caminhos;
- Substitui-se os nós por elementos somadores;
- Substitui-se os elementos somadores por nós;
- Troca-se o lugar da entrada com o da saída.

Dessa forma tem-se a representação mostrada na Figura 48.

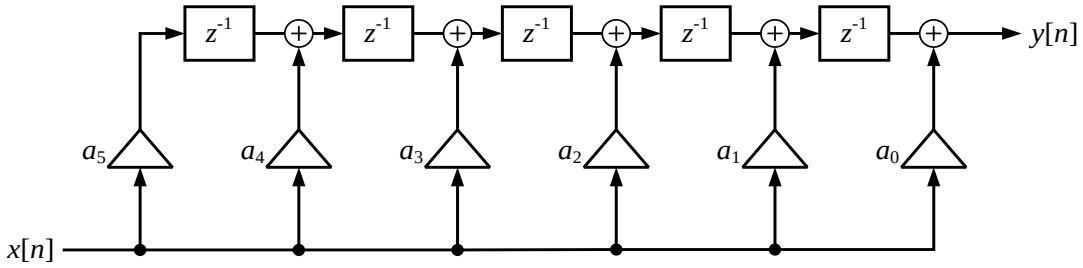


Figura 48: Representação direta transposta do filtro passa-altas da *wavelet* Daubechies 3

Na forma transposta, as somas são quebradas em várias partes entre os registradores, numa estrutura em *pipeline*.

- **Implementação da forma direta transversal (*tapped delay line*).**

A implementação em Verilog para forma direta transversal do filtro passa-altas está descrita na Tabela 38 contida no Apêndice A, no qual podem ser vistos também, os códigos de outros blocos descritos nesta seção. Foram utilizados seis parâmetros para armazenar os valores dos coeficientes (`a0` a `a5`). Os cinco registradores atrasadores foram feitos de forma semelhante ao código mostrado na Tabela 32 (ver Apêndice A). O filtro possui duas entradas de controle para habilitação de escrita e limpeza dos registradores, `en_filt` e `clr_n`, respectivamente e uma entrada de *clock* `clk`.

O fim da estrutura do código revela uma grande atribuição contínua (`assign`) na Linha 36. Porém, a FPGA é otimizada para trabalhar com a arquitetura em *pipeline*, ou seja, com o fluxo de dados seccionado por registradores. No código da Tabela 38, os registradores são utilizados apenas para atrasar as amostras, enquanto as operações são todas realizadas ao mesmo tempo, no momento em que novas entradas chegam. Esse tipo de implementação requer o uso de muita lógica não registrada dentro da FPGA e pode reduzir a frequência máxima com que a FPGA pode funcionar.

O *hardware* gerado pelo código da Tabela 38 pode ser visto na Figura 49.

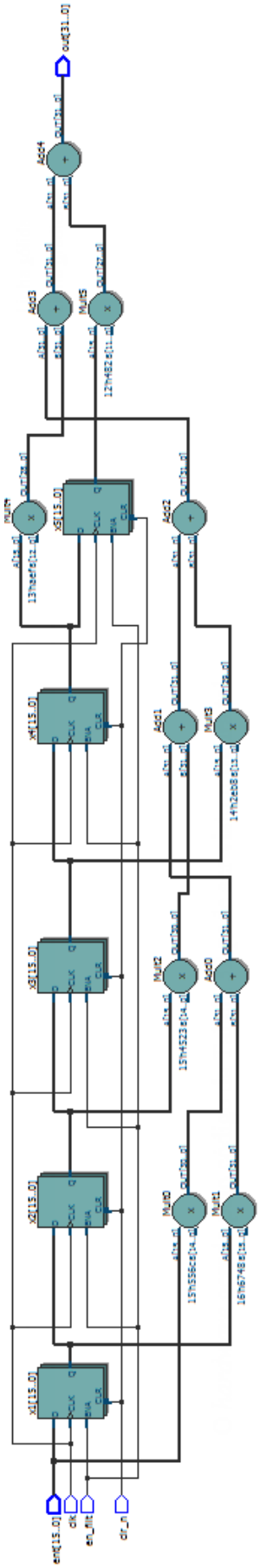


Figura 49: Representação do *hardware* gerado pelo código em Verilog da Tabela 38.

É importante observar que, embora não se pareça muito, por causa da disposição dos elementos, esse *hardware* é exatamente o mesmo representado pelo diagrama mostrado na Figura 47.

O resultado de uma simulação funcional, contendo um sinal com um transitório na entrada do filtro, pode ser visto na Figura 50.

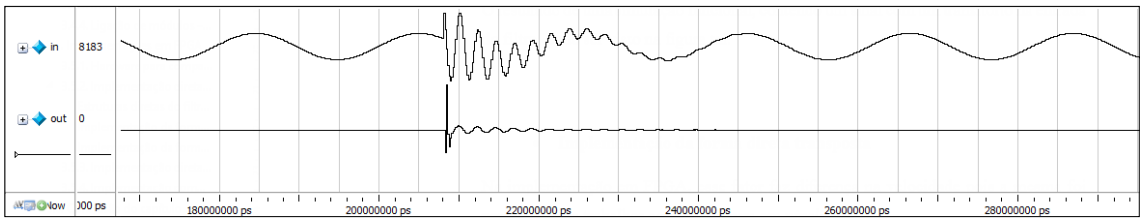


Figura 50: Simulação funcional do filtro passa-altas da *wavelet* Daubechies 3.

Após a compilação ser completada, o *software* sintetizador (Quartus II®) fornece um relatório contendo as características de consumo de recursos do *chip* pelo *hardware* compilado. A Tabela 7 mostra essas características para a implementação na forma direta transversal do filtro passa altas. Pode ser visto que 96 elementos lógicos tiveram apenas a parte combinacional utilizada, ou seja a LUT, e 65 tiveram apenas a parte registrada utilizada. Além desses, outros 53 elementos lógicos tiveram ambas as partes utilizadas, formando um total de 214 elementos lógicos utilizados. Para as operações de multiplicação foram gastos 12 elementos DSP multiplicadores de 9 bits. Enfim, a frequência máxima suportada pelo dispositivo foi de 111,23 MHz.

Tabela 7: Características de consumo de recursos do *hardware* para a implementação direta transversal.

Uso dos LEs		LEs com ambas as partes utilizadas	Totais parciais	
Parte combinacional (LUT)	96	53	149	LUTs
Apenas parte registrada	65		118	Registradores
Total de LEs	214		12	DSP (9 bits)
Frequência máxima suportada ($F_{máx}$)				111,23 MHz

- **Implementação da forma direta transposta**

O código de descrição de *hardware* em Verilog, para a implementação do filtro passa-altas, na sua forma direta transposta, pode ser vista através da Tabela 39 (ver Apêndice A). É importante notar que, agora, a atribuição da saída na Linha 36 possui apenas uma operação de soma e uma

de multiplicação, as outras operações estão divididas em operações menores. Isso favorece a quebra do fluxo, formando uma arquitetura em *pipeline*. A Figura 51 mostra o *hardware* gerado pelo código descrito na Tabela 39. Apesar de algumas diferenças visuais, ele é o mesmo que o representado pela Figura 48.

A Tabela 8 traz as características de consumo de recursos do *chip*, fornecidas pelo relatório do Quartus II®. Nela, pode-se ver que nenhum LE teve apenas sua LUT utilizada, mas 34 tiveram apenas a parte registrada. Os LE com ambas as partes utilizadas, somaram 87, em maior quantidade do que os parcialmente utilizados. Em outras palavras, das 87 LUTs usadas, todas foram provenientes de LEs que tiveram ambas as partes utilizadas. Os mesmos 12 elementos DSP estão presentes. Entretanto, um dos fatores que mais se destacam é o aumento da $F_{máx}$, que passou a valer 162,52 MHz, com mais de 50 MHz de ganho. Ao analisar a Tabela 8, pode-se perceber que a implementação da forma direta transposta possui vantagens sobre forma direta transversal, fornecendo um *hardware* mais econômico, mais rápido e mais equilibrado.

Um ponto que vale a pena ressaltar é que a implementação de um filtro apenas, consome muito pouco da quantidade de lógica disponível na FPGA. Isso é mostrado pelo diagrama de *fitting* na Figura 52, juntamente com a Tabela 9, ambos referentes à implementação do filtro em sua forma transposta, fornecidas pelo Quartus II®. Na figura, cada bloco em azul, na região central do *chip*, é um LAB. Os blocos situados na periferia são os IOEs. As duas colunas brancas, mais afastadas, representam os blocos DSP e as duas colunas centrais, são os blocos de memória. Os LABs destacados, próximos à base das colunas de DSPs são os que foram utilizados para a implementação do filtro. Dentro desse grupo, existem uns mais escuros que outros. Quanto mais escuro, significa que mais recursos do LAB foram utilizados. Como o filtro não exigiu o uso de memória, nenhum bloco das duas colunas centrais foi destacado.

O que se tem a dizer é que, mesmo sendo tão pouco o consumo da lógica utilizada, como no caso deste filtro FIR de quinta ordem, a transposição da estrutura do *hardware* proporcionou um ganho de mais de 50 MHz na frequência máxima permitida. Esse ganho na taxa de *clock* é ainda pequeno, pois o filtro é de baixa ordem. Entretanto, o ganho pode ser muito mais expressivo, considerando-se aplicações nas quais a ordem dos filtros

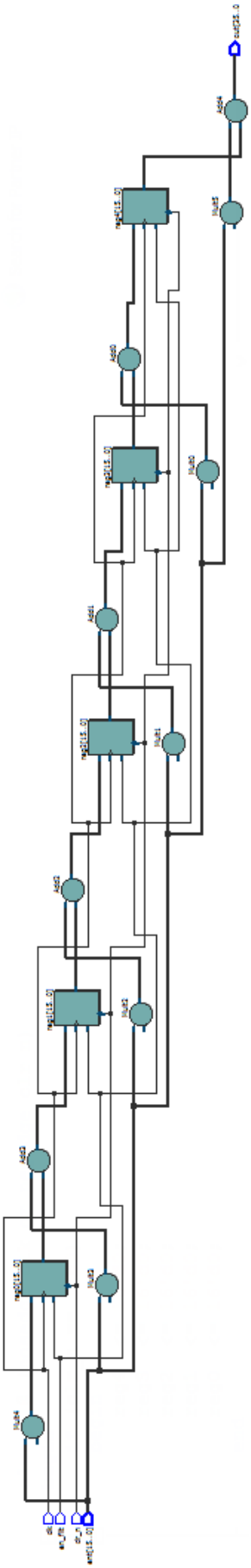


Figura 51: Representação do *hardware* gerado pelo código em Verilog da Tabela 39.

Tabela 8: Características de consumo de recursos do *hardware* para a implementação direta transposta.

Uso dos LEs		LEs com ambas as partes utilizadas	Totais parciais	
Parte combinacional (LUT)	0	87	87	LUTs
Apenas parte registrada	34		121	Registradores
Total de LEs	121		12	DSP
Frequência máxima suportada ($F_{máx}$)				162,52 MHz

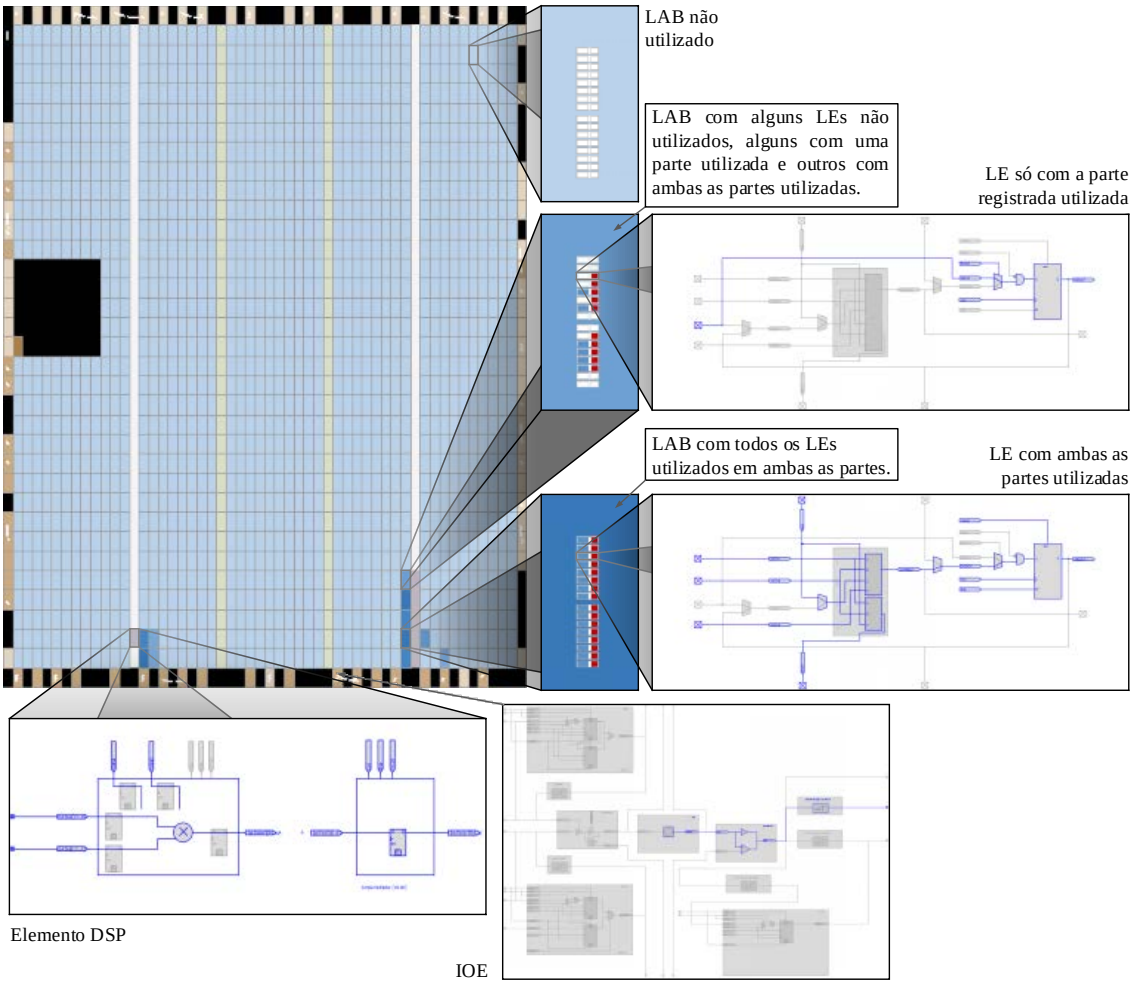


Figura 52: Representação do uso de recursos do *chip*, fornecida pelo Quartus II®.

Tabela 9: Proporção dos recursos do *chip*.

Dispositivo - EP4CE22F17C6	Utilizados	Total	Proporção
Total de elementos lógicos	121	22320	<1%
Total de funções combinacionais	87	22320	<1%
Total de registradores lógicos	121	22320	<1%
Total de memória (bits)	0	608256 bits	0%
Multiplicadores embarcados	12	132	9%
PLLs	0	4	0%

implementados pode passar de 100, como em comunicação digital de banda larga, processamento de imagens e outras áreas (Minaei & Yuce, 2007), (XU & SHUANG, 2011).

A Figura 53 traz uma comparação entre o resultado da filtragem pela FPGA (ponto fixo) e uma filtragem pelo *software* Matlab® (ponto flutuante) para o mesmo sinal de entrada com transitório, visto na Figura 50. Existe um pequeno erro de quantização entre os sinais filtrados. Como foram utilizados 16 bits de quantização, esse erro se torna imperceptível, caso não seja dada uma grande ampliação nos sinais.

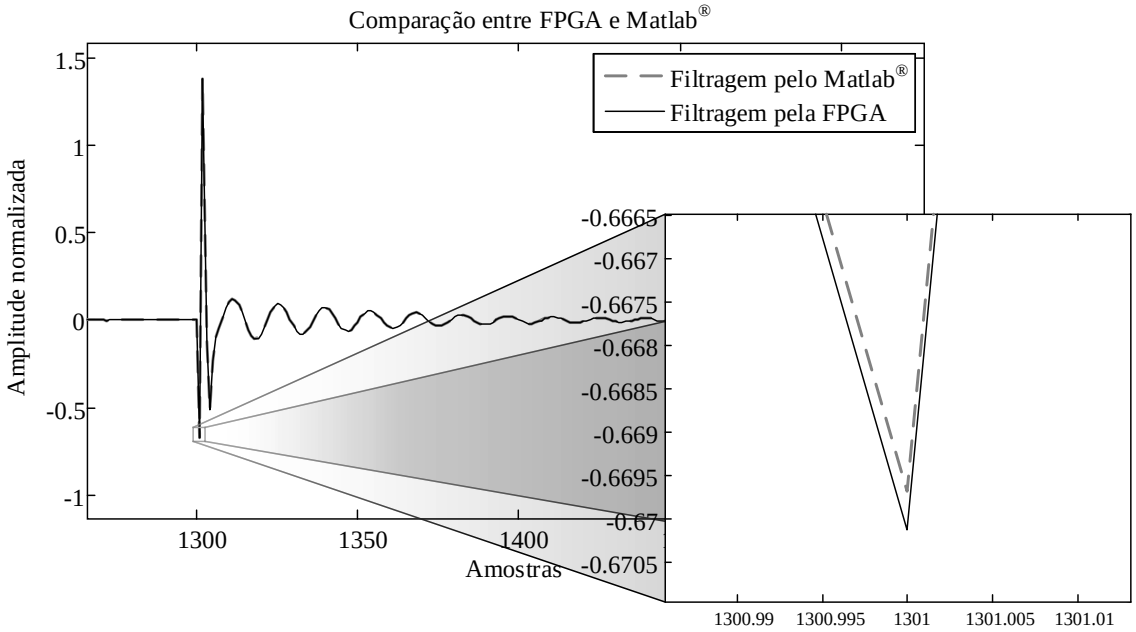


Figura 53: Comparação entre a filtragem pelo Matlab® e pela FPGA.

3.5.3. Implementação direta – Método sequencial

Percebe-se, pela Tabela 9, que o maior uso dos recursos do *chip*, proporcionalmente, foi o de multiplicação com os blocos DSP, que atingiu 9% em ambas as estruturas de filtragem. Aparentemente isso não é um problema, visto que ainda restam 91%. Porém, o sistema completo, para detecção e compressão de distúrbios proposto por este trabalho, visa comportar vários canais de medição, onde em cada canal existem várias estruturas de filtragem. Fatalmente, a quantidade máxima de blocos DSP seria atingida com facilidade, caso não fossa adotada outra estratégia.

O sistema proposto possui uma frequência de amostragem de 7680 Hz (128 pontos por ciclo de 60 Hz), enquanto que a frequência interna do *clock* da FPGA para os circuitos de processamento, foi configurada para 32 MHz. Isso permite que a FPGA tenha 4166 ciclos de *clock* para realizar todos os processos entre cada amostra.

Analisando os fatos dos dois parágrafos anteriores, decidiu-se desenvolver um circuito que fosse capaz de utilizar menos multiplicadores, mesmo que despendesse mais tempo de processamento. Para isso, o circuito sequencial, deveria ser capaz de utilizar o mesmo bloco de multiplicação várias vezes. Isso implica em menor consumo de recursos de multiplicação, mas insere maior atraso na resposta do filtro, o que não é um problema para uma frequência de amostragem de 7680 Hz, requerida pelo projeto.

O *hardware* para a filtragem pelo método sequencial é composto por vários blocos que serão descritos a seguir.

- **Atrasador de amostras**

Para implementar um filtro de quinta ordem, em uma de suas formas canônicas, são necessários 5 atrasadores (*delays*) em sequência. Cada atrasador possui 16 bits de largura, para comportar as amostras quantizadas, vindas do conversor AD. A Figura 54 mostra a representação do bloco atrasador.

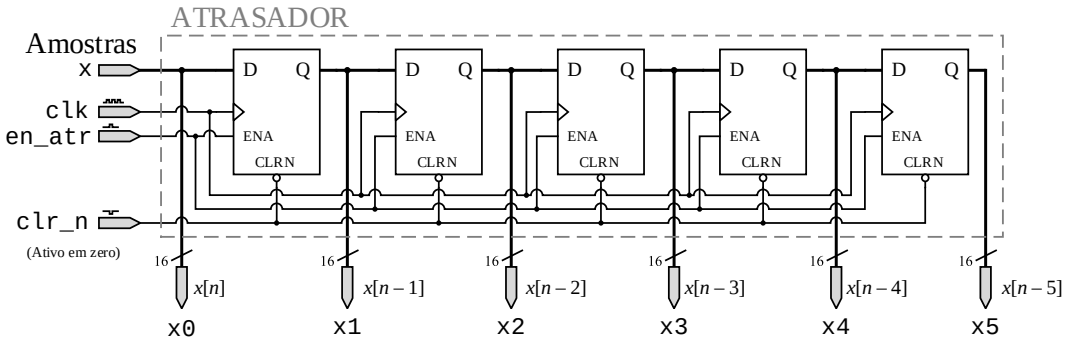


Figura 54: Atrasador de amostras para o filtro passa-altas.

A cada borda positiva de `clk`, a informação passa de um registrador para o outro se `clr_n` estiver em nível lógico alto (inativo) e `en_atr` também estiver em alto (ativo). Caso `clr_n` esteja em nível lógico baixo (ativo), a saída é zero independentemente do valor de `en_atr`. Se `en_atr` está em nível lógico baixo e `clr_n` está em alto, mesmo com bordas positivas de `clk`, a informação fica presa nos registradores e, consequentemente, disponível nas saídas.

O código em Verilog para o *hardware* do atrasador pode ser visto na Tabela 40, contida no Apêndice A.

- **Memória de coeficientes**

A fim de armazenar todos os coeficientes do filtro, uma memória do tipo somente leitura com seis posições de 16 bits cada uma, pode ser criada com o código mostrado na Tabela 41. Sua representação pode ser vista na Figura 55.

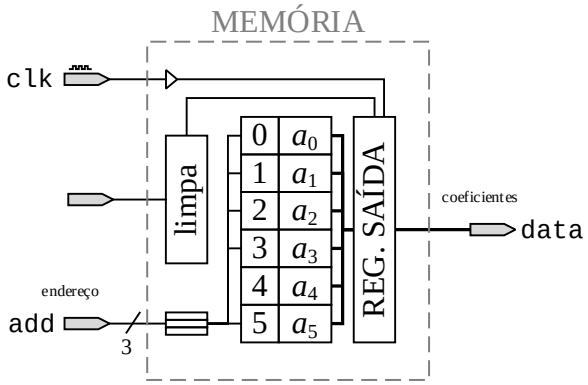


Figura 55: Representação da memória de coeficientes.

A saída das posições de memória é uma atribuição contínua da entrada de endereços. A cada borda positiva de `clk`, o valor no barramento de saída dessas posições é coletado pelo registrador de saída e é disponibilizado na saída do bloco (`data`). Essa memória tem a função de disponibilizar cada coeficiente na ordem correta, um após o outro, para então serem posteriormente multiplicados pelas amostras, com seus devidos atrasos. Sua operação é como a de um multiplexador, mas com valores das entradas sempre constantes, carregados no início do funcionamento.

- **Contador de coeficientes e amostras**

Esse bloco é responsável pelo endereçamento da memória de coeficientes e das amostras que saem do atrasador. Ele é projetado para contar de 0 a 5 e zerar automaticamente. A cada amostra que entra, ele parte do zero e é incrementado a cada ciclo de operações de multiplicação e soma até se completarem as seis multiplicações e as cinco somas. O código de descrição em Verilog pode ser visto através da Tabela 42.

A estrutura condicional da Linha 8 tem a função de limitar o contador até o seu valor máximo, que é 5. Essa estrutura é a mesma apresentada na segunda linha da Tabela 28, de operadores diversos.

A cada borda positiva de `clk`, o contador incrementa seu valor se `clr_n` estiver inativo em nível lógico alto e sua habilitação `en_cnt`, estiver ativa, também em nível alto. Um pulso de `clr_n` faz com que sua saída seja limpa. Se ele estiver com o valor 5, que é o seu valor máximo, a próxima contagem resulta em zero. Uma representação desse contador pode ser vista pela Figura 56.

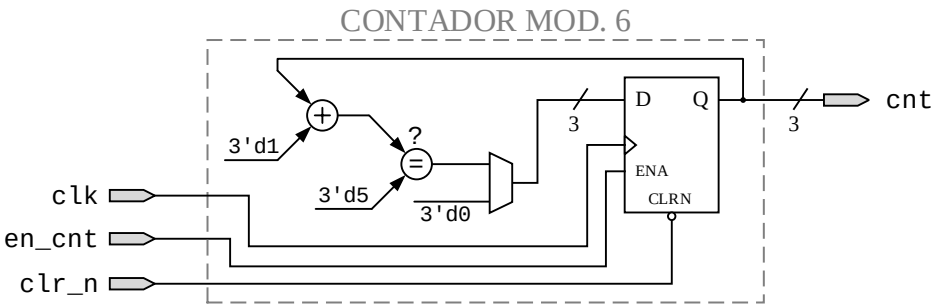


Figura 56: Representação do *hardware* do contador de coeficientes e amostras.

- **Multiplexador de amostras**

As amostras atrasadas necessitam ser disponibilizadas sequencialmente, para então serem multiplicadas pelos devidos coeficientes. O bloco que faz esse papel é o multiplexador. A Tabela 43 mostra o código em Verilog para este bloco de *hardware*.

O multiplexador não é descrito como uma atribuição contínua. Ao invés disso, um registrador foi declarado na saída `outmux` (Linha 4 da Tabela 43), de forma a sincronizar as amostras que saem do atrasador, com os coeficientes vindos da memória, que também possui um registrador de saída. Além disso, a inserção de registradores proporciona uma arquitetura em *pipeline*, que pode maximizar $F_{máx}$. A cada borda positiva de `clk`, a saída do registrador `out_mux` expõe o valor da entrada selecionada por `chn`.

As seis entradas desse multiplexador (`x0` – `x5`) são ligadas nas seis saídas do atrasador de amostras. O seletor desse bloco é o mesmo contador módulo 6, da Figura 56. Dessa forma, o endereçamento da memória controla também o seletor de canais do multiplexador, e a saída do multiplexador fica sincronizada com os valores dos coeficientes que saem da memória.

A Figura 57 mostra uma representação do bloco multiplexador de amostras.

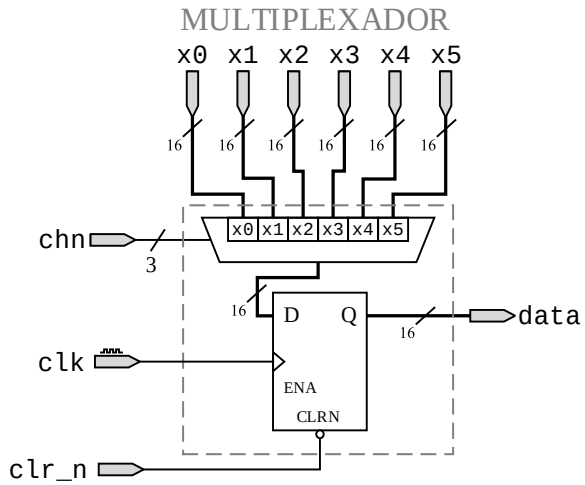


Figura 57: Representação do multiplexador de amostras.

- **Multiplicador**

As amostras vindas da saída do multiplexador precisam ser multiplicadas pelos respectivos coeficientes provenientes da memória, para a realização da filtragem. O bloco responsável por isso é o multiplicador. Seu código, mostrado na Tabela 44, resulta no *hardware* mostrado na Figura 58.

Uma atribuição contínua de multiplicação entre as entradas `ent1` e `ent2` é realizada na Linha 10. O valor dessa multiplicação é colocado no barramento de saída pelo registrador `out_mult` em toda a borda positiva de `clk` em que sinal de `clr_n` esteja inativo e o comando `en_mult` esteja em nível alto. Da mesma forma que nos blocos anteriores, o sinal de habilitação serve para o controle e para manter a informação dentro do bloco.

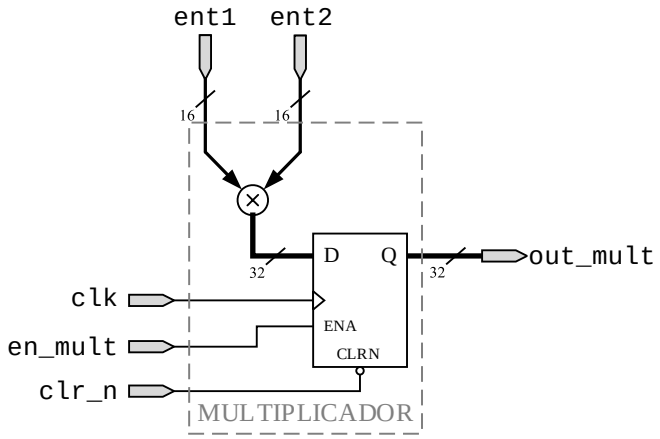


Figura 58: Multiplicador das amostras e coeficientes.

- **Acumulador**

O estágio final da operação de filtragem é a realização das cinco somas de produtos. O bloco responsável por isso é o acumulador. O seu código pode ser visto pela Tabela 45 e sua representação, pela Figura 59.

Do mesmo modo que no multiplicador, a operação de soma é feita com uma atribuição contínua, na Linha 8 da Tabela 45. Essa atribuição é a soma de uma entrada com a saída realimentada de um registrador `out_acc`, que ao final das operações fornece o valor do resultado da filtragem a cada amostra. Um detalhe essencial deste bloco, é que ele possui duas fontes de *reset*, a principal, `clr_n`, que zera todos os blocos assincronamente, e o sinal `reset_acc_n`, específico deste bloco. Este comando é necessário, pois a cada nova amostra, o valor do acumulador deve começar novamente do zero, enquanto todos os outros blocos devem manter os valores referentes às amostras anteriores.

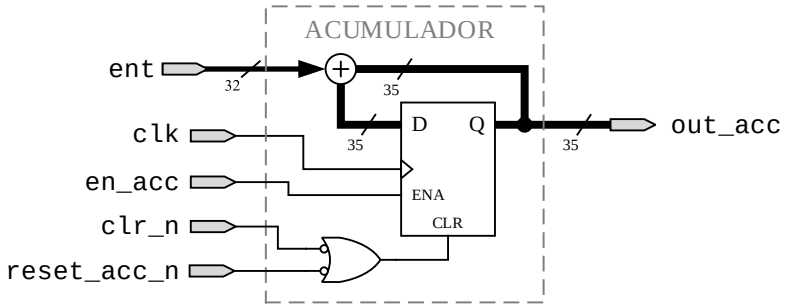


Figura 59: Acumulador.

- **Máquina de estados**

Uma das estruturas mais importantes em implementações em FPGA é a máquina de estados. A maneira de se implementar um algoritmo (circuito sequencial) que seja executado em sua forma mais otimizada, é através de uma máquina de estados (Jappe, Mussa, & Rosendo, 2010).

A máquina de estados, para este filtro, deve ser capaz de gerenciar os seguintes sinais de controle dos blocos:

- `reset_acc`, limpa o acumulador para começar uma nova filtragem.
- `next`, o mesmo que `en_cnt`, controla o contador de coeficientes e amostras.
- `en_mult`, habilita o bloco multiplicador.

- `en_acc`, habilita o bloco acumulador.
- `en_atr`, sinal que controla o atrasador de amostras. Este é o último comando a ser dado.

O diagrama temporal para a máquina de estados é mostrado na Figura 60. Ela possui sete estados diferentes: `S0` a `S6`.

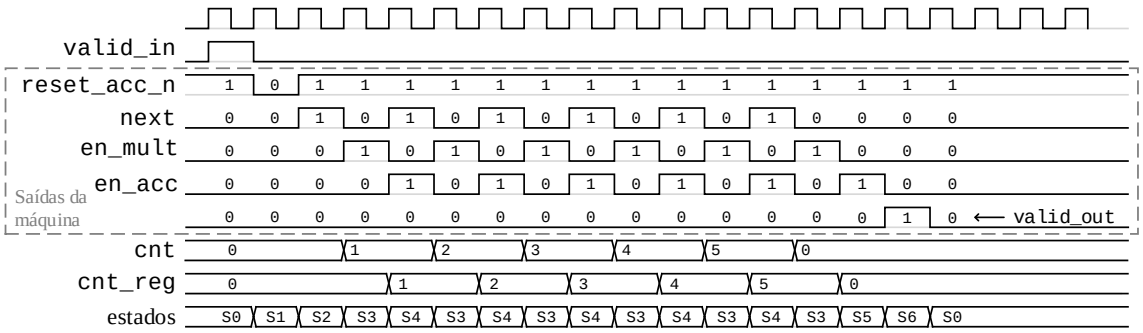


Figura 60: Diagrama temporal da máquina de estados.

O estado inicial da máquina ocorre logo após o término de uma filtragem ou logo após um *clear* geral pelo `clr_n`. Nesse estado, `S0`, nenhum bloco é habilitado. Caso seja dado um pulso de *clear*, a máquina vai para este estado, independente de qual estado esteja.

Um pulso de `valid_in` significa que uma amostra nova está no barramento de entrada dos atrasadores. O que se deve fazer em seguida é limpar o acumulador para começar uma nova filtragem. Enquanto o acumulador está sendo zerado nenhum outro bloco opera, esse é o estado `S1`.

No estado `S2`, habilita-se o contador módulo 6, para seu primeiro incremento. Isso promove a seleção da primeira entrada do multiplexador, juntamente com o primeiro coeficiente da memória.

No estado `S3`, paralisa-se o contador módulo 6, para habilitar o multiplicador, a fim de que ele multiplique as entradas, que nesse momento são, o primeiro canal do multiplexador e o primeiro coeficiente.

Uma vez multiplicados, o próximo passo é a soma, por isso o estado `S4` habilita o acumulador. Entretanto, o contador não precisa ficar parado enquanto se realiza a acumulação. Sendo assim, `S4` também reabilita o contador.

Uma vez habilitado o contador, nova multiplicação pode ser feita, e depois dessa multiplicação, uma nova acumulação, assim por diante. Isso acontece até todos os coeficientes da memória e canais do multiplexador serem utilizados. Por isso os estados `S3` e `S4` se intercalam cinco vezes, terminando de volta ao `S3`. Para controlar essa intercalação, a máquina utiliza como referência o valor do próprio contador que ela habilita, porém, registrado.

Após a última multiplicação, deve ser feita a última acumulação, porém sem habilitar o contador módulo 6. O estado `S5` realiza esta tarefa, habilitando o acumulador (`en_acc = 1`) e não habilitando nenhum outro bloco.

Por fim, o estado S6, habilita o atrasador de amostras e o sistema aguarda nova amostra a ser inserida no barramento de entrada e sua sinalização pelo `valid_in`.

A Figura 61 mostra o diagrama de estados da máquina do filtro, juntamente com sua a tabela de estados.

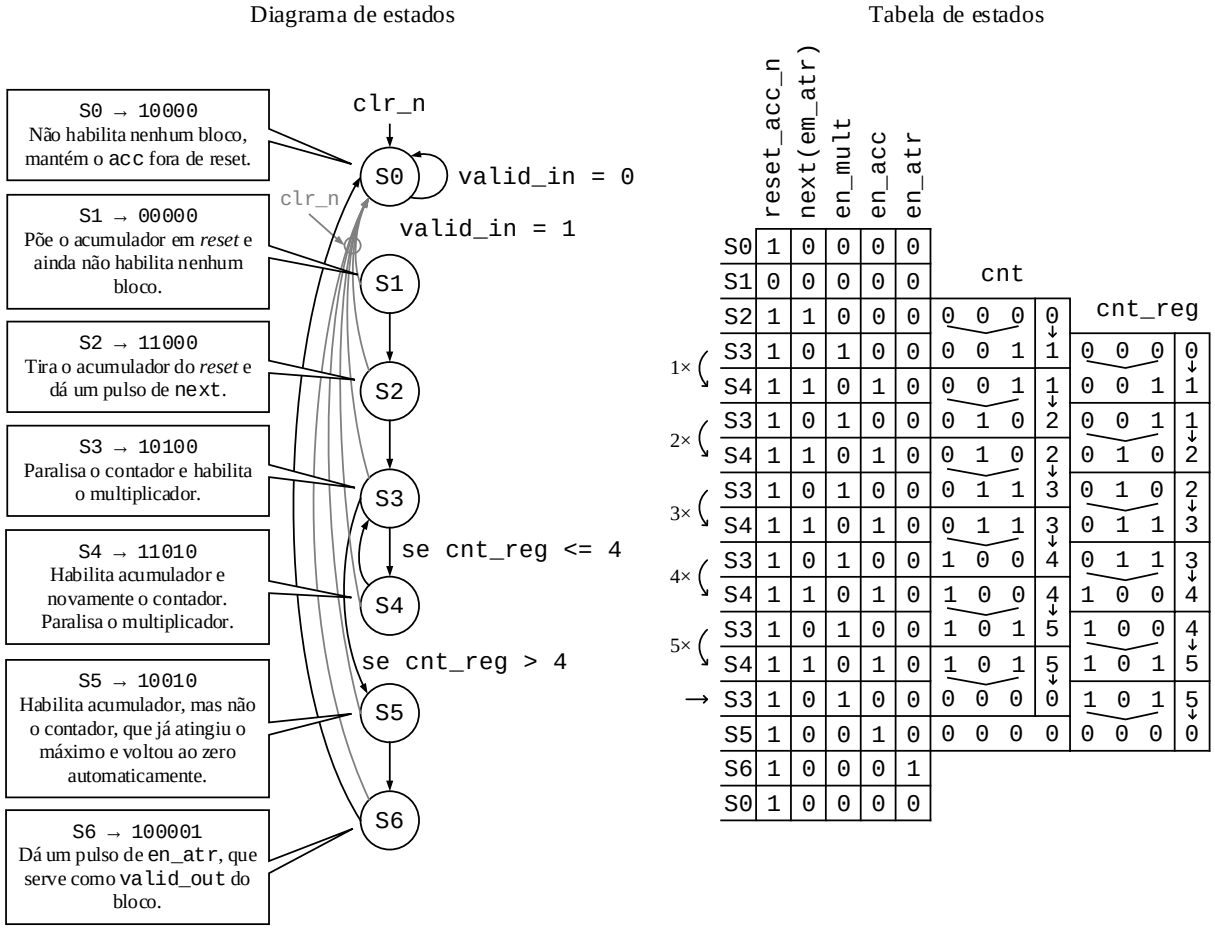


Figura 61: Diagrama de estados e tabela de estados da máquina de estados do filtro passa-altas na forma sequencial.

A Tabela 46 mostra o código de descrição da máquina de estados. O tipo de máquina utilizada fornece as saídas de controle apenas no estado em que está e não na transição.

Na Linha 10 é possível ver uma diretiva antes da declaração do registro de estados. Essa declaração permite que a máquina se recupere de um estado ilegal e retorne ao estado de *reset* (`S0`).

A máquina de estados é composta, basicamente, por dois blocos de procedimento (*procedural blocks*). A primeira serve para definir as saídas da máquina em relação à qual estado se esteja, e a segunda, para definir as transições de estado.

O primeiro *procedural block*, que começa na Linha 16, tem sua lista de sensibilidade do tipo *combinatorial process*. As saídas de cada estado são declaradas em uma estrutura do tipo `case`.

O segundo bloco, começa na Linha 86. Sua lista de sensibilidade é do tipo *clocked process*, pois só é sensibilizada na borda positiva de `clk`.

Uma representação visual da máquina pode ser vista pela Figura 62.

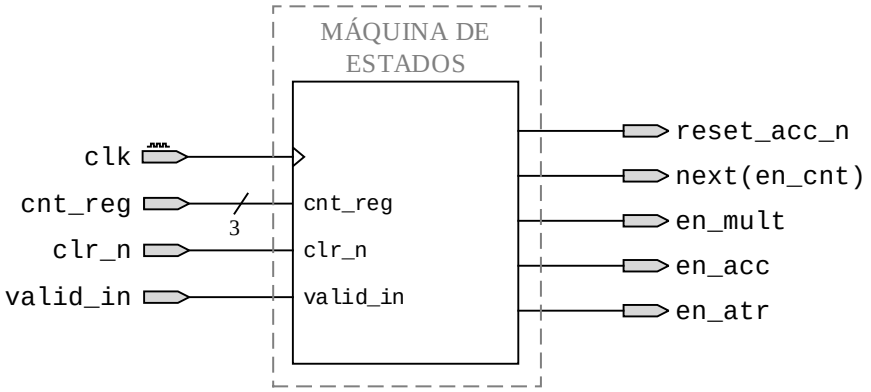


Figura 62: Representação da máquina de estados.

- **Agrupamento de blocos - instâncias**

Para uma melhor organização dos blocos, o contador de coeficientes e amostras foi agrupado com a memória de coeficientes dentro de um mesmo bloco, como é mostrado na Figura 63. O código para esse agrupamento pode ser visto pela Tabela 47.

Outro agrupamento foi realizado pela junção do multiplicador com o acumulador num bloco maior (MAC). O código do MAC pode ser visto na Tabela 48 e sua representação, na Figura 64.

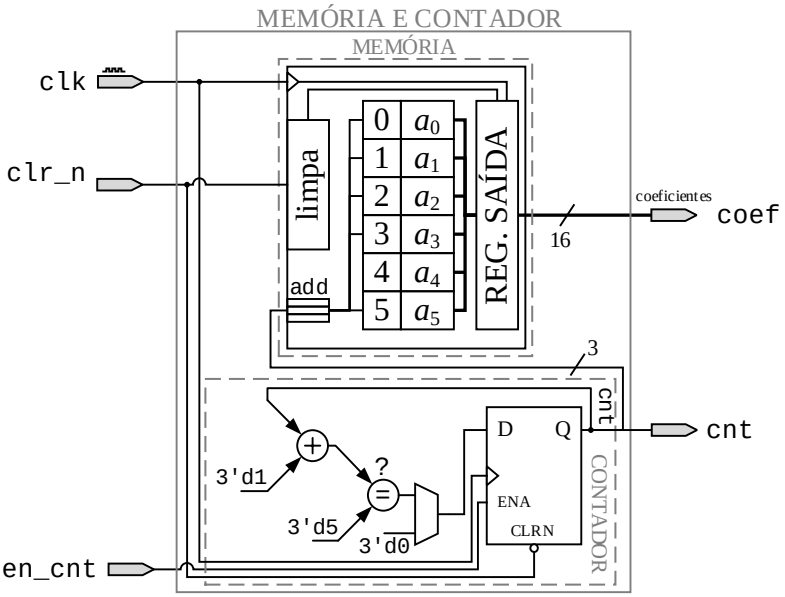


Figura 63: Agrupamento da memória de coeficientes e do contador de coeficientes e amostras.

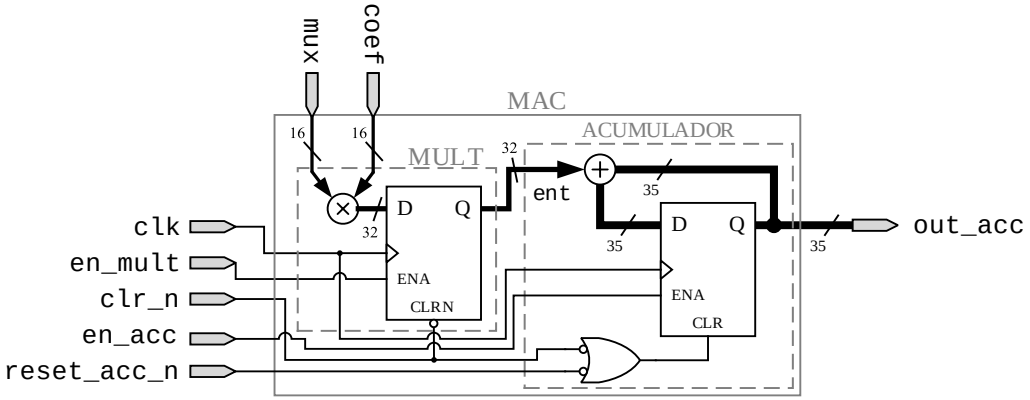


Figura 64: Agrupamento MAC.

- **Estrutura completa – filtro passa-altas *wavelet***

Todos os blocos e agrupamentos descritos anteriormente foram instanciados num mesmo *Top Level*, denominado `filtro_HP`, como mostra a Tabela 49. Uma representação pode ser vista na Figura 65.

Pode ser visto que o filtro possui uma entrada, `valid_in`, e uma de saída, `valid_out`. Esses são sinais de controle. O primeiro serve para indicar que o bloco pode começar a realizar seu processamento pois o estágio anterior a ele já foi concluído. O segundo indica que o bloco concluiu sua tarefa e o próximo pode coletar sua informação.

Esse protocolo de interface é um recurso baseado no padrão *Avalon*, da ALTERA® (ALTERA®, Avalon Interface Specifications, 2015). Ele permite que vários blocos sejam conectados sequencialmente sem necessitar de um bloco maior de controle, como uma grande máquina de estados.

O resultado de uma simulação funcional pelo *software* Model*Sim*®, com um sinal contendo um componente harmônico, pode ser visto na Figura 66. Nela, o sinal de entrada possui um componente de 15º harmônico com 10% da amplitude fundamental, e surge e desaparece de forma intermitente a cada 5 ciclos.

Uma comparação entre o filtro sequencial (Model*Sim*®) e uma filtragem por *software* (Matlab®) pode ser vista na Figura 67. Nela, é mostrado o início da filtragem, quando todos os registradores estão zerados. Isso provoca um transitório inicial, que desaparece após o atrasador de amostras estar com todos os registradores preenchidos com amostras do sinal de entrada. Em todas as entradas e saídas do componente harmônico aparecem, também, pequenos transitórios.

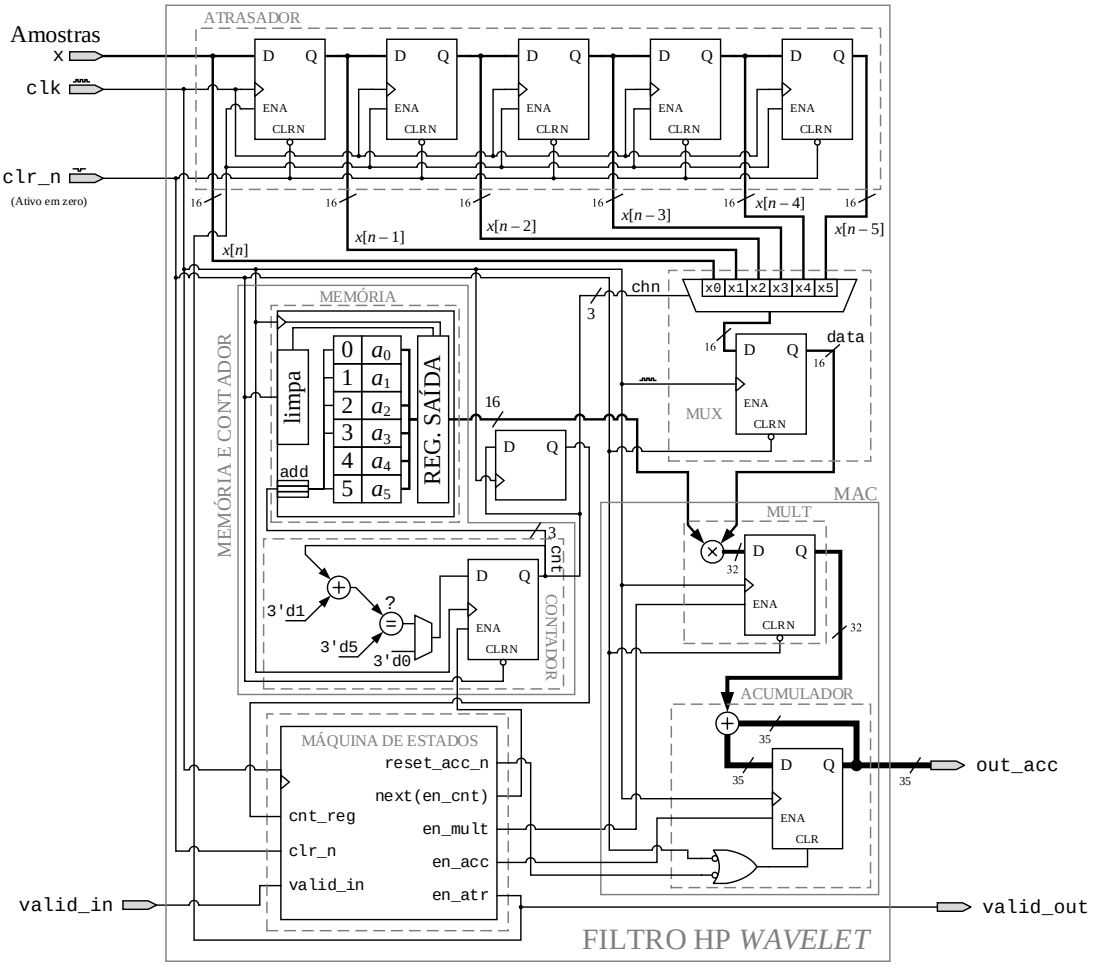


Figura 65: Filtro passa altas da *wavelet*.

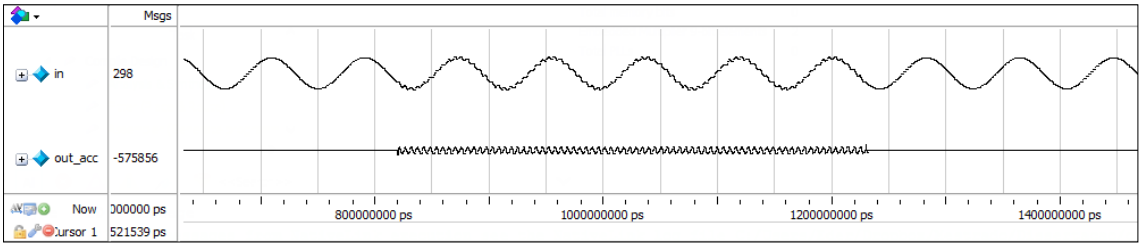


Figura 66: Simulação do funcionamento do filtro passa-altas da *wavelet* com sinal de entrada contendo harmônico intermitente.

Outro resultado, também retirado do Model*Sim*®, que mostra o funcionamento da máquina de estados, pode ser visto na Figura 68. É importante notar a semelhança entre ela e a Figura 60.

A Tabela 10 mostra o consumo de recursos que a implementação do filtro na forma sequencial exigiu, juntamente com a frequência máxima permitida.

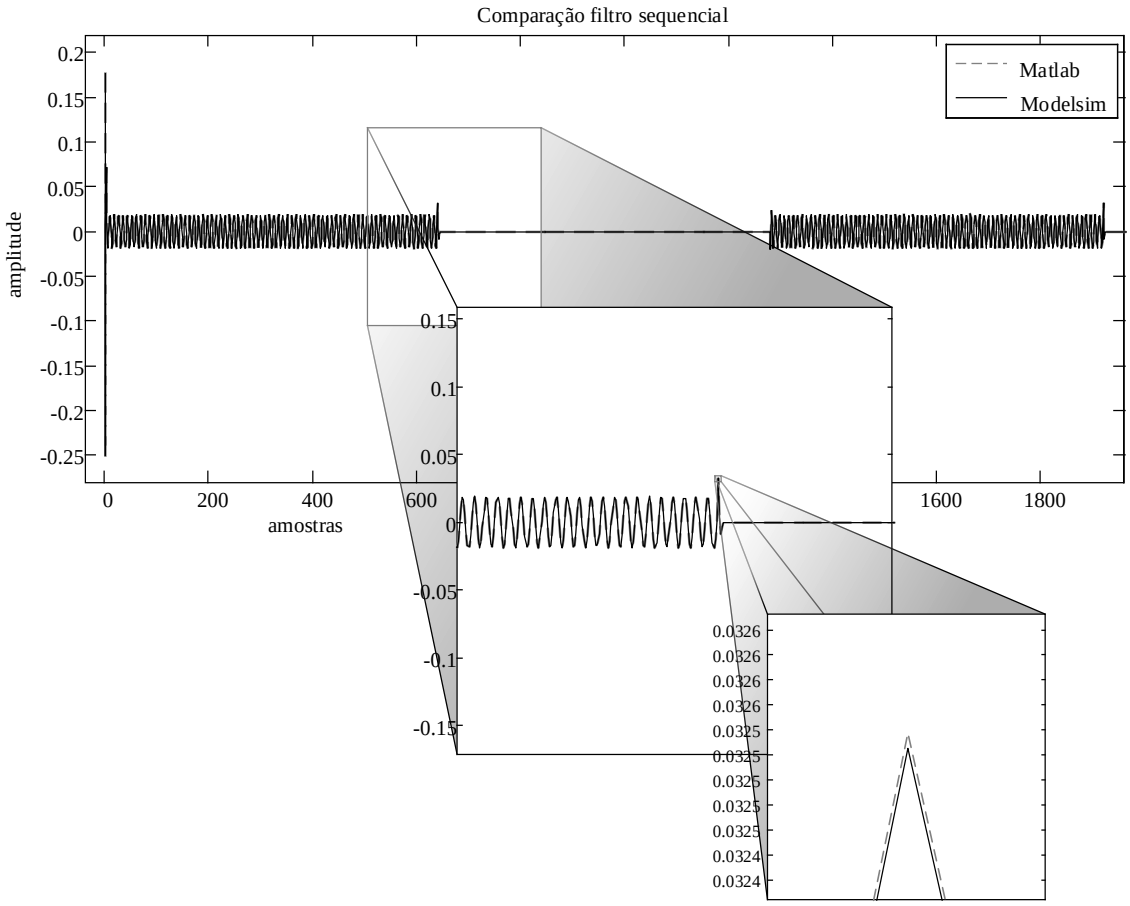


Figura 67: Comparação entre a filtragem pela FPGA e a filtragem pelo Matlab®.

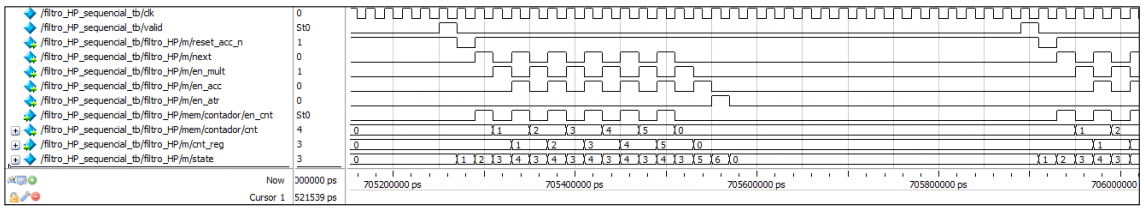


Figura 68: Simulação funcional revela o funcionamento da máquina de estados.

Tabela 10: Características de consumo de recursos do *hardware* para a implementação sequencial.

Uso dos LEs		LEs com ambas as partes utilizadas	Totais parciais	
Parte combinacional (LUT)	7	107	114	LUTs
Apenas parte registrada	66		173	Registradores
Total de LEs	180		2	DSP (9 bits)
Frequência máxima suportada ($F_{máx}$)				206,14 MHz

É importante lembrar que um dos principais motivos para a implementação na forma sequencial é a redução do uso de multiplicadores embarcados. Pode-se ver que ela utilizou apenas dois. O número total de LEs utilizados é maior que a implementação na forma transposta (Tabela 8), e menor do que na forma transversal (Tabela 7). Porém, vale a pena ressaltar que a frequência máxima de operação subiu para 206,14 MHz. Como o *hardware* é sequencial, o resultado da filtragem não é imediato, mas demora alguns ciclos de *clock*. Entretanto, como dito anteriormente, isso não é uma restrição do projeto, visto que tem-se muitos ciclos de *clock* entre uma amostra e a próxima.

3.5.4. Implementação com processador embarcado

Mesmo tendo, a implementação sequencial, economizado recursos da FPGA, como multiplicadores e LEs, ainda assim cada filtro é um *hardware* individual. Isso significa que para utilizar o *hardware* para a implementação da *wavelet*, que demanda seis filtros por canal, deve-se instanciar cada filtro seis vezes, resultando num consumo de *hardware* multiplicado por seis. Considerando-se oito canais, só a decomposição *wavelet* gastaria 48 filtros. Como cada filtro utiliza 2 multiplicadores, chegar-se-ia aos 96 de um total de 132 blocos DSP, utilizados apenas para a decomposição *wavelet*. Mais uma vez recai-se na questão: consumo de recursos de *hardware*.

Devido a essa e outras questões, como flexibilidade de integração e rapidez de implementação de algoritmos, decidiu-se desenvolver um processador embarcado em FPGA. Um *hardware* genérico, que suportasse implementar qualquer algoritmo, mas especificamente para aplicações de DSP.

Na literatura pode-se encontrar alguns exemplos de uso de processadores embarcados em FPGA (Lozano & Ito , 2014), (Schoeberl, 2011) e em outros DLPs. Isso mostra que esta prática não é incomum, mas sim uma estratégia válida para lidar com problemas de otimização de uso de recursos de *hardware*.

Durante o desenvolvimento do processador, buscou-se fazê-lo de tal forma que fosse genérico e ainda assim, mantivesse um uso de recursos aceitavelmente baixo.

O processador desenvolvido, em sua versão final, possui diversas vantagens, dentre as quais pode-se destacar:

- Flexibilidade: ele pode ser usado para implementar qualquer algoritmo de aplicação em DSP. E ainda, vários algoritmos podem ser executados dentro do mesmo programa, utilizando apenas uma unidade do processador dentro da FPGA.

- Integração: pode-se facilmente acoplar quantos processadores forem necessários, limitando-se, obviamente, à quantidade de recursos da FPGA. A inserção do processador em um circuito existente pode ser facilmente realizada por meio de instâncias, proporcionando fácil conexão com outros blocos do sistema.

- Econômico em recurso de *hardware*: utiliza uma quantidade relativamente baixa de LEs, blocos de memória e elementos DSP, quando comparado a outros processadores embarcados já existentes comercialmente (NIOS II - ALTERA®, 2014)

- Atualizável: de acordo com o que a aplicação exigir, é possível incluir operações à ULA do processador ou mesmo retirar as que não estão sendo utilizadas a fim de reduzir o consumo de recursos.

- Programação: O processador admite ser programado em uma linguagem cuja sintaxe é um subconjunto da linguagem C. Entretanto, pode também ser programado em *Assembly*.

- Genericamente reconfigurável por *software*: existe a possibilidade de se escolher entre a utilização do processador em arquitetura de ponto fixo ou ponto flutuante. Isso permite maior abrangência do uso do processador. Além disso, em ponto flutuante, permite-se determinar qual o tamanho da mantissa e qual o tamanho do expoente. Esses e outros parâmetros podem ser totalmente configurados via *software*.

- Interface de programação: facilmente programável, com uma interface de programação específica.

Devido à flexibilidade da FPGA, pode-se inserir novas instruções no decodificador de instruções, para se adaptar ao que é exigido na aplicação. Isso aumenta ainda mais o nível de adaptação e atualização do processador.

Para poder ser programado tanto em C quanto em *Assembly* foram desenvolvidos compiladores para as duas linguagens. Para isso foram utilizadas ferramentas gratuitas da GNU, como *flex* (Paxson, 1995) e *bison* (Donnelly & Stallman, 2015).

As configurações dos parâmetros do *hardware* do processador, via *software*, são realizadas por meio de diretivas de compilação no início do código em C. Isso significa que o próprio compilador é projetado para criar os arquivos em Verilog de acordo com o que foi estabelecido pelo usuário. Além disso, ainda foi desenvolvida uma IDE em C#, dedicada ao processador, por meio da qual se pode escrever e compilar o código de programa para o mesmo.

O processador possui um conjunto reduzido de instruções (RISC) e sua arquitetura é do tipo *Harvard*, na qual tem-se a memória de dados separada da memória de programa. Ele é capaz de executar as operações (como soma, multiplicação, comparação, divisão) em ponto flutuante e em um único ciclo de *clock*.

- **Representação em ponto flutuante**

A representação em ponto flutuante utilizada neste processador é particular. O bit mais significativo (MSB) é o bit de sinal. Em seguida, na ordem decrescente de significância, os bits que se seguem são os do expoente com a representação em complemento de dois. Por fim seguem os bits da mantissa. A Figura 69 ilustra a representação em ponto flutuante utilizada. Nela, a mantissa possui largura de `MAN` bits, o expoente possui `EXP` bits e o sinal, 1 bit. A palavra de dados, portanto, possui `MAN + EXP + 1` bits de largura.

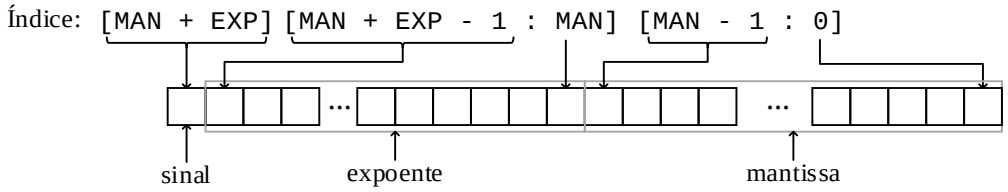


Figura 69: Representação em ponto flutuante adotada.

- **Topologia do processador**

A Figura 70 mostra uma representação do processador em sua configuração de ponto flutuante. A partir dela serão analisados os principais blocos. Por causa da extensão dos códigos em Verilog, ficaria inviável pô-los todos por extenso em tabelas, por isso as interpretações serão feitas por meio de comentários e imagens, quando oportuno.

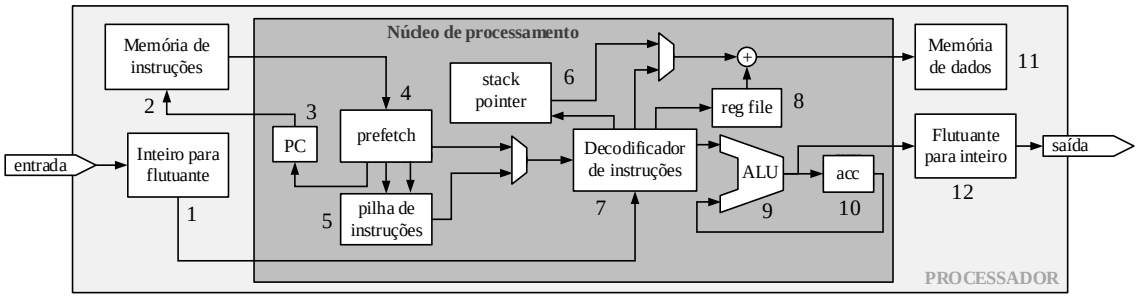


Figura 70: Representação do processador embarcado com seus blocos internos.

1. Bloco de transformação de número inteiro para flutuante: este bloco é responsável pela conversão de representação em número inteiro para ponto flutuante, uma vez que o processador faz os cálculos em ponto flutuante. Caso tenham outros blocos de algoritmos na FPGA que trabalhem com representação inteira, é possível a interconexão com a entrada do processador. A Figura 71 mostra uma representação do funcionamento desse bloco.

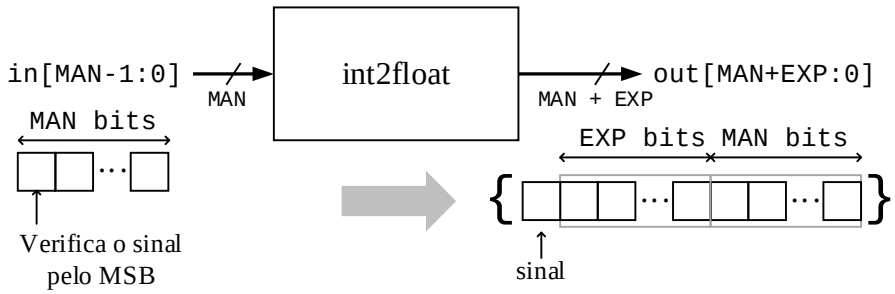


Figura 71: Representação do conversor de inteiro para ponto flutuante.

A verificação do sinal é feita pelo MSB. Para isso o dado de entrada deve estar representado como complemento de dois. A partir dela é concatenado o sinal, um expoente de largura EXP com o valor zero e uma mantissa com o valor de in, respeitando-se o sinal do dado de entrada.

Este bloco não efetua o processo de normalização da mantissa. Isto é feito dentro da Unidade Lógico-Aritmética, que será vista mais à frente.

2. Memória de instrução: é uma memória do tipo ROM (*Read Only Memory*) síncrona. Nesse bloco são armazenadas as instruções de programa. Cada instrução é dividida em duas partes: o código de operação (`opcode`) e o operando (`operand`). O código de operação significa qual operação dever ser realizada com o dado localizado em um determinado endereço da memória de dados. Esse endereço é o `operand`. A Figura 72 mostra uma representação da instrução. A largura, em bits, da instrução é `NBOPCO + NBOPER` bits.

   Todas as instruções são carregadas à memória de instrução por meio de um arquivo de inicialização de memória (mif – *Memory Initialization File*) gerado pelo compilador *Assembly* desenvolvido.

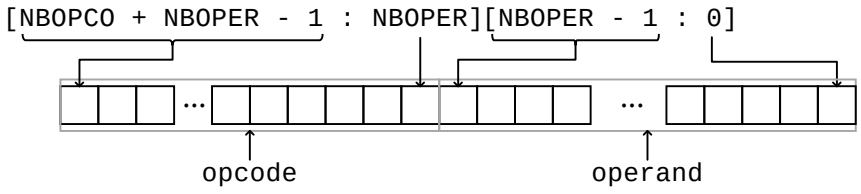


Figura 72: Instrução.

3. PC: contador de programa. Endereça a próxima instrução que vem da memória de instrução. Ele pode ser carregado para indicar uma posição específica da memória de instruções, caso seja identificada uma instrução de salto. Caso contrário, ele é incrementado a cada instrução executada.

4. Prefetch: busca a próxima instrução enquanto a instrução atual está sendo executada.

5. Pilha de instruções: este bloco serve para armazenar o endereço da instrução anterior à chamada de uma sub-rotina a fim de dar continuidade ao programa após a sub-rotina ser concluída;

6. Stack pointer: funciona como um ponteiro para a memória de dados. Começa apontando do fim de memória de dados, onde as posições ainda estão vazias e preenche em direção ao início. Ele é responsável por criar uma pilha de dados, a fim de realizar cálculos com diversas variáveis. As variáveis são alocadas nessa pilha e retiradas na ordem correta para a conclusão do cálculo. A Figura 73 mostra uma representação da operação desse bloco.

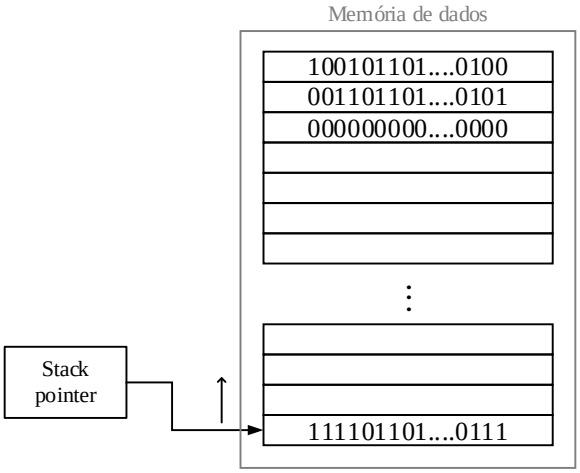


Figura 73: Stack pointer.

7. Decodificador de instruções: este bloco recebe as instruções, as decodifica e envia os comandos para a execução. Ao receber as instruções, uma estrutura do tipo `case` é usada para se decidir qual comando deve ser dado.

8. Reg file: é um ponteiro para a memória de dados. Permite a utilização de vetores no código.

9. ALU: unidade de operações lógicas e aritméticas. Este bloco é o coração do processador, por isso é importante dar-lhe mais atenção. Ele é responsável por todas as operações que o decodificador de instruções ordena. Os dados chegam em sua entrada com representação em ponto flutuante e normalizados. Para a ALU poder realizar as operações deve-se adequar as bases e os expoentes de acordo com a operação. Para isso, a ALU utiliza operações de *shift* mostradas na Tabela 27. Por exemplo, para se realizar a soma de dois números com expoentes diferentes, deve-se desnormalizá-los, de modo que eles possuam o mesmo expoente. Após esse processo, os expoentes estão iguais,

então prossegue-se com a soma das bases e a repetição do expoente (bloco `mysoma`). Ao fim das operações deve-se normalizar novamente, para isso existe o bloco normalizador.

A ALU permite, também, a comparação entre dois números, essa é a função do bloco `mycomp`. Ele permite também, operações lógicas, como “ou” (`|`), “e” (`&`), e inversão (`~`).

Todas as operações da ALU são realizadas por meio de atribuições contínuas (*Continuous Assignment*). Por isso, ela não utiliza nenhum registrador. Antes do dado chegar à saída, ele deve passar por um multiplexador para se determinar a qual das operações, o dado da saída fará parte. Em outras palavras, a ALU faz todas as operações com os dados de entrada, cabe ao multiplexador definir qual deles sairá. Existe um multiplexador para cada parte do dado: sinal (`omux_s`), mantissa (`omux_m`) e expoente (`omux_e`). A Figura 74 mostra uma representação da ALU.

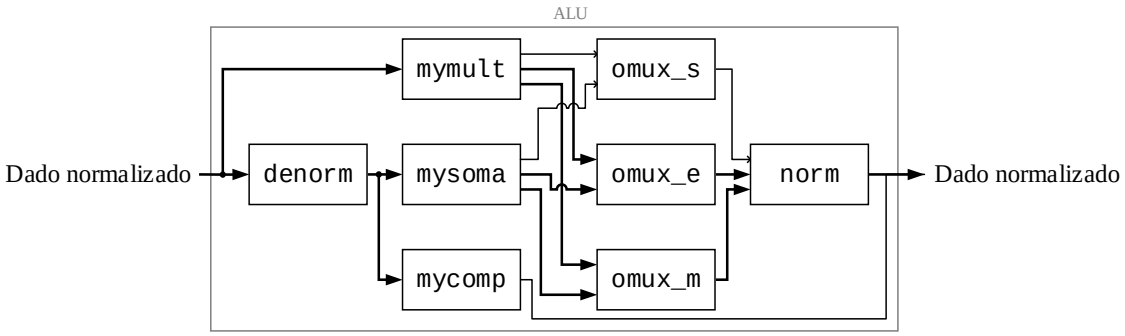


Figura 74: Representação da ALU.

A saída da ALU, é a saída do núcleo de processamento. Caso se queira acoplar um outro processador em sequência pode-se fazê-lo conectando-se esse dado diretamente ao núcleo de processamento do outro processador, ignorando os transformadores de ponto flutuante para fixo e vice-versa.

10. ACC: acumulador da ALU. Ele permite realizar cálculos de acúmulo, como os vistos na operação de filtragem.

11. Memória de dados: é uma memória RAM síncrona. Ela possui barramentos de leitura e escrita separados, tanto o barramento de dados, quanto o de endereços. Os dados iniciais são carregados por meio de um outro arquivo no formato .mif, também gerado pelo compilador *Assembly*.

12. Bloco de conversão de flutuante para inteiro: faz o papel inverso ao bloco de conversão de inteiro para flutuante. O dado de entrada em ponto flutuante possui `MAN+EXP` bits, a saída em número inteiro, possui a mesma largura de bits. Para realizar a conversão, primeiramente o bloco separa as partes do número (mantissa, expoente e sinal). Analisando o sinal, atribui um valor positivo ou negativo à mantissa. O expoente fará com que ele desloque para esquerda ou para a direita, dependendo do sinal do mesmo. Caso seja necessário se deslocar para a esquerda, ele o fará mantendo-se o sinal, através da utilização do operador >>>, mostrado na Tabela 27.

- **Programação e simulação do processador – filtro passa-altas *wavelet***

O processo de programação do processador segue as etapas descritas na Figura 75. Primeiramente, escreve-se um código cuja sintaxe é um subconjunto da linguagem C, através de uma IDE dedicada. Essa IDE passa os códigos do programa para um compilador capaz de criar um código em *Assembly* a partir do código em C (Compilador C → *Assembly* na Figura 75). Esse arquivo *Assembly* é então passado para outro compilador que o transforma em códigos de máquina (arquivos .mif). Durante a síntese da FPGA o Quartus II® carrega as informações contidas nos arquivos .mif às devidas memórias de instruções e de dados.

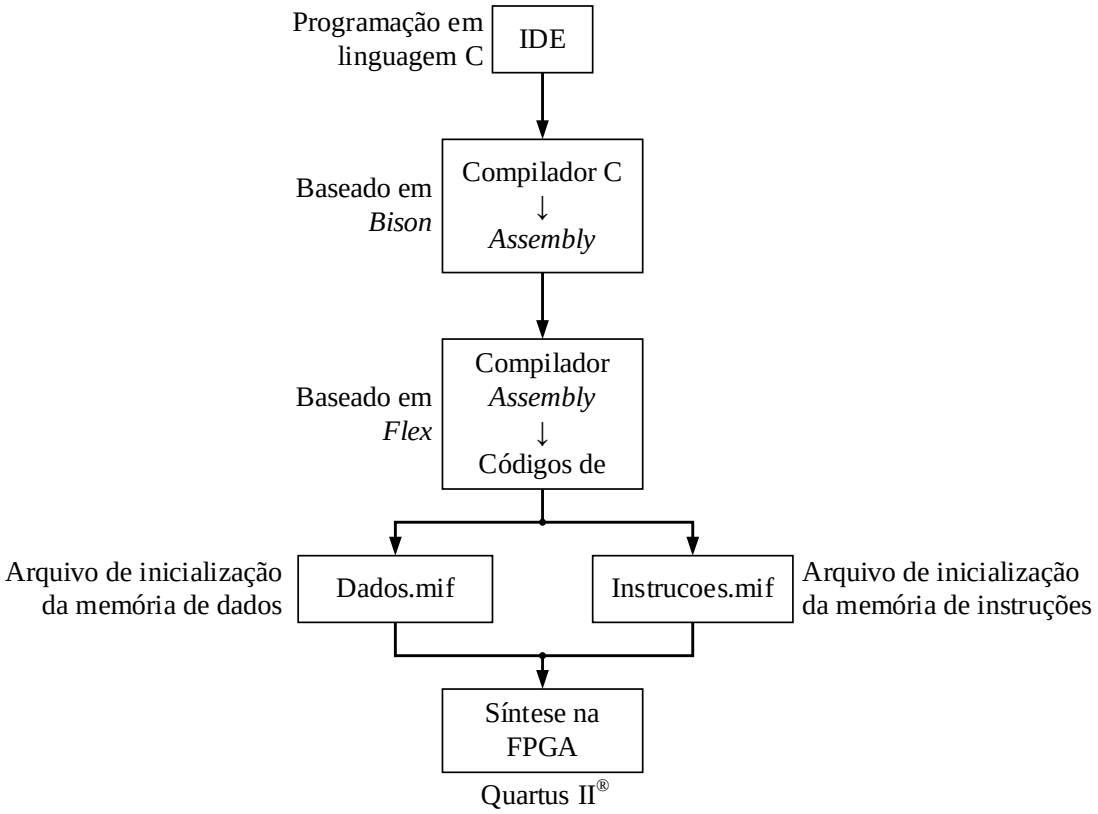


Figura 75: Etapas de programação do processador.

Para programar o processador, foi necessário, primeiramente, escrever o código de programa do filtro passa-altas. A Figura 76 mostra uma imagem da interface de programação da IDE dedicada ao processador. A sintaxe utilizada no código faz parte da linguagem C. O algoritmo descrito é a implementação do filtro passa-altas da *wavelet*.

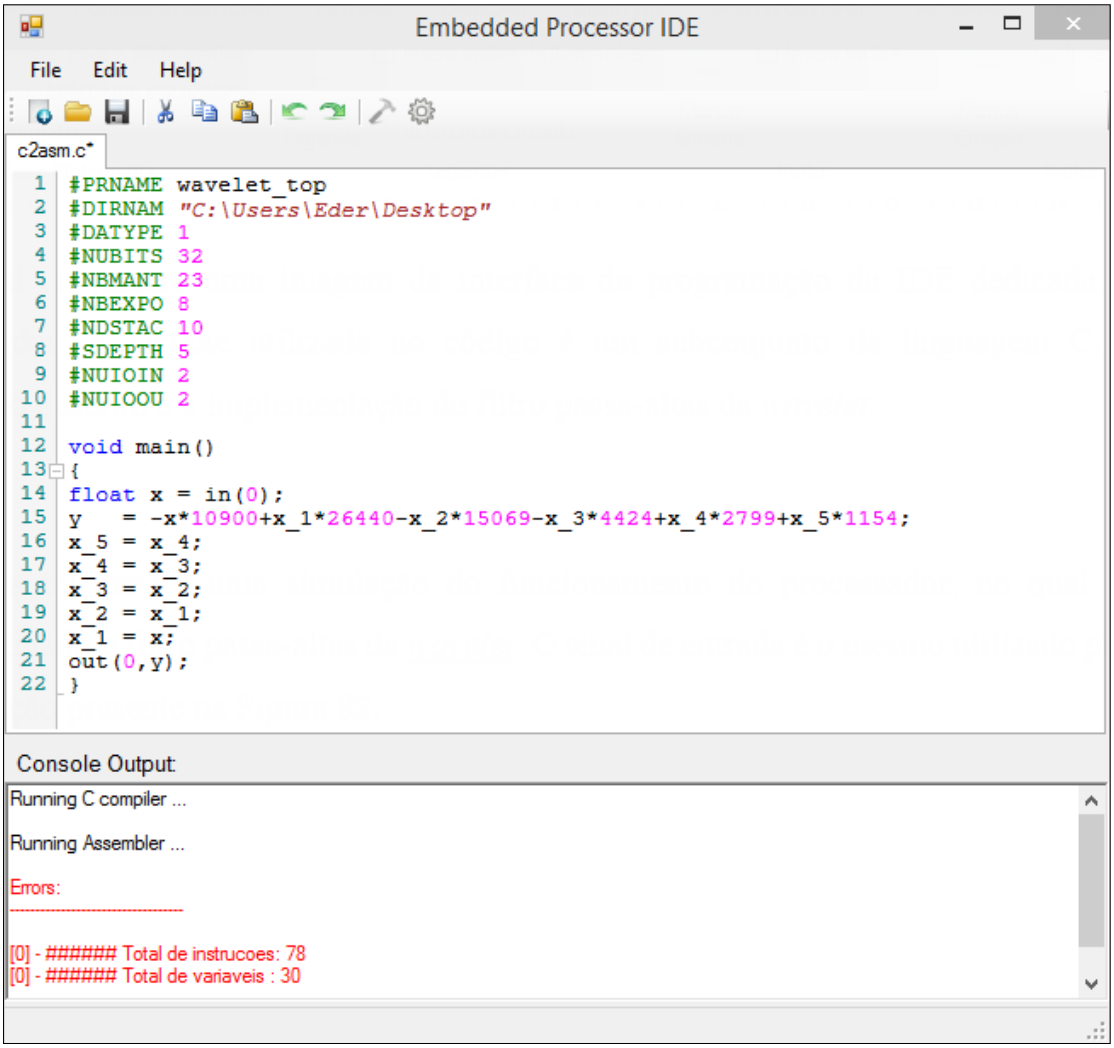


Figura 76: IDE dedicada ao processador embarcado.

A Figura 77 mostra uma simulação do funcionamento do processador, na qual foi implementado o filtro passa-altas da *wavelet*. O sinal de entrada é o mesmo utilizado para a simulação presente na Figura 66.

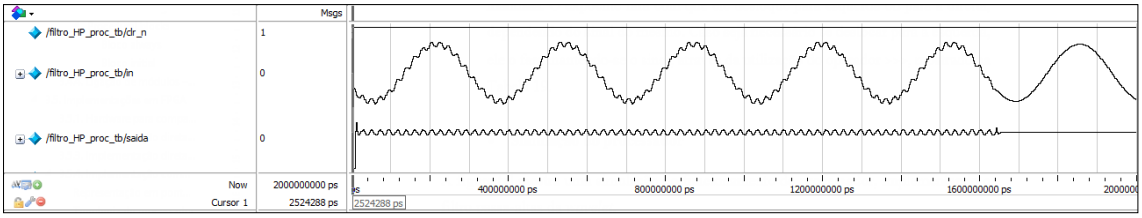


Figura 77: Simulação funcional do processador embarcado.

A Tabela 11 traz informações do uso dos recursos da FPGA com o método utilizando o processador embarcado.

Tabela 11: Uso de recursos da FPGA pelo método com processador embarcado.

Uso dos LEs		LEs com ambas as partes utilizadas	Totais parciais	
Parte combinacional (LUT)	916	44	960	LUTs
Apenas parte registrada	4		48	Registradores
Total de LEs	964		2	DSP (9 bits)
Memória (bits)	5074			
Frequência máxima suportada ($F_{máx}$)				79,66 MHz

É nítido o aumento de espaço lógico ocupado pelo processador embarcado. O número de LEs utilizados é bem maior do que as implementações em formas diretas. Além disso, utilizou-se 5074 bits de memória embarcada, manteve-se em 2 blocos DSPs e a frequência foi reduzida para 79,66 MHz. Fazendo uma análise superficial da Tabela 11, aparentemente o processador embarcado não parece uma alternativa vantajosa. No entanto, é importante lembrar que um só processador pode realizar a implementação de inúmeros filtros e outros algoritmos, instanciando-o apenas uma vez. O número de LEs não muda, nem a quantidade de blocos DSP, a não ser que seja desejado incluir novas funções e operações à ALU. Um fator que notavelmente se altera é a quantidade de bits de memória utilizados, devido ao número de instruções e dados resultantes da compilação do programa.

Para ser feita uma melhor comparação entre os métodos direto sequencial e com processador, foram feitas quatro implementações cujos resultados estão expressos na Tabela 12. A primeira foi realizada com apenas um canal de medição utilizando a implementação direta sequencial, representada pela segunda coluna da tabela (1c D.S.). A segunda implementação foi com um canal utilizando o método com processador embarcado, representada pela terceira coluna (1c P.E.). A quarta e a quinta colunas (8c D.S. e 8c P.E.) representam as duas últimas implementações, nas quais foram utilizados os oito canais de medição com o método direto sequencial e com o processador embarcado, respectivamente. Em todas as quatro implementações, além do conjunto de filtros da decomposição *wavelet*, foram sintetizados circuitos que permitem testar a funcionalidade do projeto em situações reais. Porém os recursos analisados foram somente os correspondentes à *wavelet*. As linhas da Tabela 12 mostram alguns dos recursos utilizados da FPGA: número de blocos lógicos (BL), número de blocos DSP,

número de bits de memória utilizados (BM), *clock* máximo permitido (CLK) e número aproximado de ciclos de *clock* utilizados (NCU).

Tabela 12: Comparação entre o método direto sequencial e o método com processador embarcado para 1 e para 8 canais de medição.

	1c D.S.	1c P.E.	8c D.S.	8c P.E.
BL	1163	1032	13282	1635
DSP	20	2	132	2
BM(bits)	0	5074	0	33914
CLK(MHz)	67,46	79,66	69,38	79,43
NCU	50	250	56	1712

É importante notar que para uma canal apenas, os dois métodos de implementação são equiparáveis em termos de lógica. Porém, para oito canais, o espaço lógico ocupado pela implementação direta sequencial é muito maior, dificultando a inserção de futuros blocos de *hardware*. Conforme cresce o número de canais de medição, a implementação via processador embarcado consome mais memória da FPGA, pois maior é o número de instruções e dados utilizados nas suas respectivas memórias, mas o consumo de blocos DSP é sempre constante.

Este exemplo deixa clara a superioridade da implementação pelo método com processador embarcado.

3.6. Conclusões do capítulo

Neste capítulo foram vistos tópicos referentes à FPGA, sua Linguagem de Descrição de *Hardware* (HDL) e métodos de implementação de algoritmos de DSP. Deu-se enfoque à linguagem Verilog.

Alguns métodos de implementação de algoritmos de DSP foram vistos, nos quais utilizou-se a implementação do filtro passa-altas da *wavelet* do sistema de compressão proposto, como exemplo. Dentre os métodos vistos, o que se destacou em termos de economia de lógica e performance, foi o método que utiliza o processador embarcado.

A utilização do processador implica em retardo da resposta, uma vez que ele executa as instruções sequencialmente. Porém, isso não se torna um obstáculo para a FPGA, pois na maior parte dos problemas de qualidade de energia, as taxas de amostragem se limitam a uma faixa

de algumas unidades de kilo-Hertz, enquanto a FPGA permite um funcionamento com algumas dezenas, e às vezes centenas, de Mega-Hertz.

Em relação à frequência máxima de operação, pode-se dizer que a implementação com processador embarcado reduziu $F_{máx}$, quando comparado a apenas um filtro nas formas de *hardware* dedicado (formas diretas e sequencial), porém manteve-se acima a taxa de *clock* proposta inicialmente, que é de 32 MHz.

O processador mostrou-se mais econômico do que a implementação sequencial, em termos de custo-benefício. E ainda, ele pode implementar qualquer algoritmo ou lógica, diferente das implementações em *hardware* dedicado, que permitem pouca flexibilidade. Essa abordagem mostra que o processador é a melhor forma de se implementar algoritmos de DSP para o projeto proposto.

Enfim, o método de implementação com processador se mostrou mais adequado para a realidade existente no projeto de compactação e de detecção proposto, por isso se tornou um apto substituto para as implementações em *hardware* dedicado.

4. IMPLEMENTAÇÃO DO SISTEMA DE DETECÇÃO E COMPRESSÃO DE DISTÚRBIOS ELÉTRICOS

4.1. Introdução

O Sistema de Detecção e Compressão de Distúrbios Elétricos (SDCDE) proposto neste trabalho, baseia-se, dentre outros, nos conceitos vistos anteriormente. Este capítulo apresenta como foram realizadas as implementações do sistema, dando enfoque à plataforma FPGA. Inicialmente na Seção 4.2, mostra-se uma visão ampla do sistema completo. Em seguida, na Seção 4.3 são descritos os blocos implementados na FPGA com mais detalhes. Já na Seção 4.4 é apresentado o protótipo funcional produzido para testes. Por fim, algumas conclusões sobre o capítulo são feitas na Seção 4.5.

4.2. Visão geral do sistema

A detecção realizada pelo SDCDE baseia-se no conceito de novidade, conforme visto na Seção 2.4. A compactação dos dados baseia-se em três níveis de compressão:

- Identificação da novidade[5]: utiliza-se a diferença das energias estimadas entre os *frames* (Subseção 2.4.2). Dessa forma, armazena-se apenas as amostras coletadas dos *frames* de novidade. Dos outros *frames* são retiradas apenas algumas poucas informações necessárias para a reconstrução. Dentre estas informações estão as quatro frequências médias de cada ciclo por *frame* do sinal (Seção 2.3). A reconstrução é realizada a partir do formato do *frame* de referência (último *frame* de novidade detectado), juntamente com as frequências médias estimadas. Outros parâmetros inerentes ao *frame* ajudam a identificar novidades, dentre eles estão o valor da diferença de frequências entre os *frames* e a diferença da energia do conteúdo harmônico estimada dos *frames* (THD).

- Compactação por *wavelet*: as amostras dos *frames* de novidade ainda passam por outro processo de compactação antes de serem armazenadas. Utiliza-se a decomposição

[5] Apesar da teoria de detecção de distúrbios elétricos abranger uma área muito mais ampla que a compressão de sinais, neste trabalho a detecção é tratada como um estágio de compressão pois a partir dela retiramos do sinal apenas os *frames* nos quais houveram detecções de novidades, contribuindo diretamente para a compressão.

wavelet, onde aplica-se um limiar, anulando-se os coeficientes abaixo do mesmo (Subseção 2.2.4). Assim, insere-se zeros nos dados e prepara-se o arquivo para ser entregue ao próximo estágio de compactação.

- Compactação em termos de bit: para o estágio final de compressão, utiliza-se a implementação do algoritmo de compactação Lempel-Ziv-Welch (Subseção 2.2.3).

As implementações em FPGA fazem parte de um sistema maior, que, em conjunto com outras plataformas, compõem o SDCDE. O sistema possui também, outras funcionalidades, como a capacidade de armazenar os dados num cartão micro SD, uma interface Android® para comando, controle e parametrização via *Bluetooth*, inserção de estampa de tempo e localização. Essas funcionalidades tornam o sistema mais inteligente. Para abrangê-las, o SDCDE faz uso de outras plataformas auxiliares, que realizam o gerenciamento dessas tarefas. No entanto, os algoritmos de processamento de sinais foram implementados na plataforma FPGA.

A Figura 78 mostra uma visão geral dos blocos envolvidos na composição do sistema, destacando os algoritmos presentes na metodologia proposta para a FPGA. Nela é possível ver em quais áreas das etapas de compressão a FPGA está diretamente envolvida e em quais, faz papel secundário. As descrições a seguir são referentes a esta figura.

Para ser possível o acoplamento do módulo de processamento à rede de energia, é necessário o uso do módulo analógico de condicionamento (1). Esse módulo tem a função de adequar as medições dos parâmetros aos níveis apropriados para o módulo de processamento. Nela estão contidos, dentre outros elementos, os Transformadores de Corrente (TCs) e Transformadores de Tensão (ou de potencial – TPs). O SDCDE, foi desenvolvido com o intuito de ser aplicado em medições de parâmetros de Sistemas Elétricos de Potência (SEPs), porém, nada o impede de ser ajustado e calibrado para ser usado em sistemas com diferentes portes e grandezas.

O módulo de processamento recebe as medições do módulo de condicionamento e as converte, por meio de um conversor analógico-digital (2).

Os dados convertidos são recebidos pela FPGA, para serem processados. Esse processamento começa com a divisão do sinal em *frames* (3). Os *frames* são então submetidos ao algoritmo de estimação da frequência fundamental (4), algoritmos responsáveis pela detecção de novidade (5) e pela decomposição em *wavelet* (6).

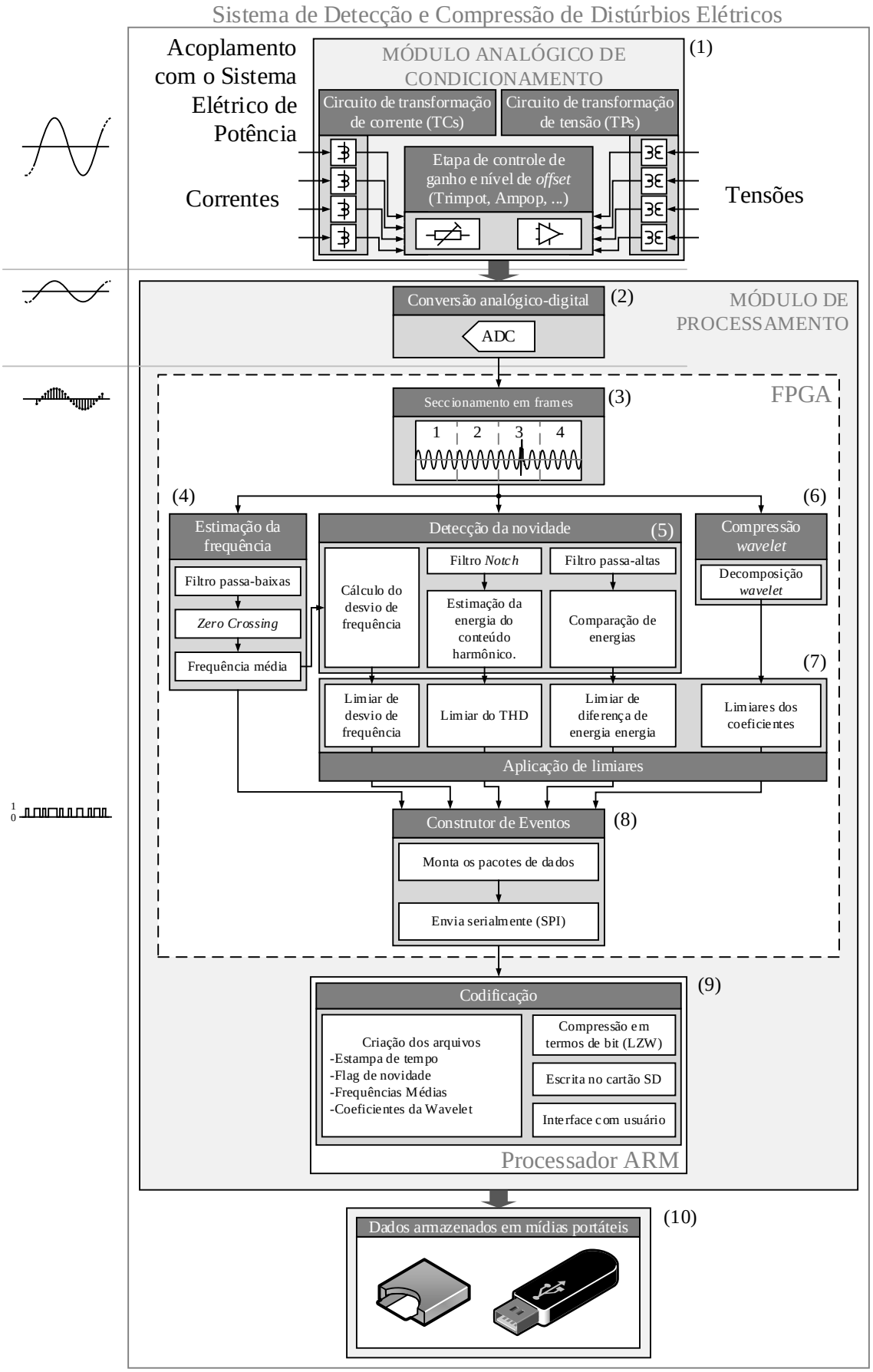


Figura 78: Representação de todo o sistema de detecção e compressão de distúrbios elétricos.

O método de detecção de novidades por diferenças de energias, foi propositalmente projetado para não indicar novidades por desvios de frequência, deixando essa tarefa, exclusivamente para o algoritmo de estimação de frequência. Obviamente, desvios muitos abruptos e agressivos na frequência, não muito comuns aos SEPs, podem provocar indicações de novidades em ambos os algoritmos. As informações de frequência, portanto, possuem dupla finalidade: são necessárias tanto para a reconstrução dos *frames* que não contém novidade, como para auxiliar na detecção da mesma.

A metodologia de detecção e compactação proposta, prevê um conjunto de quatro limiares aplicados às diferentes informações retiradas dos *frames* do sinal (7). Três deles são relacionados à detecção e um é para compressão.

O limiar de compressão é, como dito anteriormente, aplicado aos coeficientes da *wavelet* (extrema direita do bloco (7) “Aplicação de limiares” na Figura 78), anulando os coeficientes que estão abaixo dele, como mostra a representação na Figura 79. É importante ressaltar que a aplicação de limiares de compressão é feita apenas sobre os coeficientes de detalhe da *wavelet* e não sobre os coeficientes de aproximação.

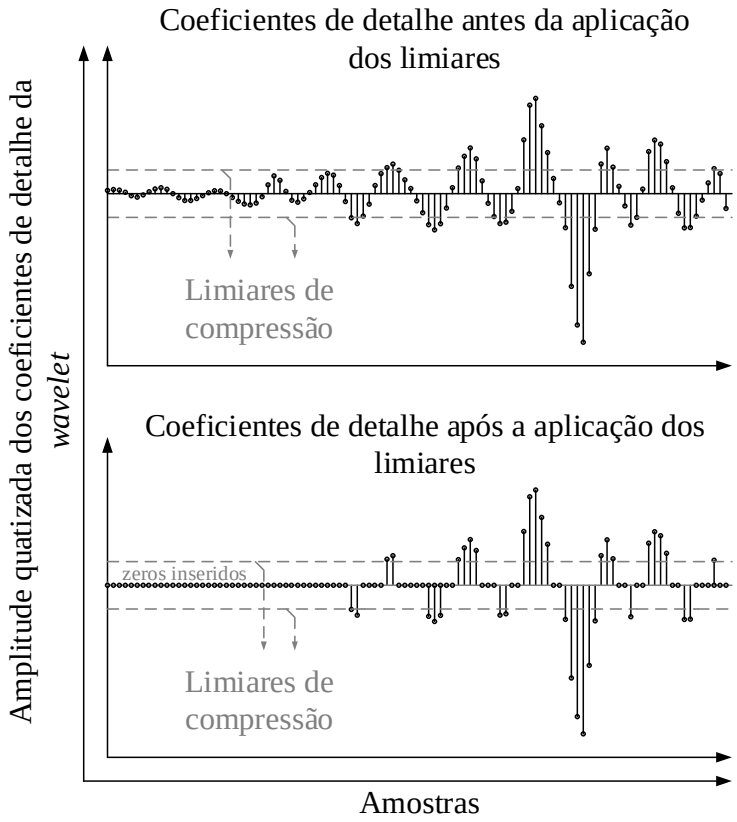


Figura 79: Representação da aplicação de limiares de compressão aos coeficientes de detalhe da *wavelet*.

Os limiares referentes à detecção estão discriminados a seguir:

- Limiar de diferença de energias: para uma detecção por energia ocorrer, a diferença de energias entre os *frames* atual e o de referência deve ultrapassar o limiar estabelecido para a diferença de energia.
- Limiar de harmônicos: à estimação de energia do conteúdo harmônico do *frame*, também é aplicado um limiar (limiar do THD – *Total Harmonic Distortion*). Caso essa estimação ultrapasse o limiar específico para ela, também é detectada uma novidade.
- Limiar de desvio de frequência: o desvio de frequência entre os *frames* também é calculado, e então é aplicado um limiar específico a ele. Caso esse limiar seja ultrapassado, uma novidade por frequência é indicada.

Através dos exemplos a seguir é possível entender o funcionamento dos três limiares de detecção.

Exemplo 4.1 – Detecção por energia

Considere o sinal mostrado na Figura 80. O primeiro *frame* a ser gravado é o *frame* 1. Como não houve nenhum *frame* de detecção antes dele, o mesmo é considerado *frame* de novidade e, portanto, possui todas as suas amostras armazenadas e se torna o *frame* de referência. O *flag* de novidade é atualizado ao fim de cada *frame*. Como o *frame* 1 é um *frame* de novidade, o *flag* de novidade permanece em nível alto.

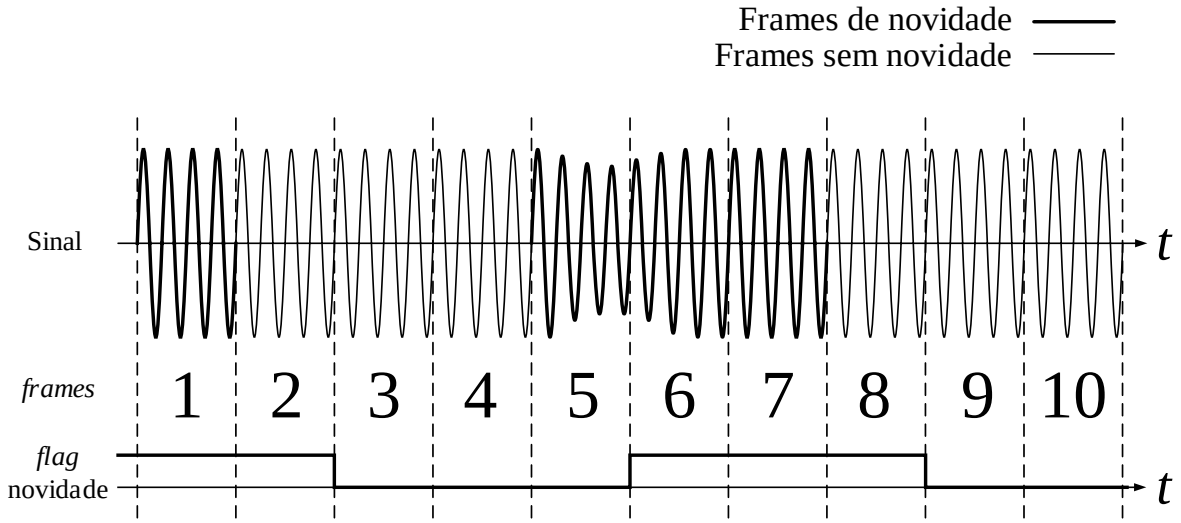


Figura 80: Ilustração da detecção de novidade por energia.

Comparando-se o *frame* 2 com o *frame* de referência (*frame* 1 por enquanto), verifica-se que não há diferença entre os mesmos. Assim, não é necessário armazenar as amostras dele, o que contribui para a compressão do sinal. Uma vez que o *frame* 2 não é um *frame* de novidade, o *flag* de novidade é colocado em nível baixo, indicando este fato.

O mesmo procedimento ocorre com os *frame*s 3 e 4 e os mesmos resultados são obtidos, isto é, os *frame*s são considerados sem novidade e o *frame* de referência ainda continua sendo o *frame* 1.

Quando o *frame* 5 é comparado com o *frame* 1 (*frame* de referência), nota-se que existe uma diferença entre eles, pois um distúrbio ocorreu no *frame* 5, o qual é considerado um *frame* de novidade e passa a ser o novo *frame* de referência. Ao fim do *frame* 5 o *flag* de novidade muda para nível alto indicando que o *frame* 5 é de novidade.

Ao se comparar o *frame* 6 com o *frame* 5 de referência, é constatada diferença entre eles. O *frame* 6 é considerado *frame* de novidade e assim passa a ser o novo *frame* de referência. O *flag* de novidade indica isso permanecendo em nível alto.

A situação se repete ao se comparar os *frames* 7 com o *frame* 6 e isso leva o *frame* 7 a ser o último *frame* de referência do intervalo observado, pois o mesmo é diferente do *frame* 6. Ao fim do *frame* 7 o *flag* de inovação permanece em nível alto.

Entretanto, ao se comparar o *frame* 8 ao *frame* 7, nenhuma novidade é detectada e assim ao final do *frame* 8 o *flag* de novidade volta ao nível baixo. Essa situação se mantém até o fim do intervalo observado, quando ainda tem-se o *frame* 7 como *frame* de referência e o *flag* de novidade em nível baixo.

A detecção por energia pode não detectar novidade em eventos onde a diferença entre a energia entre um determinado *frame* e o *frame* de referência não é tão expressiva. Tal situação ocorre, por exemplo, quando um conteúdo harmônico de pequena amplitude é adicionado ao sinal. Para contornar este problema existe o limiar de harmônicos.

Exemplo 4.2 – Detecção por variação do conteúdo harmônico

Considere o sinal mostrado na Figura 81. Nele, uma distorção harmônica é inserida a partir do meio do *frame* 6 e se mantém assim até o fim do intervalo de observação. A detecção de novidade pelo conteúdo harmônico opera da mesma forma como a detecção de novidades pela

energia do *frame*. Porém, ela considera apenas a energia do conteúdo harmônico do sinal de entrada, excluindo-se assim a energia da componente fundamental.

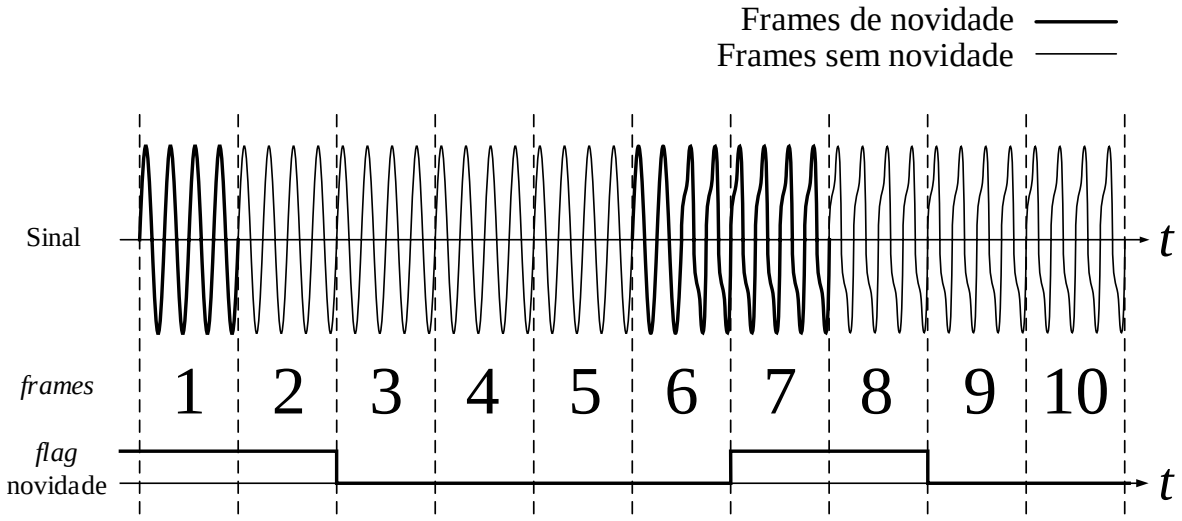


Figura 81: Ilustração de detecção de novidades por diferença de conteúdo harmônico.

No começo da observação, o *frame* 1 é tomado com *frame* de referência, indicado pelo *flag* de novidade em nível alto. Ao se comparar o *frame* 2 com o *frame* 1, percebe-se que ele não possui conteúdo harmônico diferente do *frame* 1, então ao final do *frame* 2 o *flag* de novidade é colocado em nível baixo indicando que o *frame* 2 não é um *frame* de novidade. Isso permanece até o *frame* 6, onde é inserida uma distorção harmônica.

Ao se comparar o *frame* 6 com o *frame* 1 de referência nota-se que o *frame* 6 possui conteúdo harmônico diferente do *frame* 1 e, portanto, é um *frame* de novidade e passa a ser o novo *frame* de referência. O *flag* de novidade é colocado em nível alto para indicar este fato.

Quando o conteúdo harmônico do *frame* 7 é comparado ao do *frame* 6 também existe diferença, visto que a distorção harmônica é inserida no meio do *frame* 6, ao passo que o *frame* 7 inteiro possui essa distorção. Assim, ao final do *frame* 7 o *flag* de novidade vai para nível alto indicando que o *frame* 7 é um *frame* de novidade e, portanto, o novo frame de referência.

Ao serem comparados os conteúdos harmônicos dos *frames* subsequentes ao do *frame* 7, percebe-se que não existe diferença, uma vez que a distorção harmônica se manteve. Sendo assim, a partir do *frame* 8 já não são detectadas mais novidades.

O processo de reconstrução necessita de quatro informações da frequência fundamental por *frame* para realizar a reconstrução. Essas informações são extraídas do bloco de estimação de frequência (bloco (4) da Figura 78).

Sabe-se que devido à forte inércia presente no sistema elétrico de potência, a frequência fundamental não sofre desvios abruptos. Sabe-se também, que algoritmos de estimação de frequência geralmente sofrem interferências na presença de distúrbios. Sendo assim, caso não seja detectada nenhuma novidade proveniente da detecção por energia nem pelo conteúdo harmônico, mas seja constatado um desvio expressivo no valor da estimação da frequência, subentende-se esse desvio não é real, mas sim um erro por parte do processo de estimação e essa informação de frequência não poderá ser usada para a reconstrução.

Para contornar este problema novidades devem ser detectadas em *frames* que eventualmente apresentarem desvios de frequência acima de um limiar pré-estabelecido. O exemplo a seguir ilustra esta situação.

Exemplo 4.3 – Detecção por variação repentina na estimação da frequência fundamental

Suponha que o sinal da Figura 82 tenha sua frequência fundamental estimada. Pode-se ver que algoritmo de estimação da frequência fundamental sofre interferência no segundo ciclo do *frame* 3 ($f_{2,3}$). A estimação da frequência foi comprometida para este ciclo e para que ela não seja usada no processo de reconstrução, todo o *frame* 3 é considerado como *frame* de novidade.

Esse tipo de detecção não utiliza comparação de informações entre *frames* como é feito nas detecções por energia e por conteúdo harmônico, mas compara-se a estimação de frequência de cada ciclo com um valor fixo. Caso a diferença seja maior do que o limiar de desvio de frequência todo o *frame* é considerado como *frame* de novidade, preservando-se o *frame* para a reconstrução.

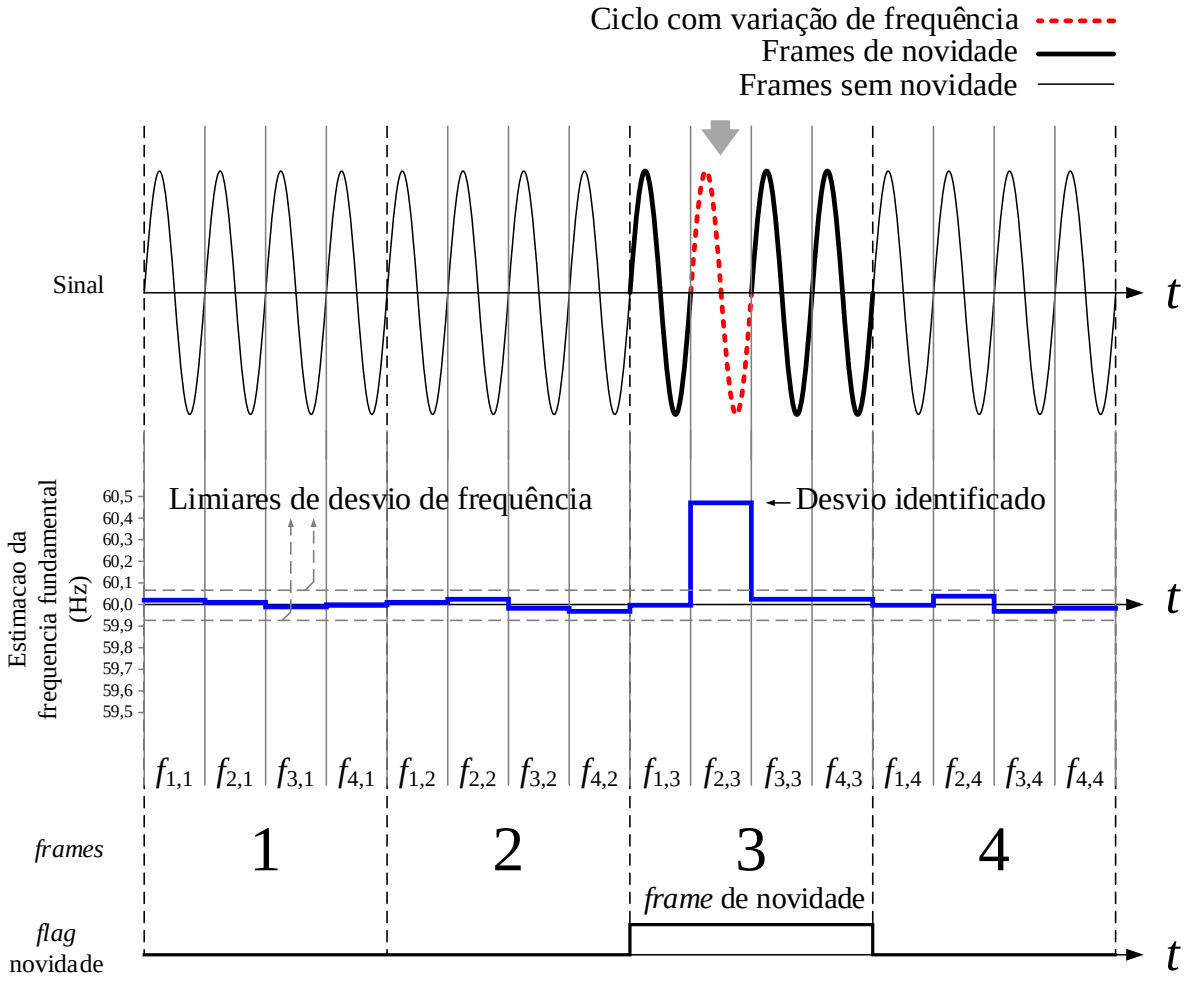


Figura 82: Detecção por variação abrupta na frequência.

Dos três limiares de detecção de novidade descritos anteriormente, o principal é o de detecção por energia, os outros dois são auxiliares.

Sendo assim, caso qualquer limiar de detecção seja ultrapassado, é indicado um *flag* de novidade; um bit, que assume o valor 1, caso haja novidade, e 0, caso ela não exista. Podem, entretanto, existir eventos que provoquem indicações em mais de um tipo de *flag* de novidade simultaneamente. Ainda existem outros fatores que promovem indicações de *flags*, os quais serão descritos mais à frente. Todos os *flags* de novidade são agrupados em uma porta lógica do tipo “OU”. Dessa forma, para que seja indicada uma novidade é necessário que, no mínimo, um dos *flags* de novidade seja disparado.

Todas as informações provenientes dos blocos de estimação, detecção e compressão são encaminhadas a um bloco denominado Construtor de Pacotes (8), cuja função é organizar as informações, montar os pacotes de dados e enviá-los serialmente via protocolo de interface SPI, à próxima e última etapa de compressão.

A última etapa de compressão (LZW), é realizada fora da FPGA, com o auxílio de outra plataforma, um processador ARM (9). Este, tem a função de criar os arquivos a serem escritos no cartão SD, contendo as informações do sinal. O processador é responsável também pelo gerenciamento da comunicação e interface com o usuário.

Ao final do processo, tem-se os arquivos escritos num dispositivo de armazenamento portátil, cartão SD ou *pen drive* (10), o qual pode ser plugado a um computador, onde os dados podem ser descompactados com o auxílio de um *software* adequado.

4.3. Implementações em FPGA

Como o foco deste trabalho são as implementações na plataforma FPGA, a parte englobada pelo tracejado, na Figura 78, será descrita mais detalhadamente nesta seção. A Figura 83 mostra como estão agrupados os blocos de *hardware* que realizam as diversas tarefas dentro da FPGA. As subseções seguintes tratam de descrever cada bloco dessa figura.

4.3.1. PLL

O bloco PLL (1 na Figura 83) é responsável pela geração dos sinais de *clock*. Para implementá-lo, foi utilizada uma IP (*Intellectual property*) gratuita da ALTERA®, presente no catálogo de IPs disponível no Quartus II®.

As configurações desse bloco são feitas por meio de um *wizard*. Cada bloco de PLL pode admitir até cinco saídas de *clock* independentes. Foram selecionadas apenas duas saídas. A primeira, `c0`, com uma frequência de 32 MHz, utilizada para os blocos gerais do sistema, e a segunda, `c1`, com uma frequência de 2,5 MHz, utilizada especificamente para a comunicação SPI.

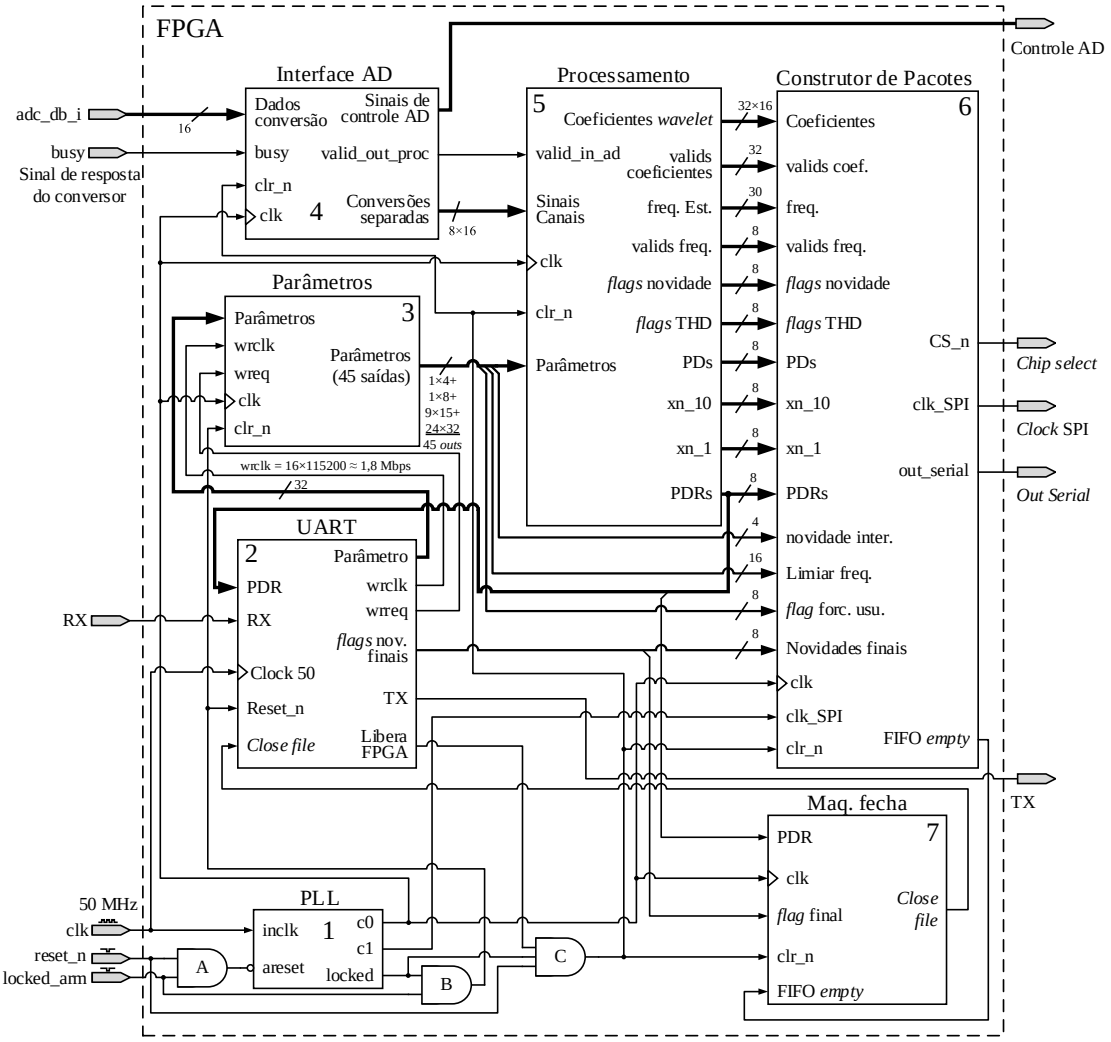


Figura 83: Blocos de *hardware* dentro da FPGA.

O bloco permite uma entrada de *clear* assíncrono, `areset`, a qual, quando ativada, impede o bloco de fornecer sinais de *clock* ao sistema.

O comando `locked_arm` é dado pelo usuário no momento de abertura e fechamento de arquivos. Se ele estiver em 0, significa que não foi dado comando de abertura de arquivo e por isso a FPGA deve estar em estado de *reset*, aguardando um comando de abertura. A porta lógica "AND" `A` permite que, tanto a entrada `locked_arm`, quanto o pino de *reset* manual `reset_n`, tenham o efeito desejado, paralisando a FPGA.

A saída `locked` tem a função de indicar quando a PLL atingiu estabilidade nas frequências de saída exigidas. Por isso o sistema é mantido em estado de *reset*, através das portas "AND" `B` e `C`, enquanto a estabilidade não é atingida.

4.3.2. UART

O bloco UART - *Universal asynchronous receiver/transmitter* (2 na Figura 83), possui a função de permitir a comunicação entre a FPGA e o dispositivo móvel Android®, conectado via *Bluetooth*, à placa de processamento. A plataforma que faz a intermediação entre o dispositivo móvel e a FPGA é o processador ARM, gerenciando a comunicação *Bluetooth*. O *clock* de entrada do bloco UART não provém da PLL, mas diretamente do pino de entrada de *clock* com uma frequência de 50 MHz.

Antes de ser enviado o comando de abertura de arquivo pelo usuário, através do dispositivo móvel, a FPGA permanece em estado de *reset* (pino `locked_arm`). Ao ser enviado um comando de abertura, a PLL é liberada, mas antes que o arquivo seja efetivamente aberto, os parâmetros de configuração do SDCDE são enviados à FPGA, um a um, por meio do pino de entrada RX, no bloco UART. Por isso, existe uma saída denominada `Libera FPGA`, que mantém o resto do sistema em *reset*, enquanto os parâmetros são carregados aos registradores dentro do bloco Parâmetros (3). Nesse momento, apenas os blocos UART e Parâmetros estão habilitados. A saída TX, do bloco UART, serve para enviar os valores das posições dos registradores a serem preenchidos com os parâmetros recebidos. Primeiro envia-se ao dispositivo móvel o número do registrador, e então o dispositivo móvel responde de volta com o valor do parâmetro requerido. Quando o parâmetro é recebido pela FPGA, além de ser armazenado, é enviado de volta ao dispositivo móvel, a fim de se corrigir algum erro de transmissão, caso exista. Isso é feito pelo mesmo pino TX.

Cada parâmetro que chega à FPGA possui 32 bits de largura, porém, por questões técnicas, são enviadas palavras de 1 byte de largura por vez. Assim, é necessário um sub-bloco para gerenciar esse recebimento e fazer a devida concatenação de 4 bytes por parâmetro. Essa tarefa é realizada por uma máquina de estados presente dentro do bloco UART.

4.3.3. Parâmetros

Cada parâmetro recebido pelo bloco UART, está associado a um registrador dentro do bloco Parâmetros (3 na Figura 83). Antes de serem armazenados nos registradores, os valores dos parâmetros são escritos numa memória FIFO de tamanho 4×32 bits, situada na entrada do bloco (FIFO_ent na Figura 84). Essa memória é do tipo *dual-clock*. O sinal de *clock* de escrita (`wrclk`)

e o comando de requisição de escrita (`wrreq`) são fornecidos pelo bloco UART. O *clock* de leitura (`rdclk`) é o próprio sinal de *clock* do sistema (`c0`, vindo da PLL).

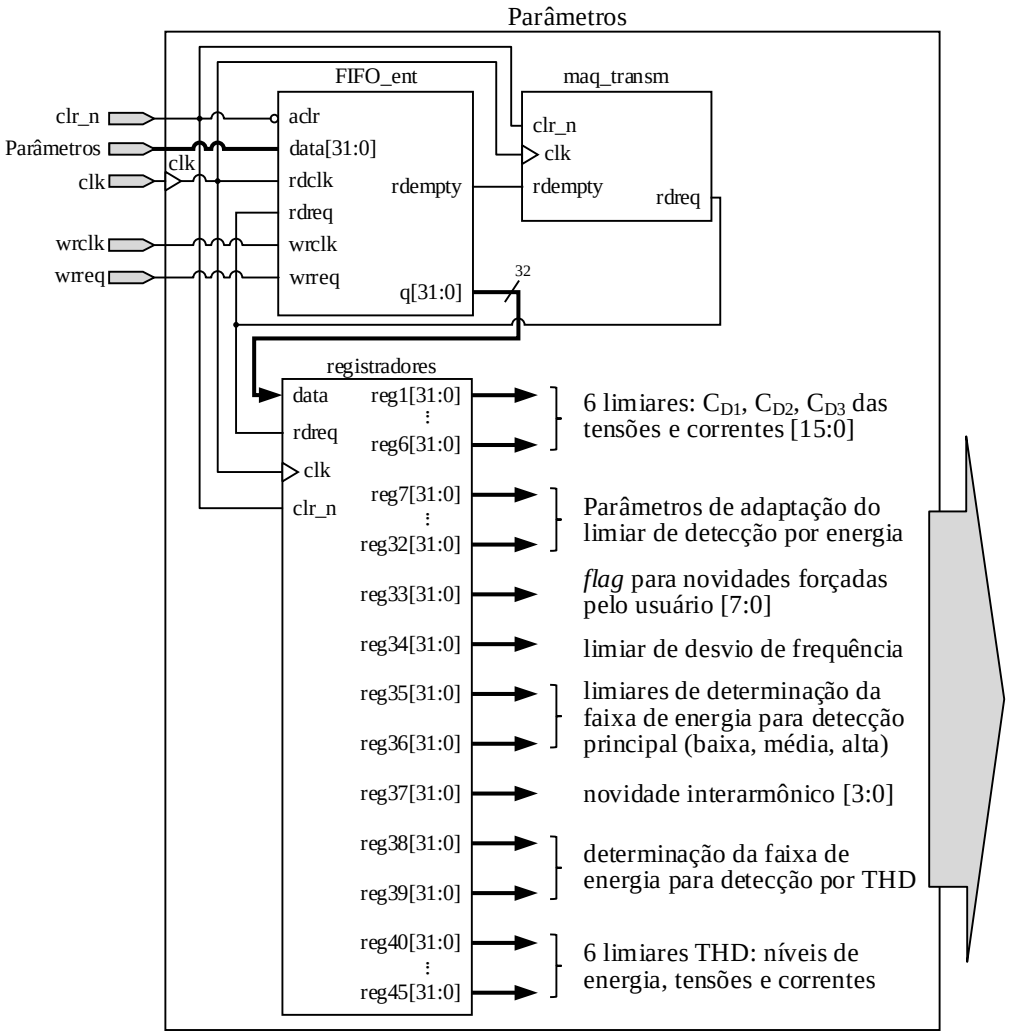


Figura 84: Estrutura interna do bloco parâmetros.

Após a escrita na memória FIFO, os parâmetros são conduzidos aos seus respectivos registradores. O comando de requisição de leitura da memória FIFO (`rdreq`) é enviado por uma máquina de estados que controla a transmissão do parâmetro da memória para o registro (`maq_transm` na Figura 84). A máquina observa o *bit* que representa se a memória está ou não vazia (`rdempty`). Caso este sinal assuma nível lógico baixo, significa que ela não está mais vazia e já pode ser lida. A máquina então envia um comando de leitura para a memória e transfere o dado para o registrador. Ao todo, são 45 parâmetros de configuração. A Figura 84 mostra uma representação do bloco Parâmetros.

Esses parâmetros, enviados pelo usuário ao SDCDE têm as seguintes finalidades:

- Estabelecer limites para os cortes dos coeficientes da *wavelet*: esses valores são os primeiros seis registradores. Os limiares de tensão são separados dos de corrente.

- Estabelecer limites para a adaptação do limiar de detecção por diferença de energia: os registradores 7 a 32 fornecem vários parâmetros para limitar a curva de adaptação do limiar de comparação das diferenças de energias. O comportamento dessa curva de adaptação, juntamente com seus respectivos parâmetros de configuração, podem ser vistos na Figura 85.

- Enviar comando para manter detecção em todo o tempo em determinados canais: em algumas situações, pode ser desejável manter um canal com novidades em todos os *frames*. Desse modo a compactação por detecção sobre ele não terá efeito. Para configurar, por exemplo, o canal 1, como tendo novidades a todo o tempo, basta enviar o parâmetro [10000000] para o Registro 33, ou se for para o terceiro canal, o valor deve ser [00100000] no mesmo registro.

- Determinar o valor para o limiar de desvio de frequência: através desse registro, define-se o valor a ser comparado ao cálculo do desvio de frequências entre os *frames* atual e de referência;

- Definir os valores que delimitam faixas de energia do sinal de entrada: além do limiar de detecção por energia ser adaptativo, a curva de adaptação não é única. Existem três curvas possíveis, relacionadas à energia: baixa, média e alta. Se a energia estimada for maior que o valor contido no registro 35, é considerada alta. Caso seja menor que o valor contido no registro 36, é considerada baixa. Se não estiver em nenhum dos casos anteriores, é considerada média. Dependendo do valor de energia do sinal de entrada uma das curvas de adaptação será escolhida.

- Escolher quantos *frames* de novidade devem ser armazenados por detecção (1, 3 ou 15): essa escolha auxilia na reconstrução de sinais com presença de inter-harmônicos;

- Estabelecer faixas de energia para detecção por THD: do mesmo modo que na detecção por energia existem faixas de energia do sinal de entrada, na detecção por THD, também são determinados esses limiares;

- Determinar o valor dos limiares de THD: esses registros são os valores dos limiares do THD por faixa de energia.

A maioria dos parâmetros é enviada para o bloco Processamento. Alguns, porém, são enviados diretamente para o bloco Construtor de Pacotes.

A Figura 85 mostra um exemplo de adaptação do patamar de comparação para detecção por energia. A curva inferior representa a atualização do patamar utilizado para a comparação com a diferença de energias entre os *frames*.

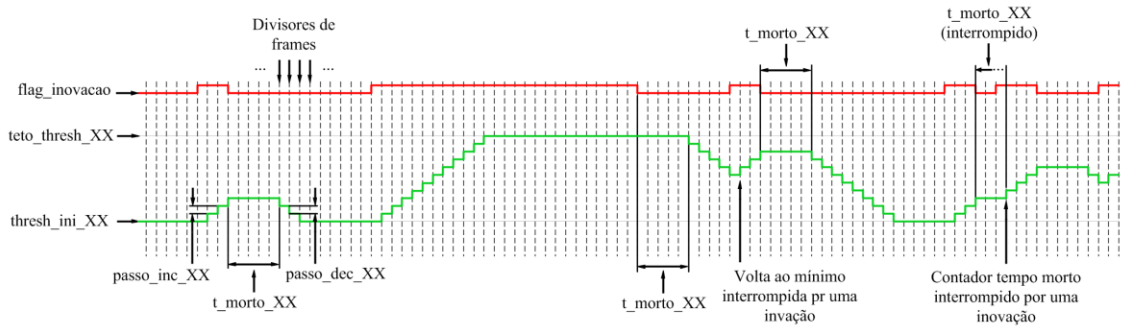


Figura 85: Adaptação do patamar de detecção por diferença de energia.

Os parâmetros que controlam essa curva são:

- `thresh_ini`: valor inicial que o patamar assume. Esse valor é o mínimo possível para a faixa de energia utilizada;
- `teto_thresh`: valor máximo assumido pelo patamar;
- `passo_inc`: passo de incremento;
- `passo_dec`: passo de decremento;
- `t_morto`: tempo de espera antes de se atualizar o valor do patamar.

Quando várias novidades são detectadas consecutivamente, o valor do patamar é incrementado, visando reduzir as detecções. Quando estas sessam, o patamar é atualizado após o tempo morto, sendo decrementado até o mínimo, caso não haja nenhuma detecção.

4.3.4. Interface com o conversor AD

O bloco Interface AD (4 na Figura 83) serve para controlar o funcionamento do conversor AD utilizado, coletar as suas conversões e disponibilizá-las ao resto do sistema. O conversor utilizado no projeto do SDCDE, é o conversor AD7606 da empresa fabricante ANALOG DEVICES®, e possui capacidade de converter até oito canais simultaneamente, com resolução de 16 bits. Assim, o bloco Interface AD possui uma entrada de 16 bits (`adc_db_i`), vinda do conversor, pela qual passam os dados convertidos. Além das entradas de *clear* e de *clock* existe uma outra entrada que vem do conversor (`busy`), a qual indica que o mesmo está ocupado realizando as conversões. As saídas existentes nesse bloco são os sinais de controle e os dados convertidos.

A Figura 86 mostra os sub-blocos existentes dentro do bloco Interface AD. O conversor AD utiliza o mesmo barramento de 16 bits para pôr os dados das conversões de todos os canais. Para indicar a qual canal corresponde cada conversão, o sub-bloco Controle AD fornece uma saída, `data_rd_ready_o`, em forma de pulsos. Cada pulso corresponde a um resultado de conversão de um canal no barramento `data_o`. O sub-bloco Controle AD foi construído adaptando-se uma IP fornecida pelo próprio fabricante do conversor.

O sub-bloco Demux AD é responsável por separar cada pulso vindo do sinal `data_re_ready_o`, correspondente ao respectivo canal, e fornecer os sinais de escrita (`valid_FIFO1` a `valid_FIFO8`) para que as oito memórias FIFOs existentes nos blocos m_FIFO1 a m_FIFO8, recebam os dados de seu canal, individualmente. A Figura 87 mostra essa demultiplexação.

Cada bloco m_FIFO, possui dentro de si uma memória FIFO com 4 posições de 16 bits cada, e uma pequena máquina de estados para controlar a leitura e gerar os pulsos de `valid_out`. Esses pulsos vão para 1 quando o dado da conversão está disponível no barramento de saída `data_out`.

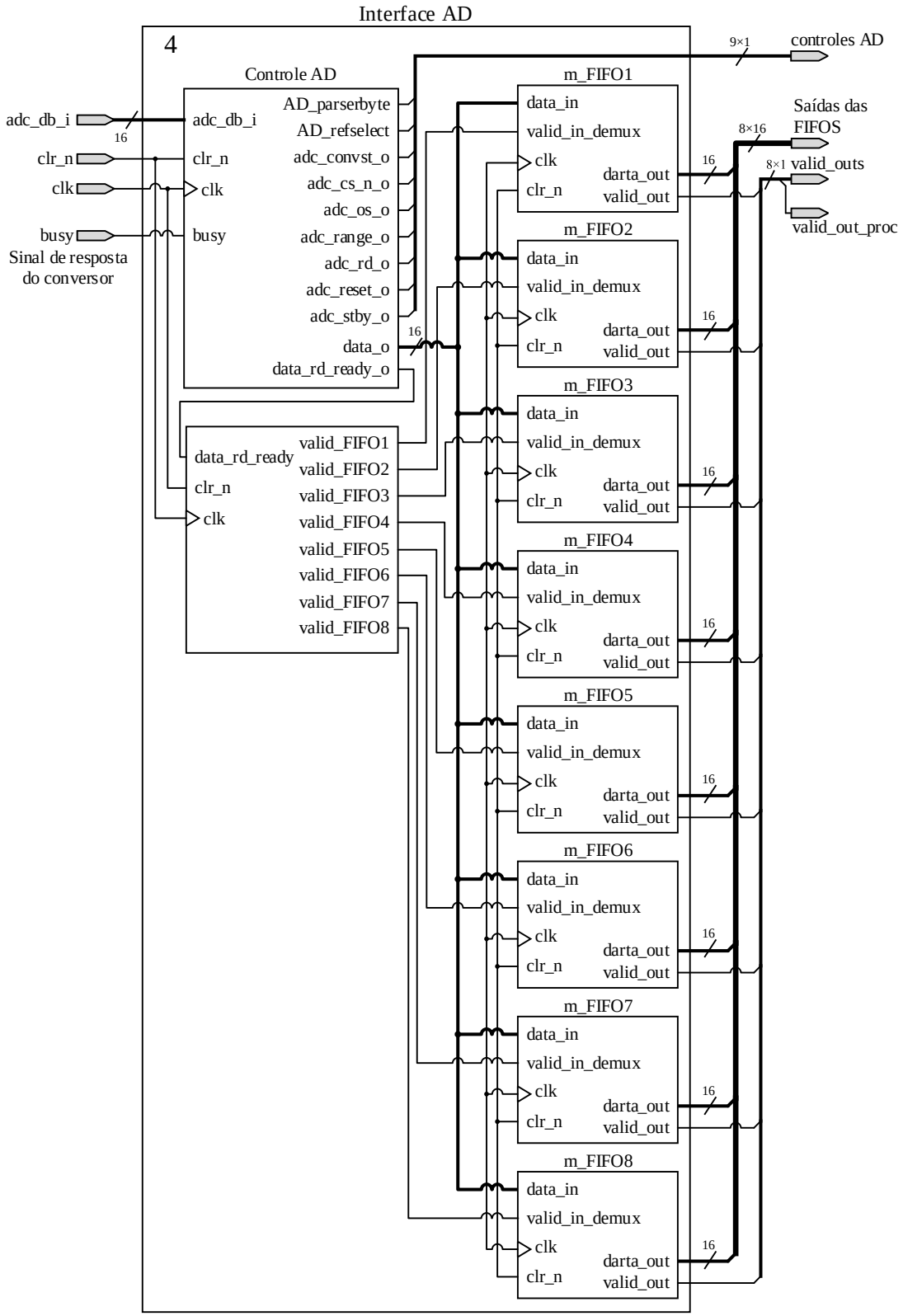


Figura 86: Estrutura interna do bloco Interface AD.

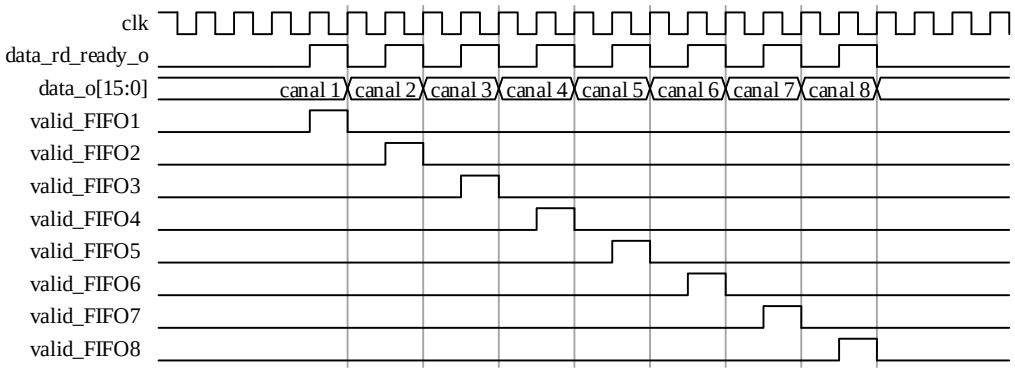


Figura 87: Comportamento das saídas do sub-bloco Demux AD.

Todas as memórias FIFO existentes no projeto, foram implementadas utilizando-se uma IP da ALTERA®, da mesma forma como foi feito com a PLL.

Por fim, as saídas do Interface AD são os sinais de controle para o conversor AD, os oito dados de conversões e os oito sinais de `valid_out`, um para cada canal. Quando todos os dados convertidos estão disponíveis para leitura, o sinal `valid_out` da última memória FIFO preenchida é utilizado como sinal de `valid_out_proc`, conectado na entrada `valid_in` do bloco de processamento.

Para um melhor entendimento das funções de cada um dos sinais de controle presentes no bloco de controle do conversor AD, é necessária uma breve consulta ao Apêndice BApêndice B. Nele também estão descritos outros detalhes importantes do funcionamento do conversor, como função dos pinos, modos de sobreamostragem, dentre outras informações.

4.3.5. Processamento

O bloco Processamento (5 na Figura 83) contém quatro processadores iguais ao que foi visto na Subseção 3.5.4. Cada um possui uma finalidade específica. O primeiro, realiza o processo da detecção pela diferença de energia (Detector por Energia na Figura 88). O segundo estima a frequência média fundamental do sinal de entrada (Estimador de Frequência). O terceiro, estima o conteúdo harmônico e aplica os limiares de THD (Estima THD). Finalmente o quarto, realiza a decomposição *wavelet*. A Figura 88 mostra os quatro processadores dispostos dentro do bloco Processamento. Esse bloco é capaz de executar os oito canais em sequência.

A detecção por energia é feita aplicando-se um filtro passa-altas ao sinal de entrada. A saída desse filtro tem a energia estimada e comparada ao *frame* de referência pelo método da diferença entre as energias (Subseção 2.4.2). O processador de estimação da frequência fundamental implementa o método do cruzamento por zero (Subseção 2.3.2). A estimação do conteúdo harmônico é realizada pelo cálculo da energia da saída de um filtro *Notch* aplicado ao sinal de entrada. Esse filtro é ajustado pela estimação de frequência vinda do processador Estimador de Frequência.

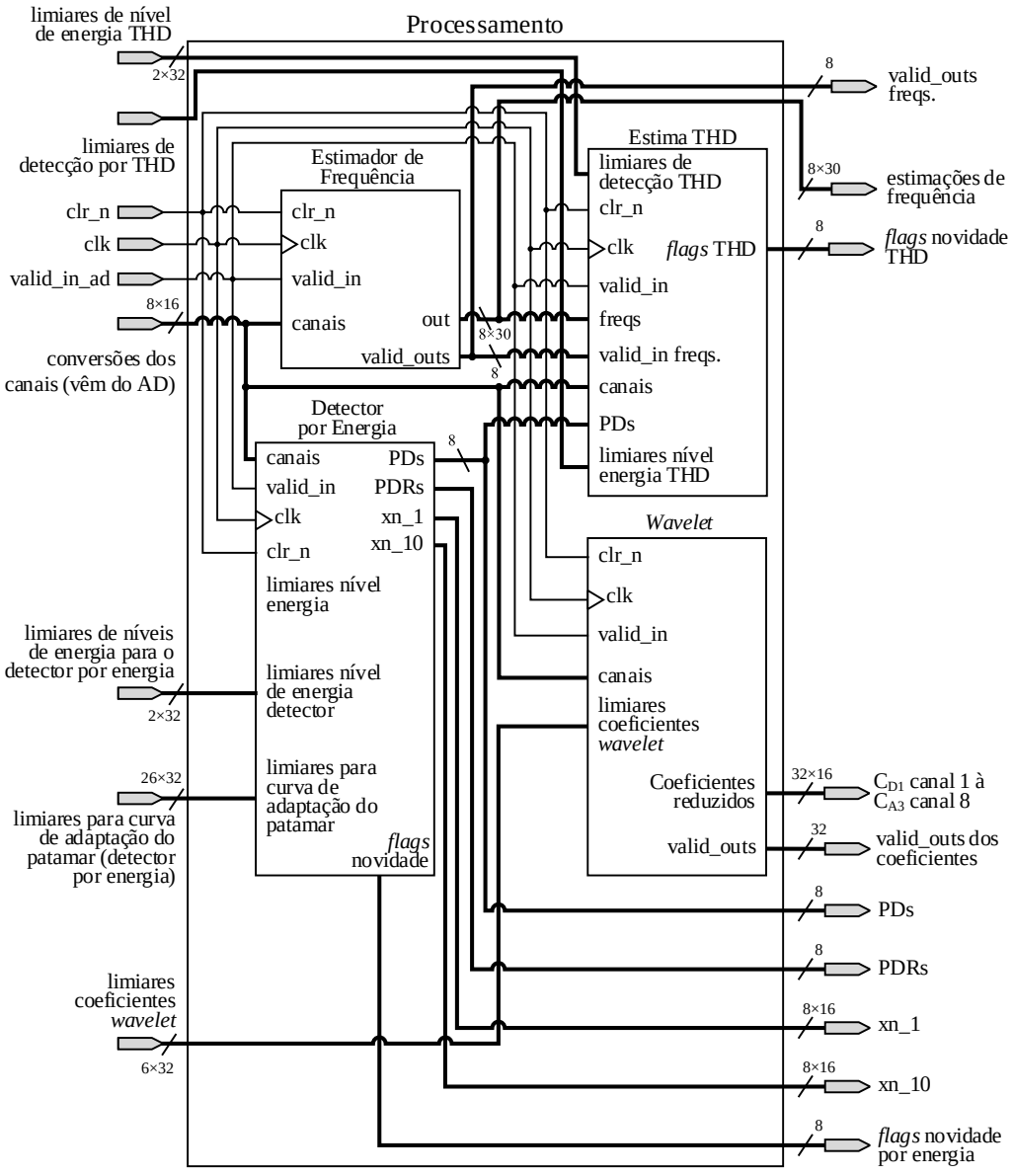


Figura 88: Representação do bloco Processamento.

Sempre que são concluídas as conversões solicitadas ao conversor AD, novas oito amostras (uma por canal) entram no bloco de processamento através da entrada “conversões dos canais”.

Para isso um pulso de `valid_in_ad` é dado. Esse mesmo pulso é usado como *reset* de todos os processadores. Dessa forma, reinicia-se os algoritmos de processamento sempre que novas amostras entram. A reinicialização, porém, não reinicializa os dados calculados na amostra anterior, permitindo a continuidade do funcionamento dos algoritmos, sempre que novas amostras chegam.

O seccionamento do sinal em *frames* é realizado pelo processador Detector por Energia. Para isso, em seu programa, é utilizado um contador módulo 512 (contando de 0 → 511), que é incrementado a cada *reset*. Quando é atingido o valor máximo do contador, o processador realiza a comparação das energias, e indica em sua saída se houve, ou não, uma novidade.

Para fornecer qualquer saída, o processador possui três sinais: o primeiro é o barramento de saída, propriamente dito. O segundo é um sinal de habilitação de saída com 1 bit (`out_en`) ativo em nível alto, que indica que o dado presente no barramento de saída pode ser lido. O terceiro é um barramento de endereço para saída, utilizado para direcionar os valores dos dados de saída para diferentes registradores.

No momento em que o processador Detector por Energia expõe o resultado de uma comparação de energias entre *frames* no barramento de saída, o sinal de habilitação de saída é utilizado como separador dos *frames*. Esse sinal é denominado Pulso Divisor (PD). Cada canal possui o seu PD, formando ao todo oito PDs. Os sinais PDRs são os PDs registrados e, portanto, atrasados de um ciclo do *clock* do sistema. Eles são utilizados para o bloco Construtor de Pacotes. A Figura 89 mostra uma representação do seccionamento do sinal.

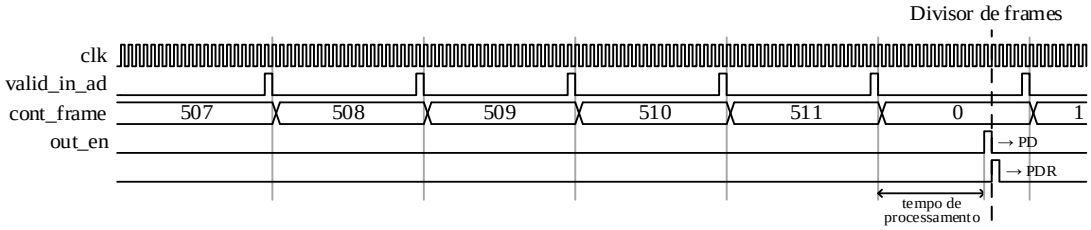


Figura 89: Representação do divisor de *frames*.

Outra função do processador Detector por Energia é forçar alguns poucos *frames* de novidades iniciais, sempre que são abertos novos arquivos, a fim de esperar que os transitórios do sistema desapareçam.

Quando se tem a transição de um período sem novidades detectadas para um *frame* com detecção de novidade, a metodologia de reconstrução necessita que dois pontos do último *frame* sem novidade sejam enviados: o último ponto, denominado `xn_1`, e o ponto atrasado em relação a este por nove amostras, `xn_10`. Esses pontos são necessários para auxiliar no processo e interpolação, existente na reconstrução. O processador Detector por Energia fornece esses pontos, um por canal. A representação desses pontos pode ser vista pela Figura 90.

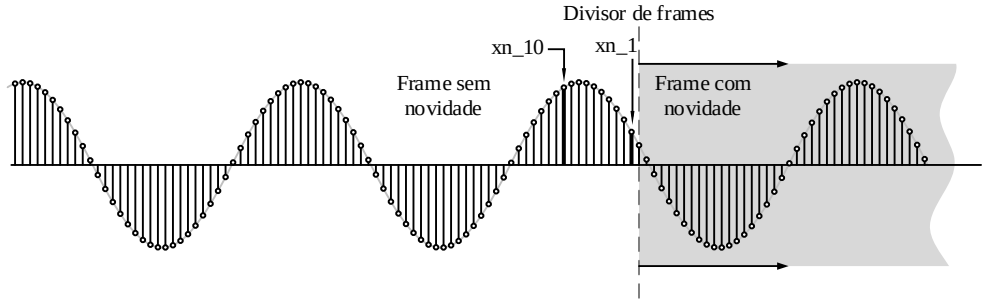


Figura 90: Representação dos pontos xn_1 e xn_10.

4.3.6. Construtor de Pacotes

O bloco Construtor de Pacotes (6 na Figura 83), é responsável por agrupar os dados vindos do bloco Processamento. Ele também realiza as detecções pela variação de frequência, aplicando o limiar de desvio de frequência às informações provenientes do processador Estimador de Frequência. Para isso, o parâmetro do registrador 34 (limiar de desvio de frequência) do bloco Parâmetros deve ser enviado diretamente ao Construtor de Pacotes. Outros dois parâmetros que também são enviados diretamente são os dos Registradores 33 (novidades forçadas pelo usuário) e 37 (novidade para inter-harmônico). Uma representação de sua estrutura interna do Construtor de Pacotes pode ser vista pela Figura 91.

O bloco possui oito unidades idênticas, denominadas Canal X (onde X representa o canal, X = 1, 2, ..., 8), que realizam a mesma tarefa, porém, com os dados relacionados ao seu respectivo canal. Na figura, são mostradas com mais detalhes apenas duas das oito unidades, a primeira (Canal 1) e a última (Canal 8). O bloco possui também, uma memória FIFO de 4096×16 bits, utilizada para montar os arquivos a serem enviados. O bloco SPI realiza a comunicação entre a FPGA e o processador ARM. Para preencher a memória FIFO com os dados dos canais, utiliza-se um multiplexador.

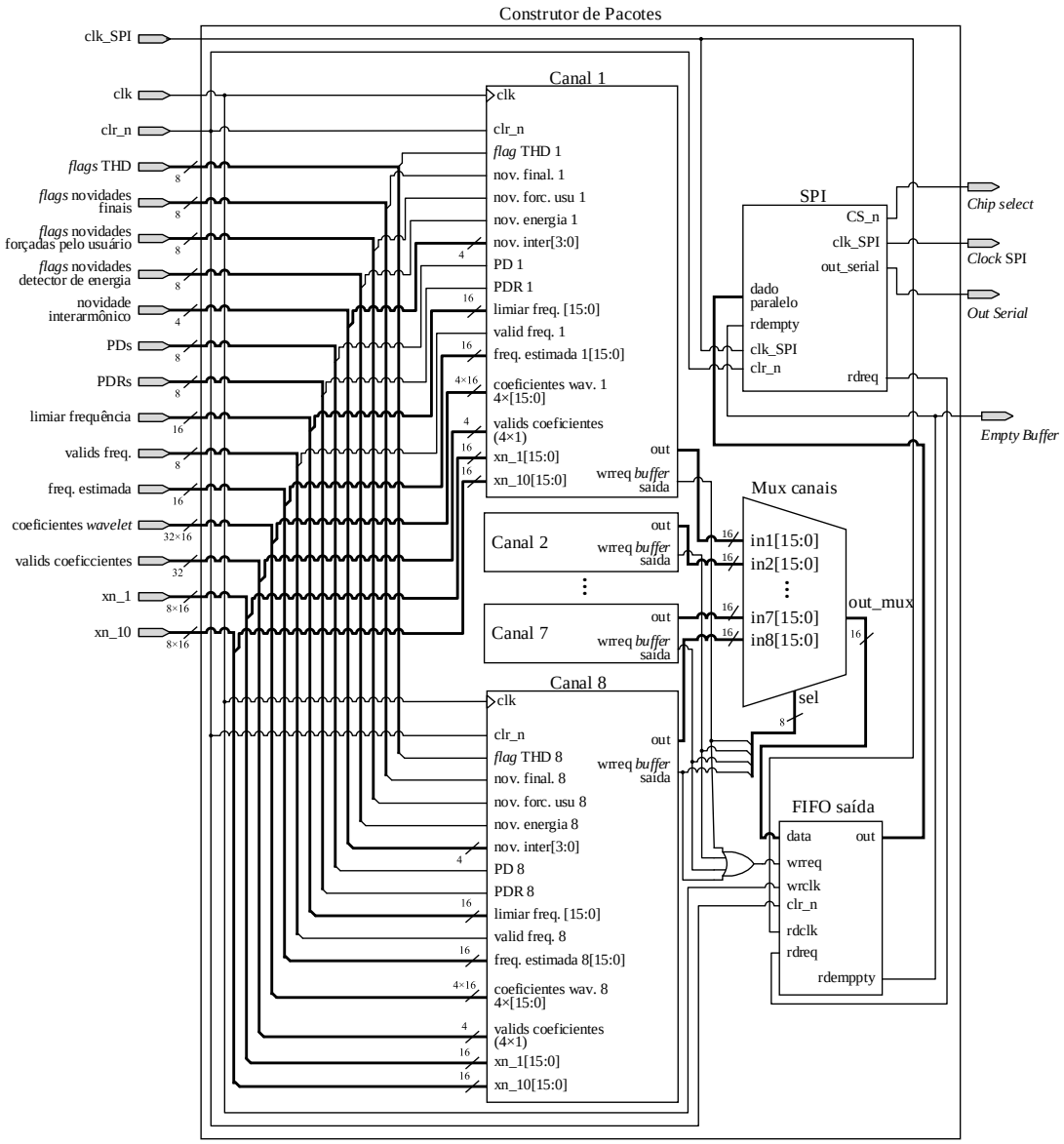


Figura 91: Representação do bloco Construtor de Pacotes.

A identificação das informações pertencentes a cada canal é realizada através da inserção de números indicadores. Esses números são: “100” para as informações relativas ao primeiro canal, “200” para as do segundo canal, e assim por diante até o último canal, cujo número indicador é “800”. Ao número indicador é de canal é somado um número de transição de novidade para se obter o “*flag* identificador de canal”, seguindo o seguinte protocolo:

- Se o *frame* atual é um *frame* de novidade e o anterior também: soma-se um número de transição com valor “10” ao número indicador de canal formando um *flag* identificador

da forma “X10”, onde X = 1, 2, ..., 8. Após este flag identificador, seguem os quatro valores de frequências do frame atual e por último os 512 coeficientes da *wavelet* formando um total de 517 elementos. Essa situação é representada pelo Caso 1 da Figura 92.

- Se o *frame* atual é não é um *frame* de novidade e o anterior também não: soma-se um número de transição com valor “20” ao número indicador de canal formando um *flag* identificador da forma “X20”, seguido apenas pelas quatro frequências do frame atual, formando um total de 5 elementos. Essa situação é representada pelo Caso 2 da Figura 92.

- Caso o *frame* atual não seja de novidade, mas o anterior seja: soma-se um número de transição com o valor “30” ao número indicador de canal formando *flag* identificador da forma “X30” seguido pelos coeficientes da *wavelet* correspondentes ao último ciclo da fundamental do frame anterior (que foi de novidade) e após estes seguem-se os quatro valores de frequência do frame atual formando um total de 134 elementos. Essa situação é representada pelo Caso 3 da Figura 92. O ciclo extra de coeficientes da *wavelet* é justificado pelo fato de que a reconstrução da *wavelet* apresenta efeitos de borda e não seria possível reconstruir o frame inteiro corretamente, caso não houvesse esse ciclo extra.

- Se o *frame* atual é um frame de novidade, mas o anterior não é: soma-se um número de transição com o valor “40” ao número indicador de canal formando um *flag* identificador da forma “X40” seguido pelos dois pontos xn_1 e xn_10 (Figura 90), aos quais seguem-se o *flag* identificador “X10” seguido dos quatro valores de frequência e por último os 512 coeficientes da *wavelet* formando um total de 520 elementos.

Sendo assim, com os *flags* identificador, todos os casos de transição entre *frames* são cobertos.

Dessa forma, o Construtor de Pacotes insere o *flag* identificador de canal e em sequência insere as informações relativas aquele canal.

Esses *flags* são de suma importância para o processo de descompressão, visto que a partir deles é possível saber a qual canal pertencem as informações, que tipo de *frame* que é tratado e ainda, prever quais informações subsequentes ao *flag* identificador.

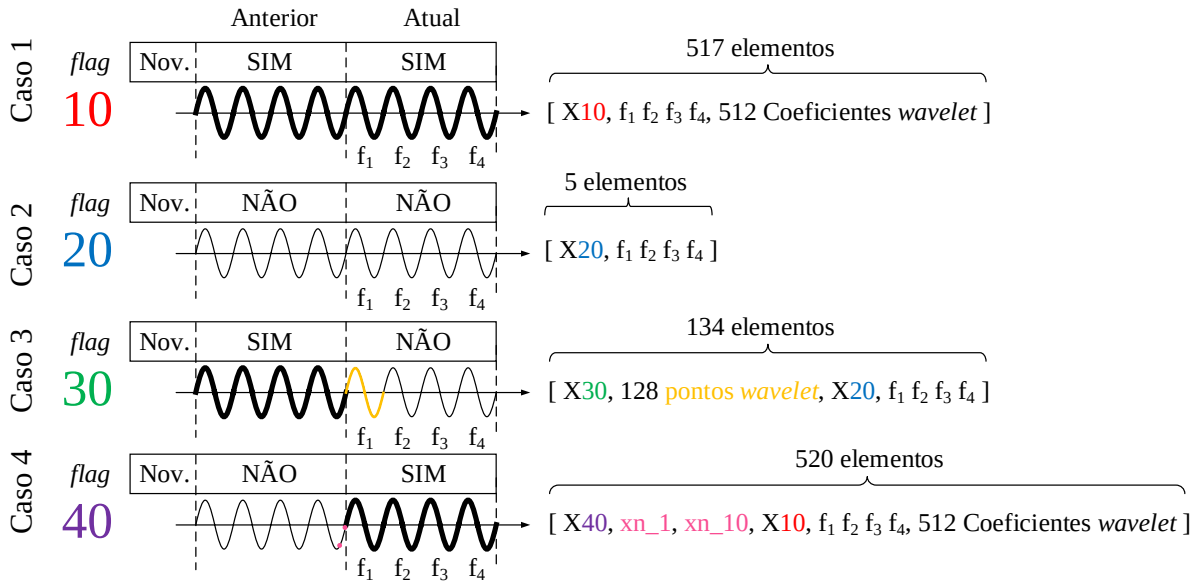


Figura 92: Protocolo de *flags* do Construtor de Pacotes.

4.3.7. Máquina para fechamento de arquivos

Quando o usuário envia um comando de fechamento de arquivo através do aplicativo Android®, o processador ARM comunica a ordem à FPGA. Porém, antes de fechar, ela força três *frames* de novidade seguidos e então prossegue com o fechamento do arquivo. Essa é a função do bloco Maq. Fecha.

Esse bloco é composto de uma máquina de estados que é ativada quando se envia o comando de fechamento (*flag* final). Nesse processo o bloco UART aciona os *flags* de novidades finais para cada canal.

Após serem descarregados os três *frames* completos o arquivo é finalmente fechado (sinal *close file*).

4.4. Protótipo desenvolvido

A FPGA utilizada para as implementações dos algoritmos é da empresa ALTERA®, e pertence à família Cyclone IV (modelo EP4CE22F17C6). Essa FPGA possui 22320 elementos lógicos, 154 pinos, 608256 bits de memória e 4 PLLs (Cyclone IV, Device Handbook, 2010).

A FPGA faz parte de um *kit* de desenvolvimento: DE0-Nano. Uma imagem desse *kit* pode ser vista na Figura 93 (ALTERA®, DE0-Nano Development and Education Board, 2015).

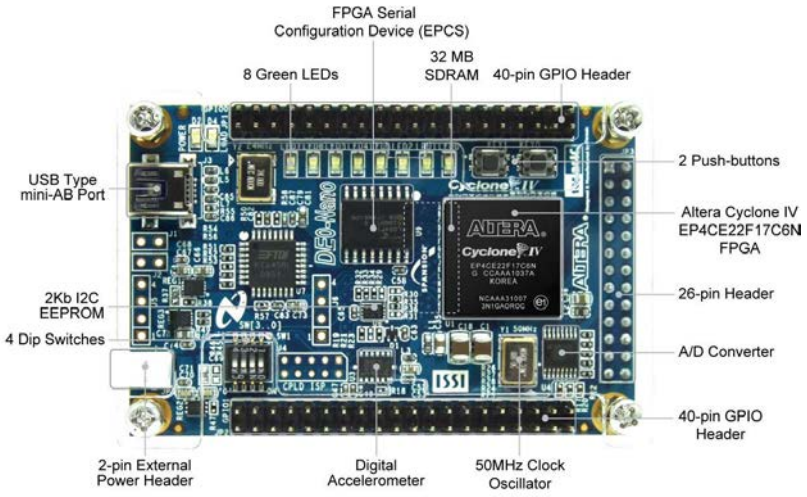
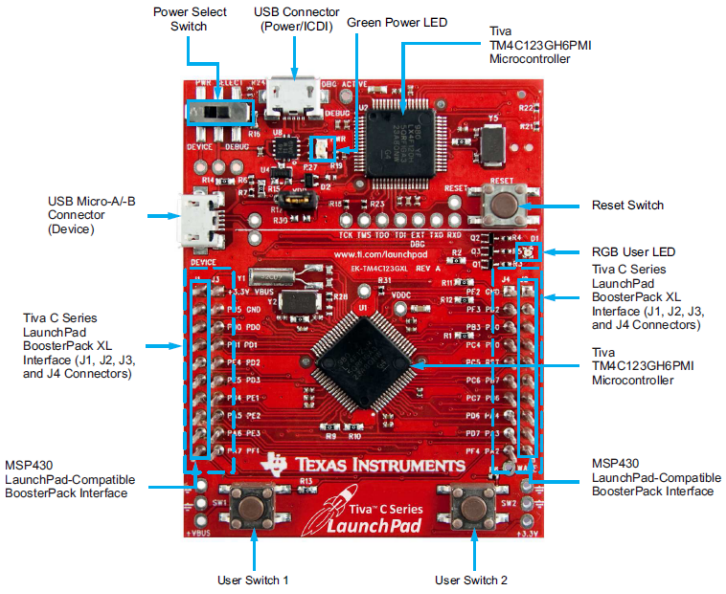


Figura 93: Imagem do *kit* de desenvolvimento DE0-Nano.

O processador ARM que foi utilizado é da família Stellaris/Tiva, da empresa Texas Instruments. Ele é um processador ARM Cortex-M4F 80MHz, de 32 bits, com Unidade de Precisão em ponto Flutuante (FPU), e interface multicanal serial/SPI (Texas Instruments, 2013). A Figura 94 mostra uma imagem do processador utilizado (Tomar, 2013).



Tiva C Series TM4C123G LaunchPad Evaluation Board

Figura 94: *Kit* com o processador ARM Cortex M4.

O gerenciamento e controle do cartão SD, é feito por um CI dedicado (CH376) (L. Nanjing QinHeng Electronics, 2013). Ele manipula todas as rotinas de escrita de arquivos nos diretórios do cartão SD. O processador ARM controla a operação deste IC, bem como o envio de dados que devem ser armazenados. Uma interface SPI é utilizada para a comunicação entre o processador ARM e o CI CH376.

O protótipo é composto por duas placas principais: uma placa de processamento e uma placa analógica de condicionamento. A Figura 95 mostra a placa de processamento. Além da FPGA e do processador ARM, outros elementos podem ser destacados, como o controlador do cartão SD, o *Bluetooth*, as interfaces USB de comunicação com o ARM e com a FPGA, as baterias e o conversor AD.

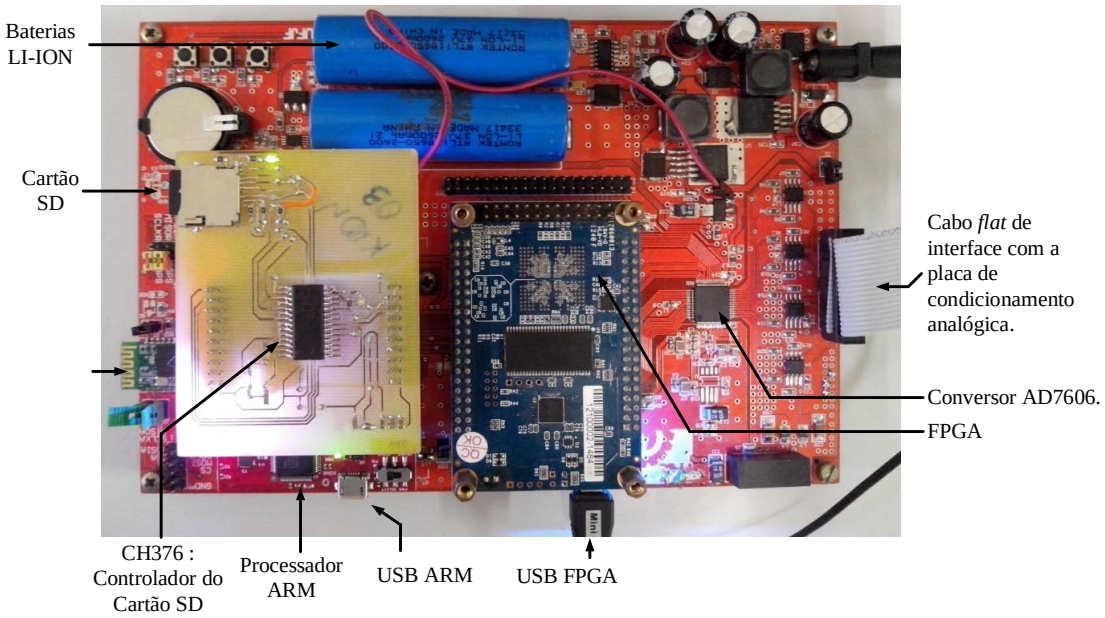


Figura 95: Foto da placa de processamento.

A Figura 96 traz uma imagem da placa de condicionamento. Ela é composta por dois principais circuitos: o circuito de tensão e o circuito de corrente. Eles são responsáveis por adequar os sinais a níveis aceitáveis para o conversor AD. Essa placa permite duas configurações de condicionamento, para tensões e correntes. No caso das tensões, pode-se escolher utilizar-se um divisor resistivo ou os TPs. A primeira opção fornece melhor acurácia na medição, enquanto a segunda, fornece isolamento galvânico. Os sinais de corrente podem ser acoplados aos TCs fixos na placa com um valor máximo de 7,5 A, ou então podem ser aplicados a um transformador externo.

Os transformadores foram construídos com lâminas de ferrosilício e permitem uma largura de banda de 20 kHz.

Todos os sinais da placa de condicionamento passam por filtros *anti-aliasing* Butterworth de quarta ordem com uma frequência de corte de 3 kHz. Esses filtros permitem controle de ganho e *off-set* para cada canal.

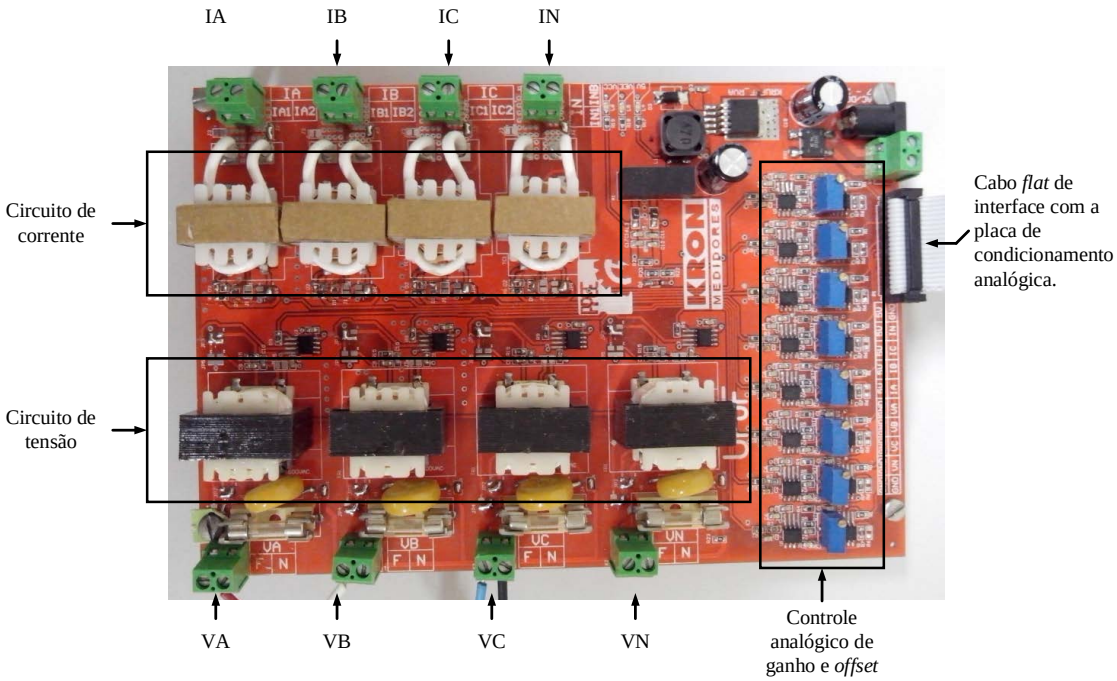
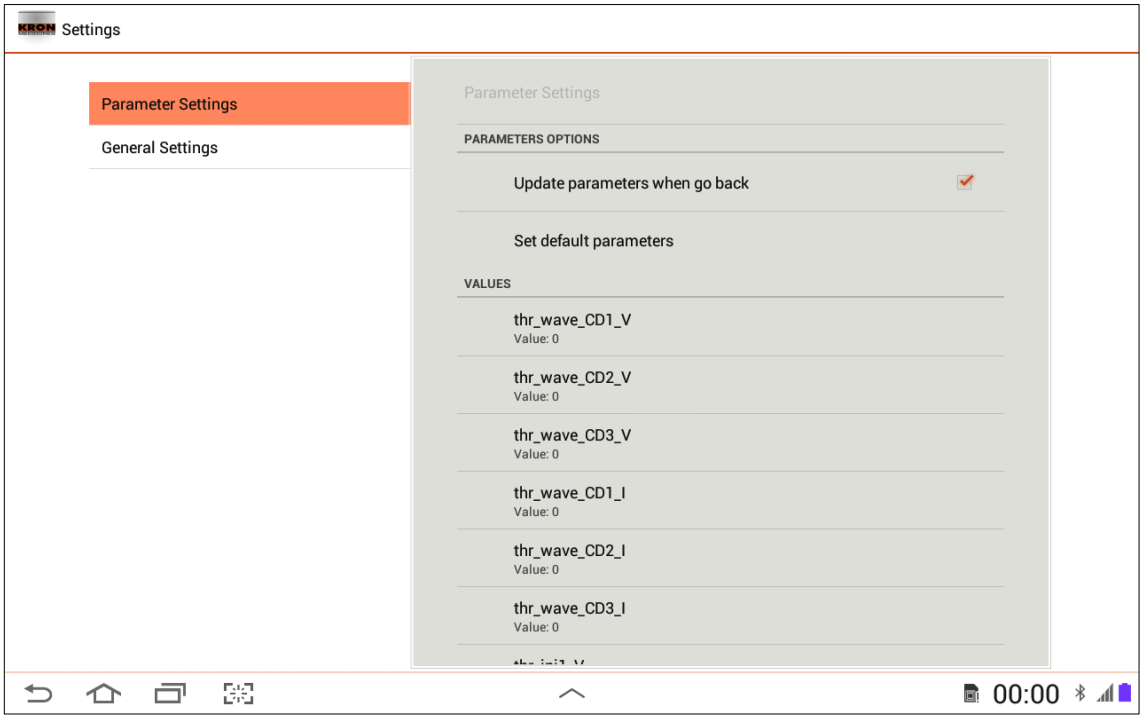


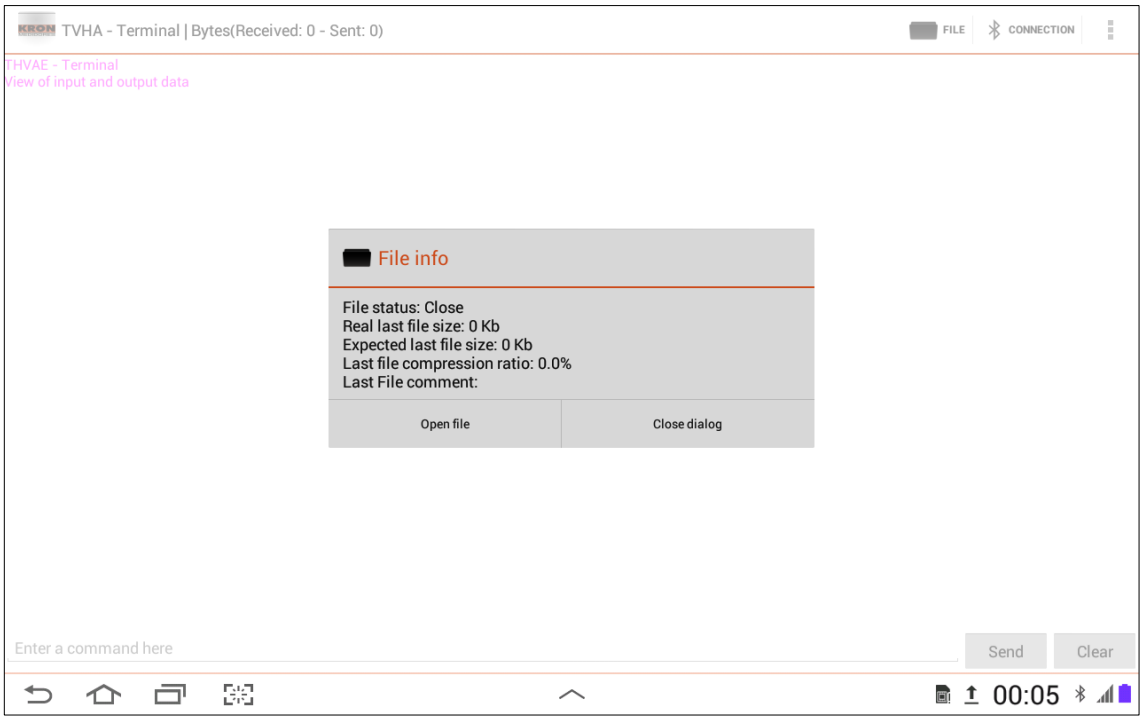
Figura 96: Foto da placa de condicionamento.

A parametrização do sistema é realizada por um aplicativo Android®. Os parâmetros configurados na última utilização do sistema, ficam armazenados na memória interna do processador ARM. Mesmo que o sistema seja desligado, os parâmetros ainda continuam armazenados. Toda vez que é ligado, ou quando é solicitada uma atualização de parâmetros, estes são transmitidos da memória interna do ARM para a FPGA, a qual possui uma memória volátil.

As Figuras 97 a) e b) mostram imagens retiradas de um *Tablet* executando o aplicativo Android®. A primeira mostra a tela de configurações e a segunda mostra uma tela de comando de abertura de arquivos.



a)



b)

Figura 97: Aplicativo de configuração do SDCDE. a) Tela de parâmetros. b) Comando de abertura de arquivo.

4.5. Conclusões do capítulo

Nesse capítulo foi abordada a implementação do Sistema de Detecção e Compressão de Distúrbios Elétricos (SDCDE) proposto no trabalho. Deu-se enfoque nas implementações em FPGA.

Foi visto que o sistema possui algumas características importantes, que valem a pena considerar:

- Armazenamento de dados: capacidade de salvar a informação num dispositivo de armazenamento portátil, como um cartão micro SD ou um *pen drive*, proporcionando segurança, fácil transporte e cópia rápida dos arquivos. A descompressão pode ser facilmente realizada num microcomputador.

- Sistema de localização: a placa é equipada com sistema GPS/RTC. Capaz de inserir informações de estampa de tempo e localização, nos arquivos de medição.

- Interface com o usuário: além da IDE, desenvolvida especificamente para a programação do processador embarcado, foi desenvolvido um aplicativo Android®, que permite a interface com o usuário, dando-lhe autonomia completa sobre o dispositivo. Esse aplicativo pode ser instalado num dispositivo móvel, o qual se comunicará à plataforma via *Bluetooth*. Através dele é possível enviar comandos de abertura e fechamento de arquivos, controlar os níveis de limiares para detecção, compressão e outros parâmetros.

Além da comunicação via Bluetooth, o processador ARM é capaz de se comunicar via USB a um computador.

5. RESULTADOS

5.1. Introdução

Este capítulo objetiva apresentar os resultados do funcionamento do SDCDE, incluindo as etapas dos processos de detecção e de compressão intermediários.

Na Seção 5.2 é apresentada duas bancadas de testes para o protótipo desenvolvido. Na Seção 5.3 é mostrado o funcionamento do bloco de controle do conversor AD. Na Seção 5.4 é mostrada a estimação da energia e o seccionamento do sinal. A Seção 5.5 mostra resultados provenientes da compressão *wavelet*. A Seção 5.6 traz resultados da estimação da frequência. Na Seção 5.7 são apresentados resultados sobre a detecção de distúrbios. Testes de compressão total e de reconstrução são apresentados na Seção 5.8. Na Seção 5.9 são apresentados resultados de estudos de casos. Por fim, na Seção 5.10 são feitas conclusões do capítulo.

5.2. Bancadas de testes

Para serem realizados testes com o protótipo, duas bancadas de teste foram montadas. A primeira, teve a finalidade de comprovar o funcionamento da placa de processamento apenas. Para isso, os sinais foram aplicados diretamente nas entradas da placa de processamento, através de um gerador de sinais. A segunda, contemplou o teste de ambas as placas do sistema, na qual foi utilizada uma fonte conectada à placa de condicionamento, e esta, à placa de processamento.

A Figura 98 mostra uma imagem da bancada de testes para a placa de processamento. Nela, além da placa de processamento, pode ser visto o gerador de sinais e um osciloscópio, para monitorar os sinais da placa.

O gerador de sinais utilizado é da empresa Agilent®, modelo 33120A. Possui largura de banda de até 15 MHz, e suporta sinais sintéticos de até 16.000 pontos para serem carregados, através de um *software* dedicado (Agilent, 2015). Uma imagem desse *software* pode ser vista na Figura 99, na qual um sinal com transitório é apresentado.

O osciloscópio, também da marca Agilent®, modelo DSO-X 2002ª, possui dois canais e largura de banda de 70 MHz.

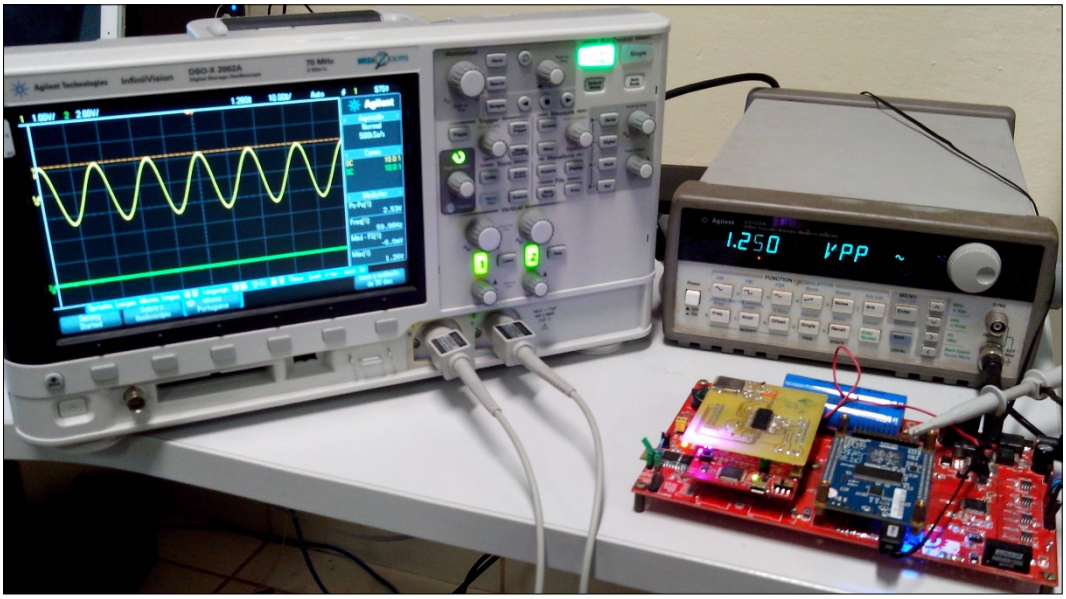


Figura 98: Bancada de testes para a placa de processamento.

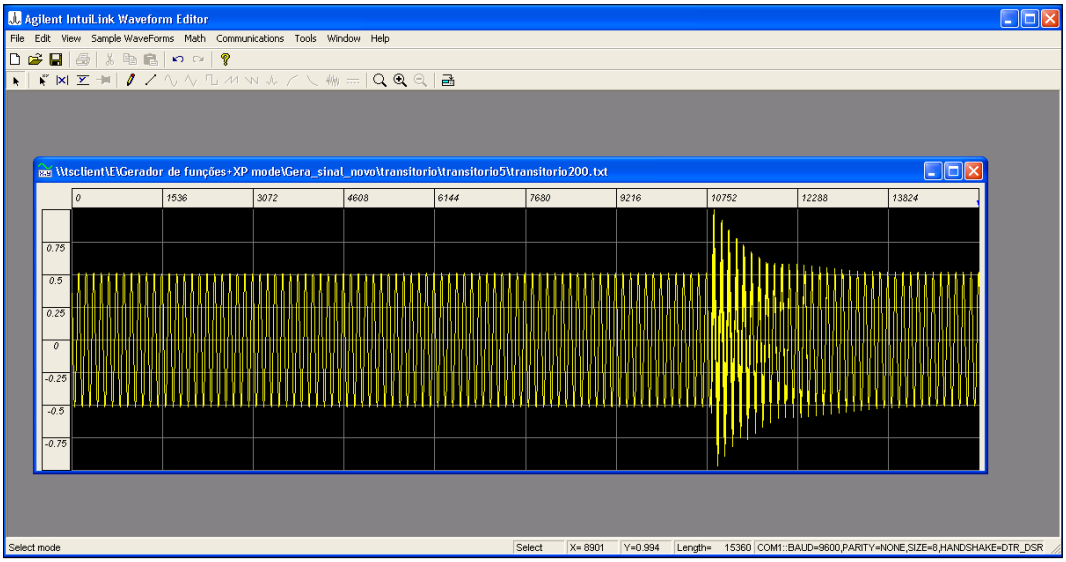


Figura 99: *Software* do gerador de sinais.

A bancada de testes do sistema completo utilizou a fonte Omicron CMC-256-6 *plus* para gerar os sinais de tensão e de corrente para os testes. Com essa fonte, é possível gerar sinais de tensão de até 500 V entre fases e corrente de até 25 A numa configuração trifásica. A Figura 100 mostra essa bancada de testes. Nela, podem ser vistos os dois módulos do sistema, juntamente com a fonte.

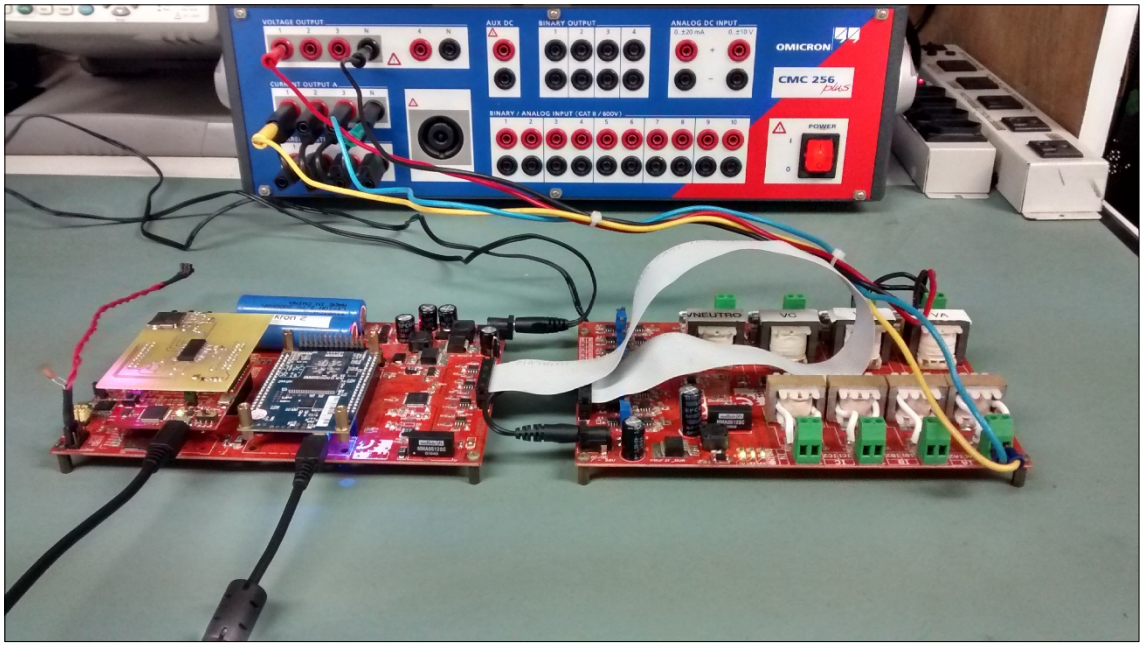


Figura 100: Bancada de testes para o sistema completo.

Para realizar a análise dos elementos dentro da FPGA, o Quartus II® possui seu próprio analisador lógico, denominado Signaltap II *Logic Analyser*®. Através dele, é possível ver o valor de qualquer bit do sistema digital sintetizado na FPGA em tempo real.

5.3. Controle do conversor AD

Para testar o funcionamento do bloco de controle e interface com o conversor AD, foram aplicadas quatro tensões no barramento de entrada da placa de processamento. Através do SignalTap II, foram escolhidas as saídas das memórias FIFOs presentes no bloco Interface AD. No funcionamento correto, essas saídas devem apresentar os valores das conversões dos sinais de entrada.

A Figura 101 mostra a tela do SignalTap II, na qual os quatro canais que estão sendo convertidos podem ter o comportamento senoidal monitorado visualmente.

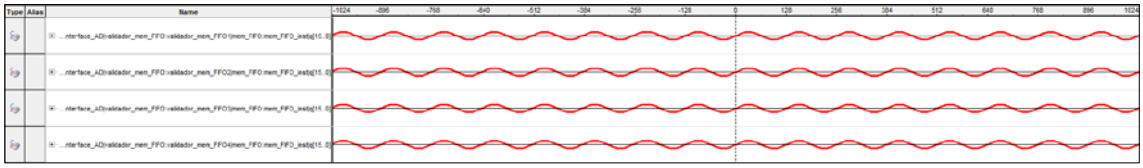


Figura 101: Funcionamento do bloco Interface AD.

Para um segundo teste, foi carregado no gerador, e aplicado ao sistema, o sinal mostrado na Figura 99. O resultado pode ser visto pela Figura 102.

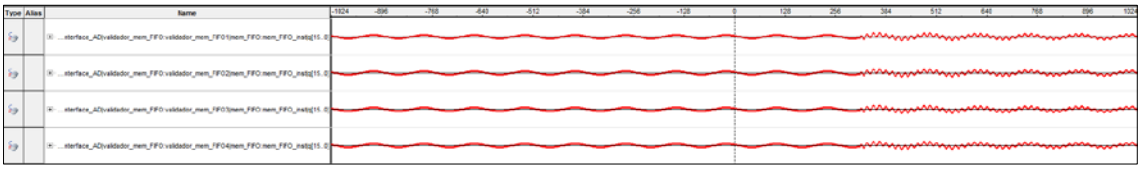


Figura 102: Aplicação de um sinal com transitório à placa de processamento.

A comparação destes sinais com os correspondentes sinais reais, escalados adequadamente, foi realizada nesta etapa, porém optou-se em não mostrar estes resultados nesta seção, deixando para apresentá-los na Seção 5.7 com todos os blocos em funcionamento.

5.4. Estimação da energia e seccionamento do sinal

Conforme visto na Subseção 4.3.5, o bloco de detecção e estimação de energia é o responsável por dividir os sinais em *frames*, computar a sua energia e tomar a decisão se o *frame* é um *frame* com novidade ou não. A Figura 103 mostra a tela do SignalTap II com um sinal de tensão senoidal convertido, e abaixo dele a estimação da energia de cada *frame* em tempo real.

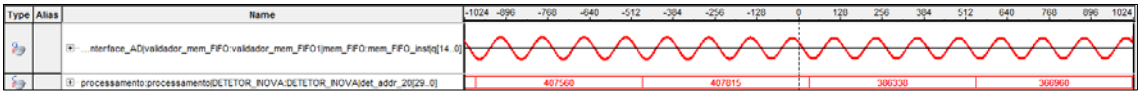


Figura 103: Seccionamento e estimação da energia do *frame*.

5.5. Decomposição e compressão *wavelet*

Para a comprovação do funcionamento do bloco de decomposição *wavelet*, primeiramente fez-se uma simulação utilizando o *software* Model*Sim*®, da empresa Mentor Graphics®. Foi utilizado um sinal sintético contendo um transitório para melhor visualização. O resultado dessa simulação, para um canal, pode ser visto pela Figura 104.

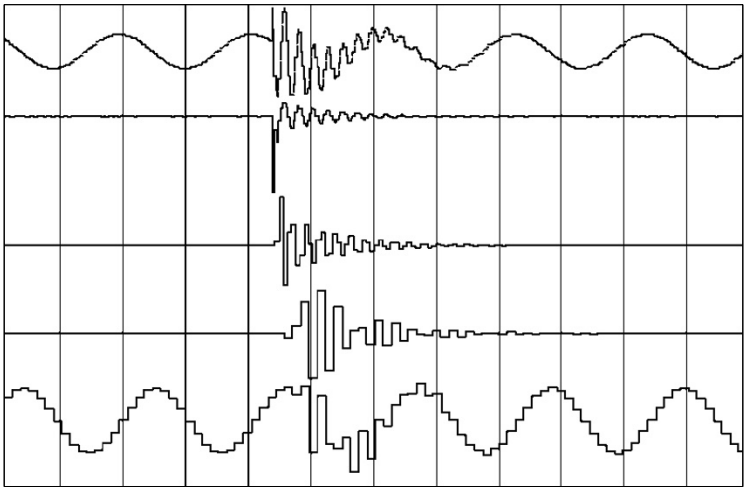


Figura 104: Simulação funcional do bloco de decomposição *wavelet* utilizando o processador embarcado.

Analisando a Figura 104, de cima para baixo, são mostrados o canal de entrada, em seguida o coeficiente de detalhe C_{D1}, abaixo deste, C_{D2}, depois C_{D3} e por fim C_{A3}. É importante notar o momento do distúrbio transitório, quando as altas frequências também oscilam. É notável também a redução de resolução temporal dos coeficientes de mais alta ordem, devido à presença dos decimadores na estrutura de análise da *wavelet* (Figura 8), dando um aspecto mais quadriculado à onda.

Após realizar a simulação a FPGA foi gravada. O mesmo sinal com transitório foi aplicado à placa. O resultado da decomposição pode ser visto pela Figura 105.

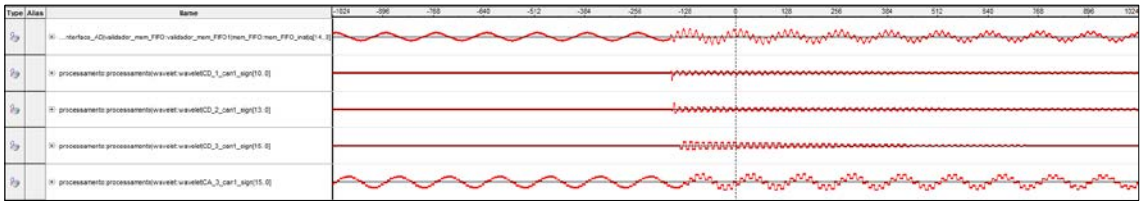


Figura 105: Decomposição *wavelet* em tempo real.

Após o teste com o transitório, um sinal real contendo um distúrbio do tipo interrupção foi aplicado ao sistema completo. Após a decomposição *wavelet*, foram aplicados limiares aos coeficientes de detalhe da *wavelet*. As Figuras 106, 107 e 108 mostram os coeficientes C_{D1}, C_{D2} e C_{D3}, antes e depois da aplicação desses limiares, representados pelas linhas horizontais tracejadas. É importante notar que o corte insere zeros nos coeficientes, tornando-os coeficientes esparsos.

A partir dos coeficientes esparsos e dos intactos foram feitas duas reconstruções a fim de realizar uma comparação com o sinal original. A Figura 109 apresenta as duas reconstruções, juntamente com o sinal original em uma região onde começa o distúrbio de interrupção, na qual é mais distinguível a diferença entre os sinais.

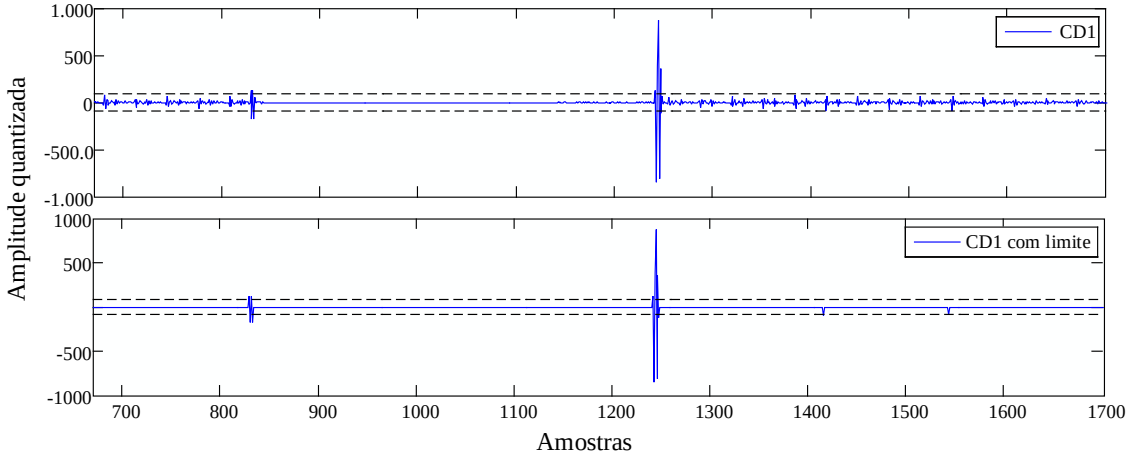


Figura 106: Coeficiente de detalhe C_{D1}.

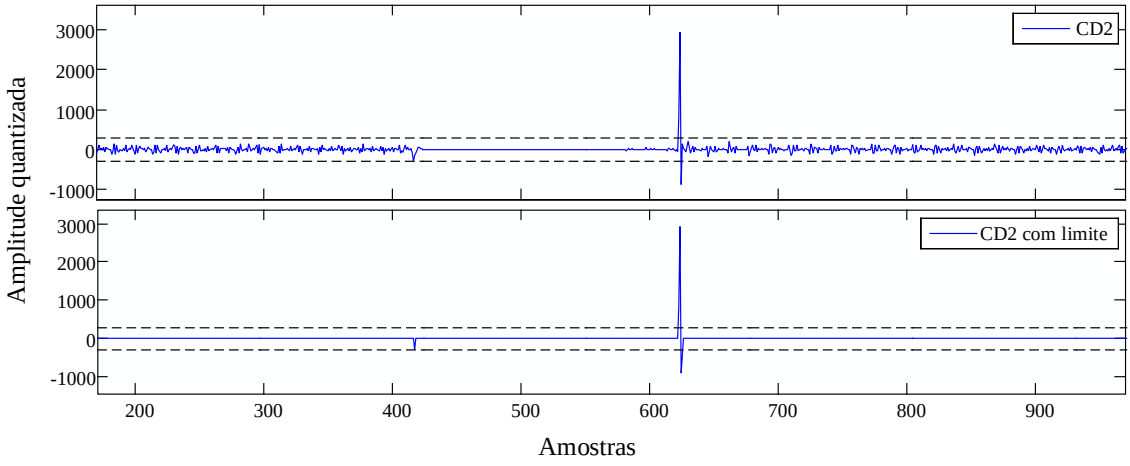


Figura 107: Coeficiente C_{D2}.

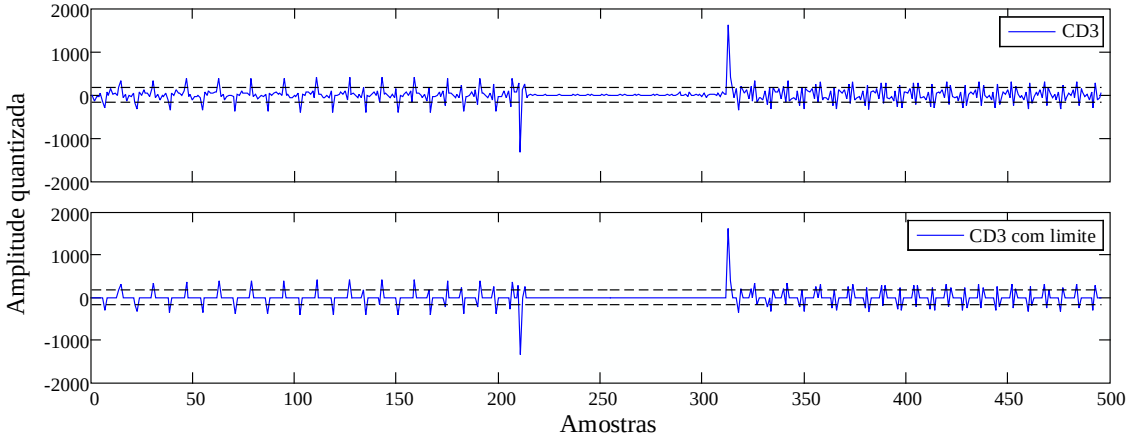


Figura 108: Coeficiente C_{D3}.

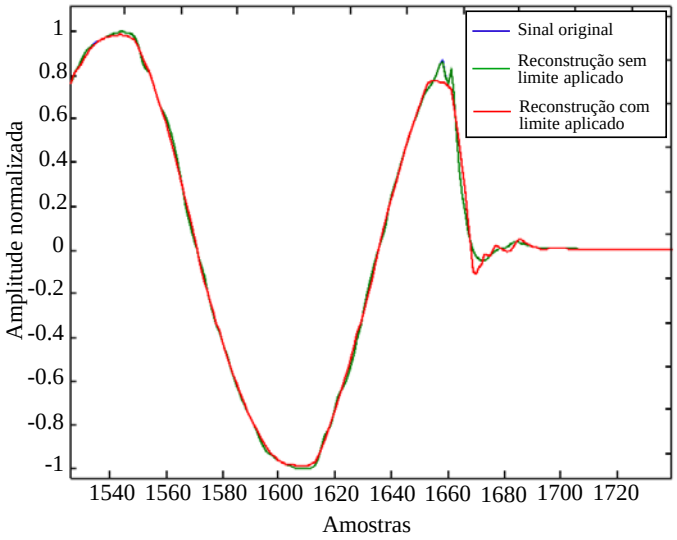


Figura 109: Comparação entre a reconstrução com coeficientes esparsos, intactos e sinal original.

É notável que a diferença entre os sinais original e o reconstruído com aplicação de limiar só se evidencia no instante de maior turbulência. Em contrapartida, a comparação entre os sinais original e o reconstruído com coeficientes intactos, em momento algum demostra uma diferença substancial, podendo ser até mesmo desprezada.

Após a aplicação dos limiares aos coeficientes, alguns parâmetros podem ser analisados a fim de se avaliar o desempenho da compactação:

- A quantidade de zeros inseridos após o corte: quanto mais zeros inseridos maior é a compactação, porém maior o erro na reconstrução.

- A energia mantida depois da aplicação dos limiares: quanto mais próxima da energia do sinal original, menos impacto a reconstrução sofre e menos distorção terá no sinal reconstruído.

- Erro médio quadrático entre a reconstrução com inserção de zeros e o sinal original: quanto menor for o erro, menor a perda de informação.

A energia *E* e o erro médio quadrático *EMQ* podem ser calculados de acordo com as Equações (5.1) e (5.2), respectivamente, onde *N* é o tamanho do sinal em questão, $x_{\text{orig}}[n]$ são as amostras do sinal original e $x_{\text{rec}}[n]$ são as amostras do sinal reconstruído.

$$E = \sum_{n=0}^{N-1} x[n]^2 \tag{5.1}$$

$$EMQ = \frac{\sum_{n=0}^{N-1} (x_{orig}[n] - x_{rec}[n])^2}{N} \tag{5.2}$$

Para o teste com o sinal real do exemplo anterior, houve uma taxa de 87,42% das amostras substituídas por zero. A energia mantida foi de 99,9352% da energia do sinal original. O erro médio quadrático teve o valor de 0,00024039.

5.6. Estimação da frequência

Para realizar testes que comprovassem o funcionamento do estimador de frequências, aplicou-se um sinal com frequência fundamental de 60 Hz. O valor da saída do processador Estimador de Frequência foi colocado no SignalTap II, juntamente com o sinal original. A Figura 110 mostra o resultado.

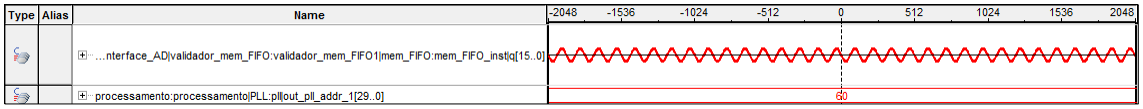


Figura 110: Estimador de frequência com sinal limpo.

Em seguida aplicou-se um outro sinal com 1% de THD e 40 dB de SNR. A frequência fundamental permaneceu em 60 Hz, como mostra a Figura 111.

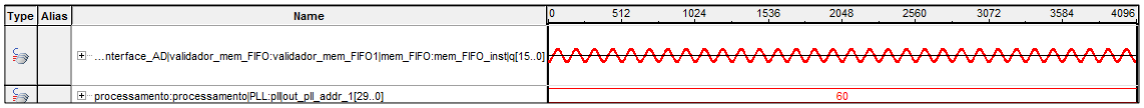


Figura 111: Estimação da frequência com THD de 1% e 40 dB de SNR.

5.7. Detecção de distúrbios elétricos

O sinal contendo o transitório da Figura 99 foi aplicado ao sistema. Em um dos canais do osciloscópio foi plugado o sinal e no outro o *flag* de novidade, a fim de ver se o mesmo indicava o evento. A detecção pode ser vista pela Figura 112, que mostra a tela do osciloscópio.

Outros testes de detecção foram feitos, aplicando-se sinais com eventos de entrada e saída de harmônicos. Os testes revelaram que o sistema detecta eventos com THD de 1%. A Figura 113 mostra um desses casos. A Figura 114 mostra uma ampliação em uma das detecções da Figura 113. Pode-se perceber que a distorção inserida no sinal com a entrada do evento é imperceptível a olho nu. A detecção de pequenos THDs só é possível devido ao bloco estimador do conteúdo harmônico, com o filtro *Notch*.

A Figura 115 mostra uma detecção de um evento do tipo *sag*. É possível observar o *flag* indicando a entrada e a saída do evento.

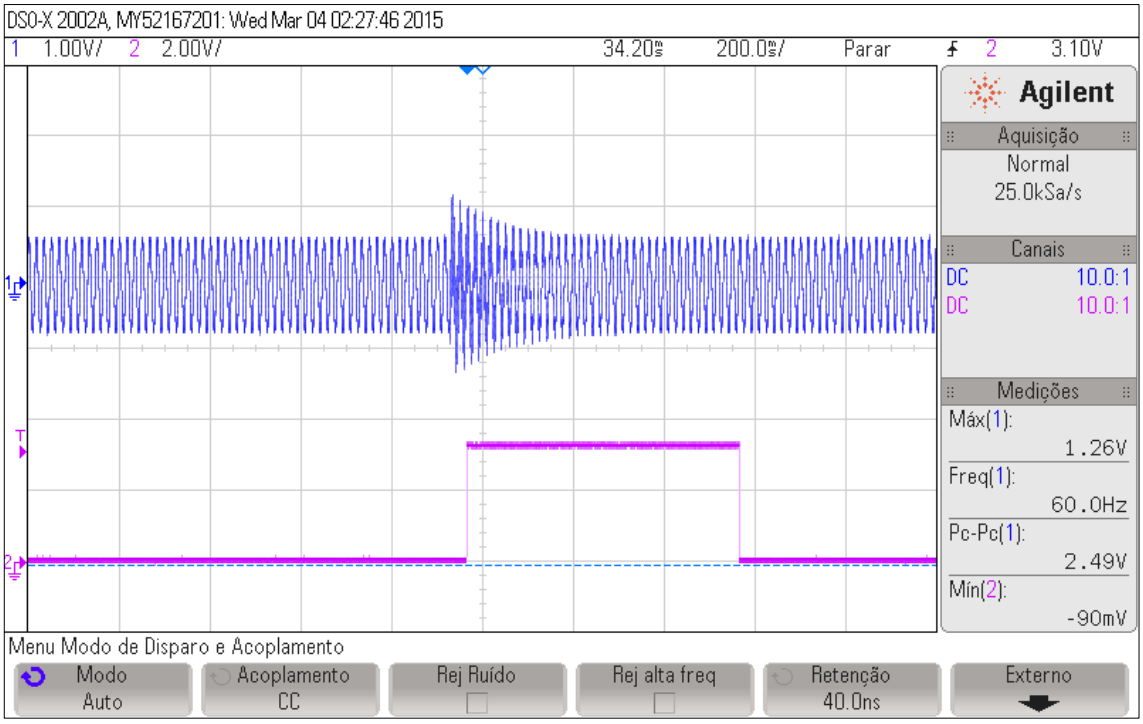


Figura 112: Detecção de transitório.

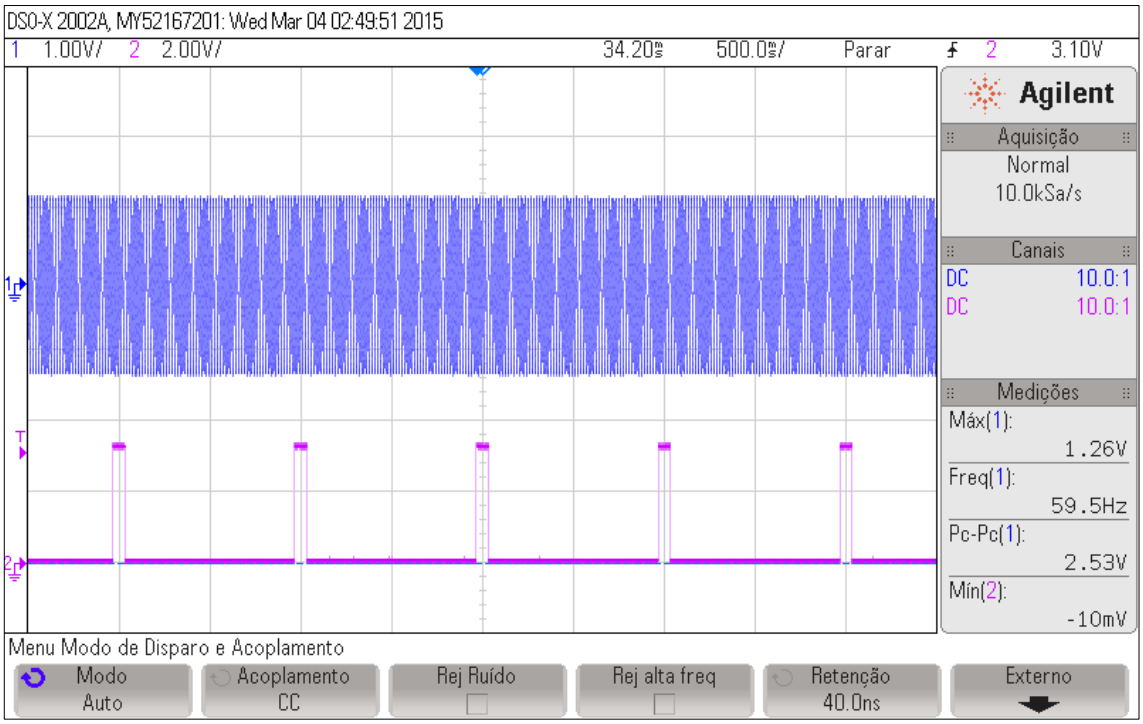


Figura 113: Detecção de entrada e saída de harmônicos com THD de 1%.

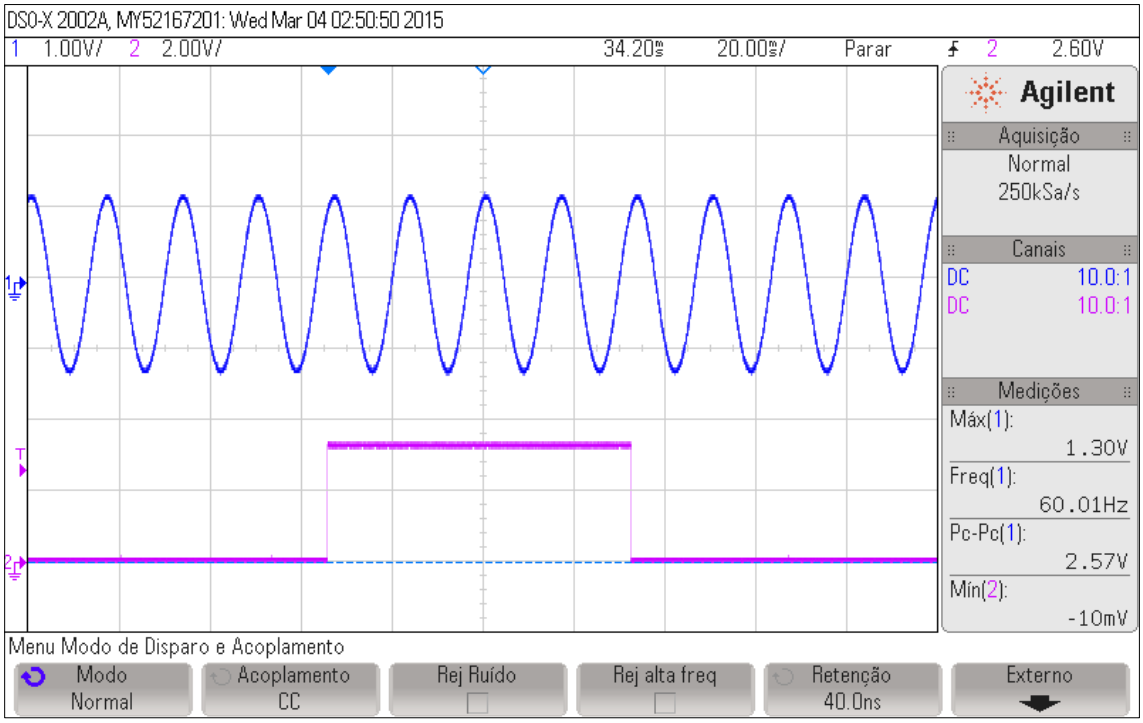


Figura 114: Ampliação de uma das detecções da Figura 113.

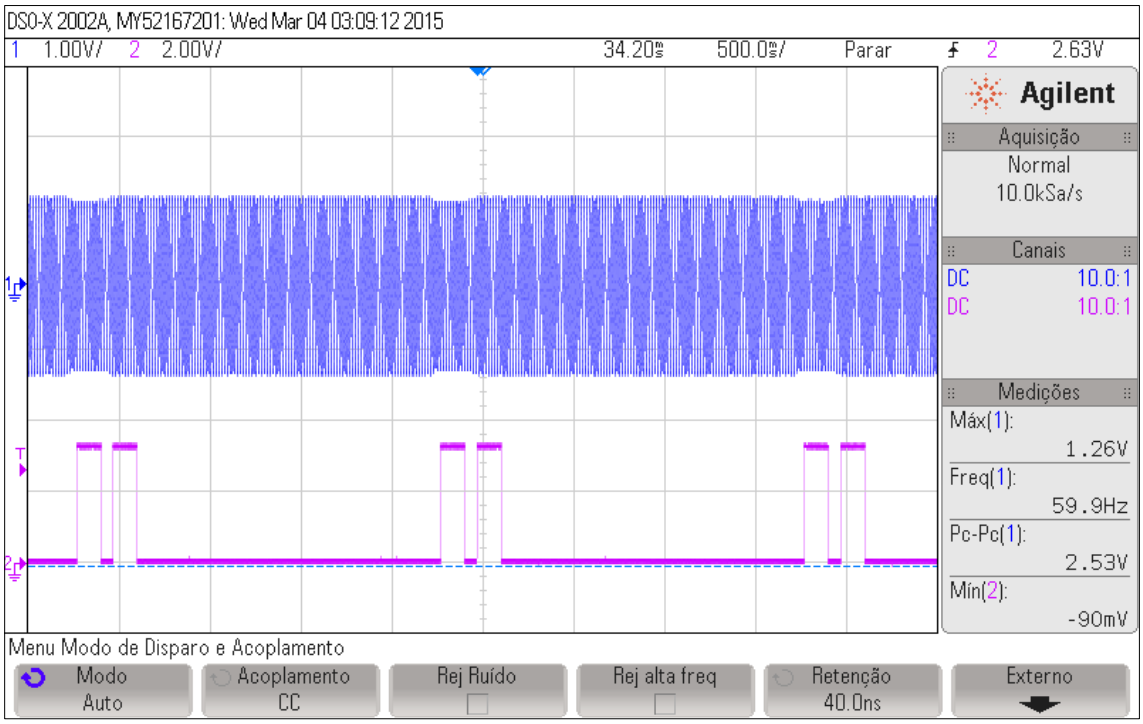


Figura 115: Detecção de *sag*.

5.8. Compressão total

Nesta seção serão mostradas representações de testes de compressão e reconstrução. Nas figuras, o esquema adotado é sempre o mesmo: uma linha pontilhada representa o sinal original, uma linha contínua mais espessa representa os *frames* de novidade, e uma linha mais fina representa os *frames* sem novidade. As hastes verticais são os divisores de *frames*.

Uma vez que os dados foram comprimidos e enviados, o algoritmo mostrado na Figura 116 é usado para o processo de descompressão.

Um sinal com variação da amplitude em rampa foi aplicado ao sistema. A frequência fundamental foi de 60 Hz. A magnitude do sinal variou de 180 V a 70 V em aproximadamente 10 ciclos, se manteve constante em 70 V por 20 ciclos e então retornou a 180 V. Esse comportamento foi periódico a cada 2,5 s. Pela observação da Figura 117, pode ser notado que os *frames* de novidade só são detectados enquanto a amplitude varia. Neste teste, o sinal sem nenhum tipo de compressão teria 423 kbytes, com a metodologia proposta, foi registrado com apenas 42 kbytes. Assim, obteve-se uma taxa aproximada de 10:1. É difícil identificar a linha pontilhada, pois a mesma está sobreposta à linha do sinal reconstruído. A ampliação mostra o encontro de um *frame* sem novidade com um no qual há novidade.

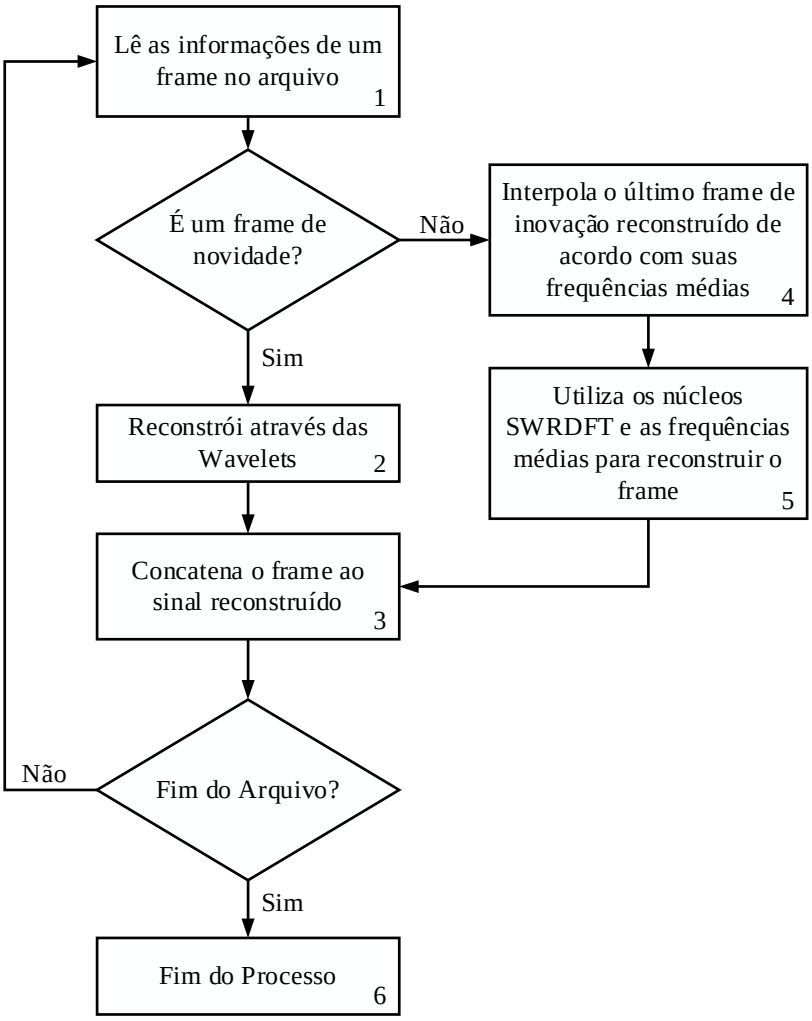


Figura 116: Diagrama de descompressão.

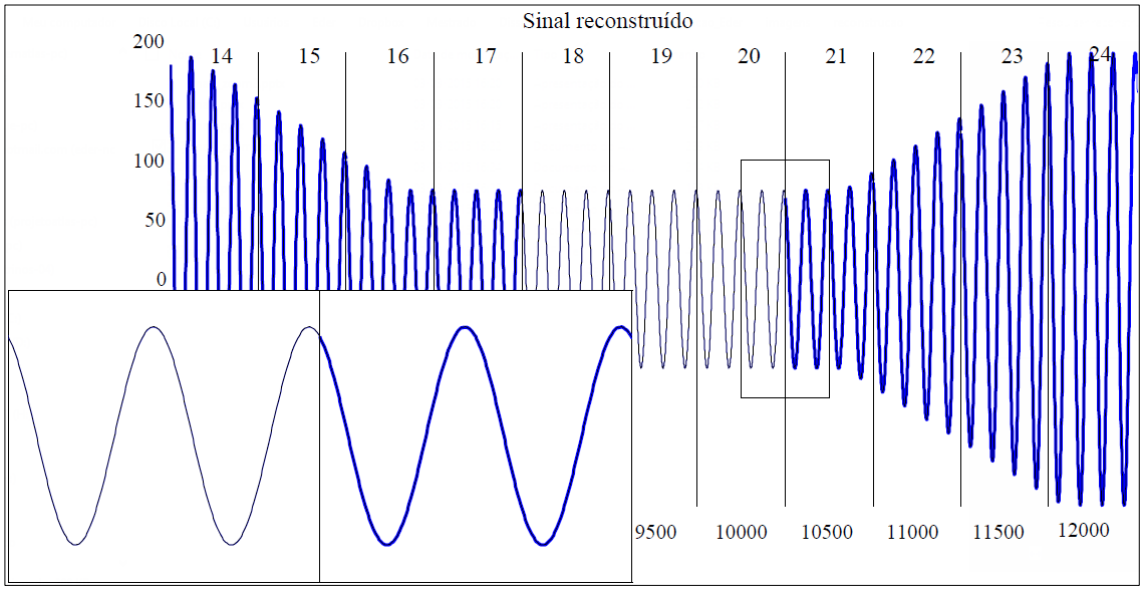


Figura 117: Variação da amplitude em rampa.

Um outro teste com um distúrbio do tipo interrupção foi aplicado ao sistema. A duração do evento foi de 5 ciclos, e se repetiu a cada 1,2 s, conforme mostra a Figura 118. Novidades foram detectadas em apenas alguns *frames*. O sinal comprimido ocupou 85 kbytes de espaço, enquanto o sinal original ocuparia 1.000 kbytes. Assim, a taxa aproximada é de 12:1.

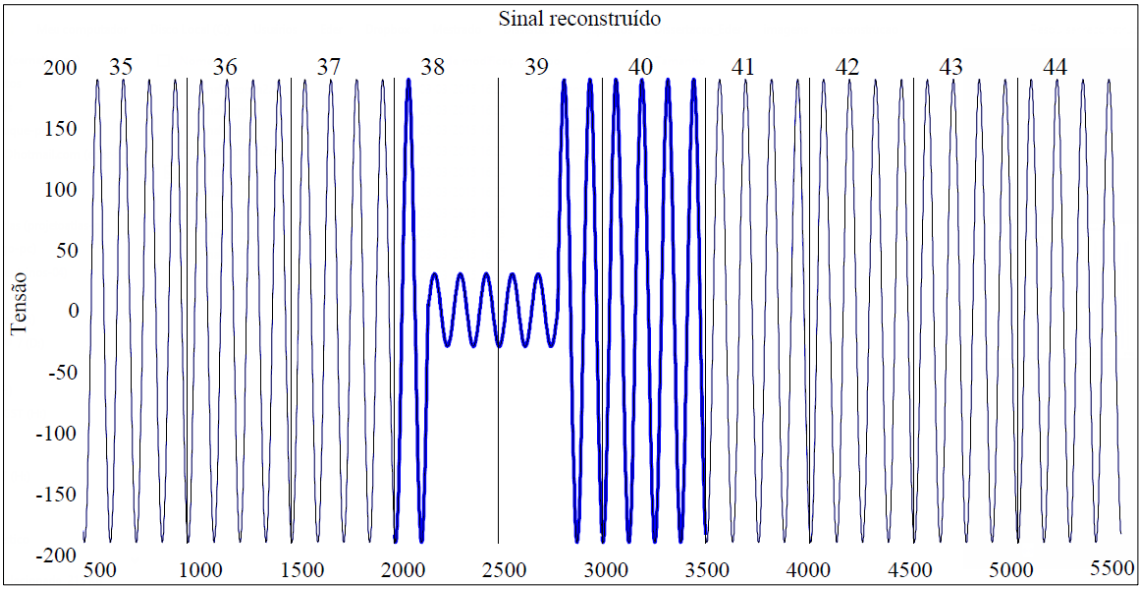


Figura 118: Interrupção.

Uma das vantagens da metodologia proposta é que, se o sinal apresenta apenas variações leves na frequência fundamental, nenhuma novidade, além das iniciais, é detectada, e o sinal pode ser reconstruído com base no *frame* de referência e nas informações de frequências médias estimadas durante a aquisição. A Figura 119 mostra uma situação como essa. A frequência fundamental foi iniciada com 60 Hz e sofreu uma variação ascendente em rampa, que demorou 3 s para chegar em 62 Hz, frequência na qual permaneceu por mais 3 s e então retornou a 60 Hz com uma rampa decrescente que também durou 3 s. Após ter voltado, ela permaneceu em 60 Hz por mais 3 s e repetiu o ciclo novamente. Mil *frames* do sinal, foram armazenados, mas destes, apenas alguns eram de novidade: 6 no começo e um no final. O sinal comprimido ocupou 10 kbytes, ao passo que, se fosse armazenado todo o sinal sem compressão, seriam utilizados 1.000 kbytes. Dessa forma atingiu-se uma taxa de compressão de 100:1.

A metodologia proposta também mostra um bom resultado mesmo em cenários de alta distorção harmônica. A Figura 120 apresenta um caso em que o sinal contém o 3º, o 5º e o 7º harmônicos em, 30%, 20% e 10%, respectivamente, adicionados à fundamental com amplitude de 180 V. Pode ser notado pela figura, que as novidades foram detectadas apenas no início e no final do

sinal. Mesmo assim o sinal reconstruído correspondeu perfeitamente com o original. Nesse exemplo, a taxa de compressão se aproximou de 100:1.

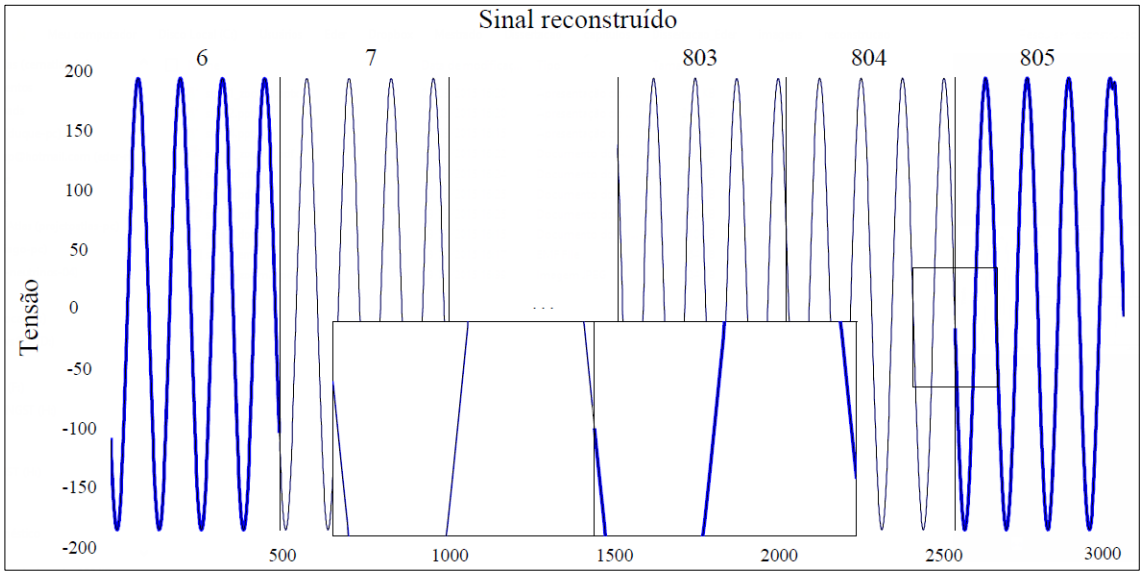


Figura 119: Variação de frequência.

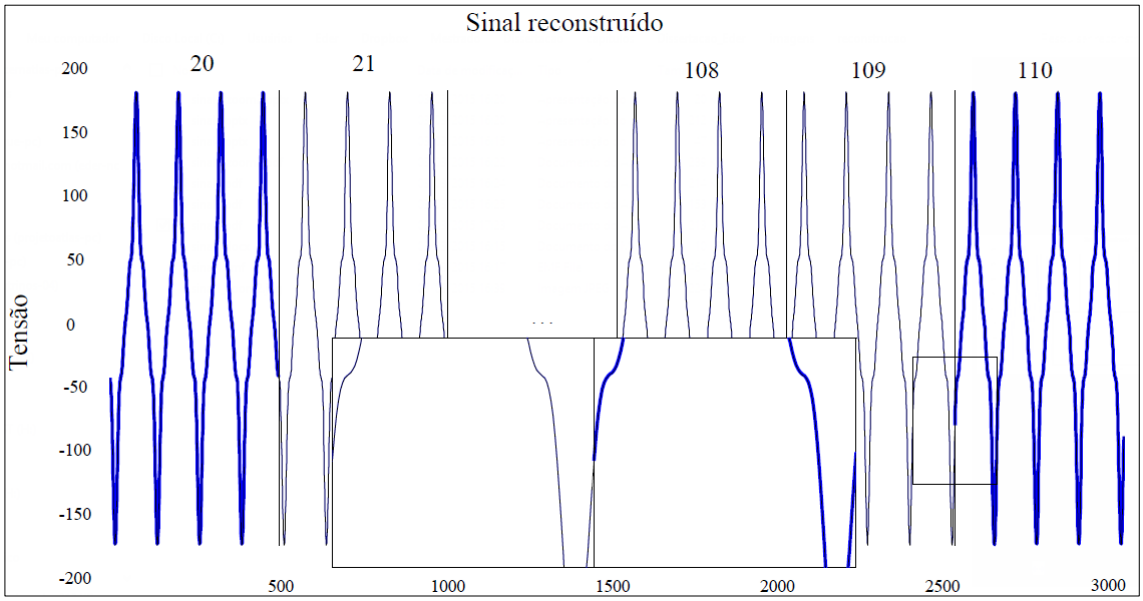


Figura 120: Distorção harmônica.

5.9. Estudos de casos

5.9.1. Partida de motor

O protótipo desenvolvido foi levado a uma pedreira localizada na cidade de Conselheiro Lafaiete, em Minas Gerais, na qual foi acoplado ao sistema de alimentação de um motor trifásico, 380 V, 100 kVA, britador do tipo mandíbula, com capacidade para até 50 t. Algumas aquisições foram feitas. Dentre elas, uma está apresentada na Figura 121. O arquivo gravado possui 2918 *frames*. A parte reconstruída mostrada na figura abrange do 1º *frame* até o 150º, para uma melhor visualização. Para este teste foi colocado um canal estando em constante detecção de novidade através do parâmetro visto na Figura 84, que é o Registrador 33 (`flag_forc_usu`), no qual foi armazenado o valor da palavra binária [01000000], para se obterem novidades constantes no canal 2. A Figura 122 mostra essa configuração no dispositivo móvel Android®.

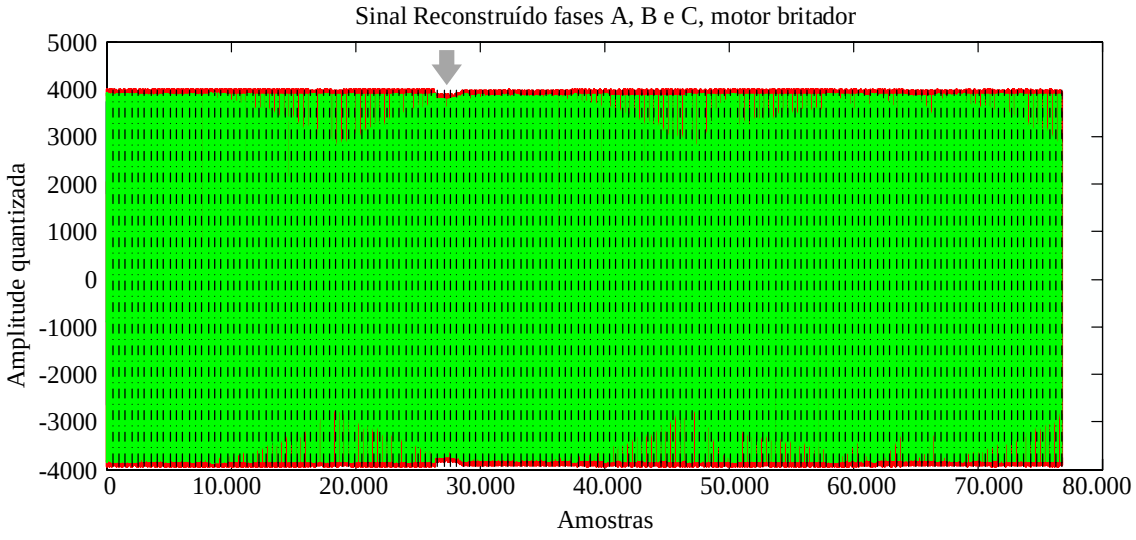


Figura 121: Sinal reconstruído das tensões de um motor britador.

Pode-se notar que, um pouco antes da amostra 30.000, existe um distúrbio do tipo *sag* devido à partida do motor (região indicada com a seta). Em sua operação normal ele é partido de forma direta. Através de uma ampliação da parte do distúrbio mostrada na Figura 123, pode-se ver que o evento ocorreu entre os *frames* 52 e 57. É notável também, que as novidades foram detectadas nas mudanças da amplitude, enquanto nas outras áreas não houveram detecções.

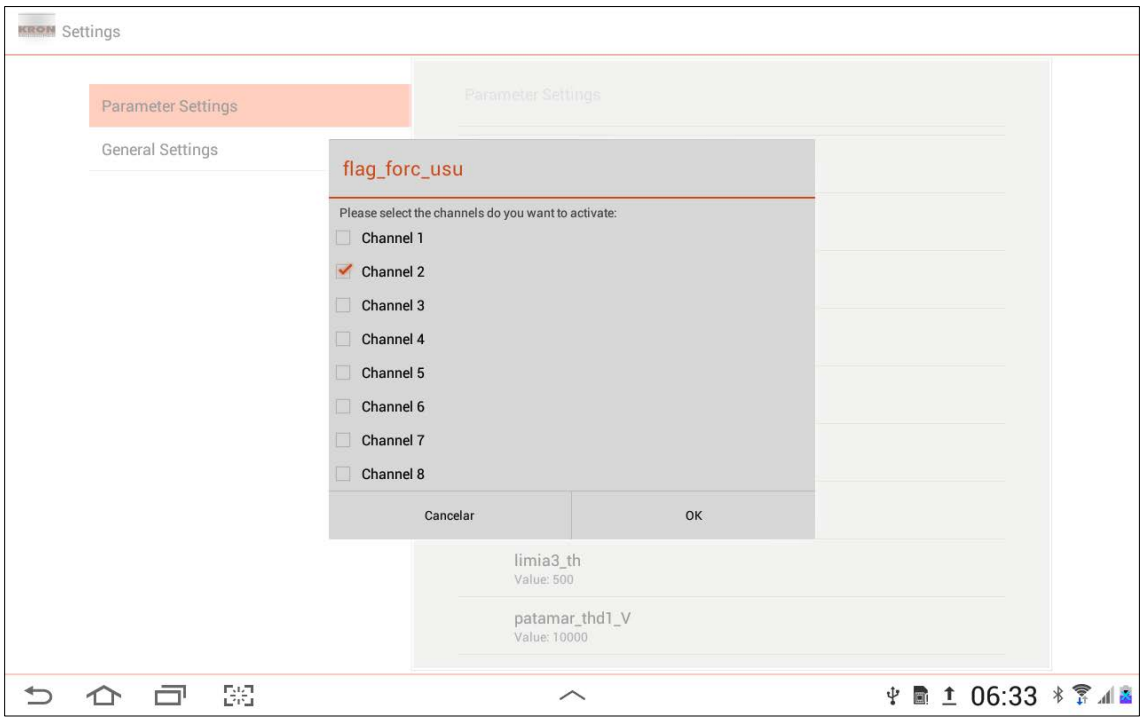


Figura 122: Configuração no dispositivo móvel para detecções constantes de novidade para o canal 2.

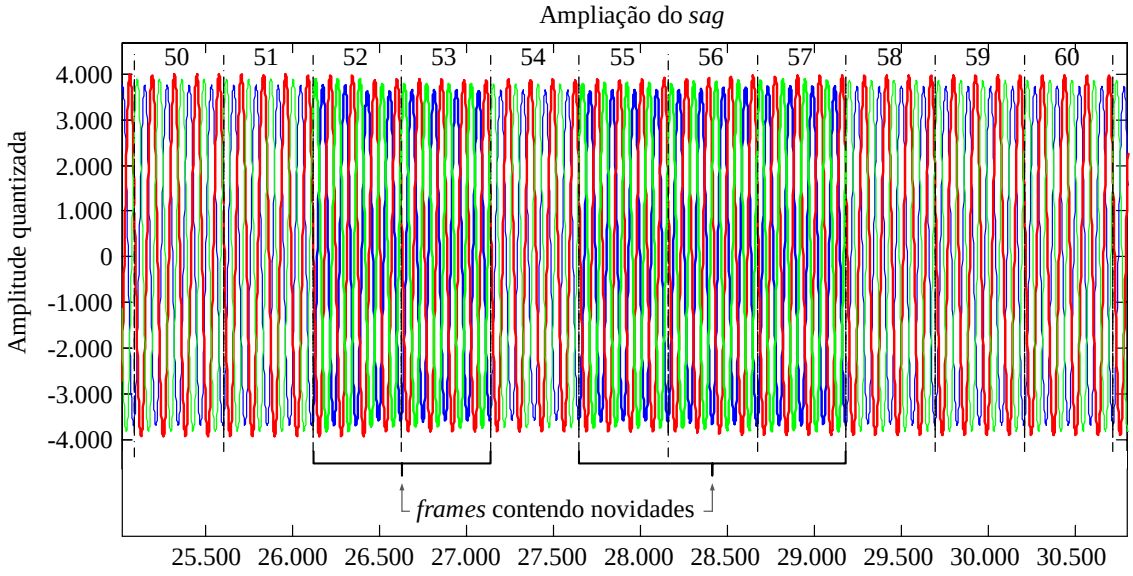


Figura 123: Ampliação do distúrbio da Figura 121.

O arquivo armazenado por esse registro possui 4.256.768 bytes (aproximadamente 4 MB). Como ele representa 2918 *frames* com três canais, pode-se concluir facilmente que um sinal amostrado a uma taxa de 7680 Hz (128 pontos por ciclo da fundamental) e com resolução de 16 bits, armazenado completamente, deveria ocupar 8.964.096 bytes (Próximo a 9 MB). Porém, deve-se lembrar que um dos canais foi detectado constantemente, por isso resultou-se em uma taxa de compressão próxima de 2:1.

5.9.2. Quadro de distribuição de circuitos

Um outro teste foi realizado dentro das dependências da Universidade Federal de Juiz de Fora. O protótipo foi acoplado ao quadro de distribuição de circuitos do Laboratório de Processamento de Sinais e Telecomunicações da Faculdade de Engenharia da UFJF, como mostra a Figura 124. Foram medidas as tensões nos terminais de entrada de um disjuntor trifásico com tensão de 127 V da fase ao neutro e 220 V de uma fase à outra. Os dois objetivos dessas medidas foram: verificar a sensibilidade da detecção e como se comporta a detecção de interrupção.

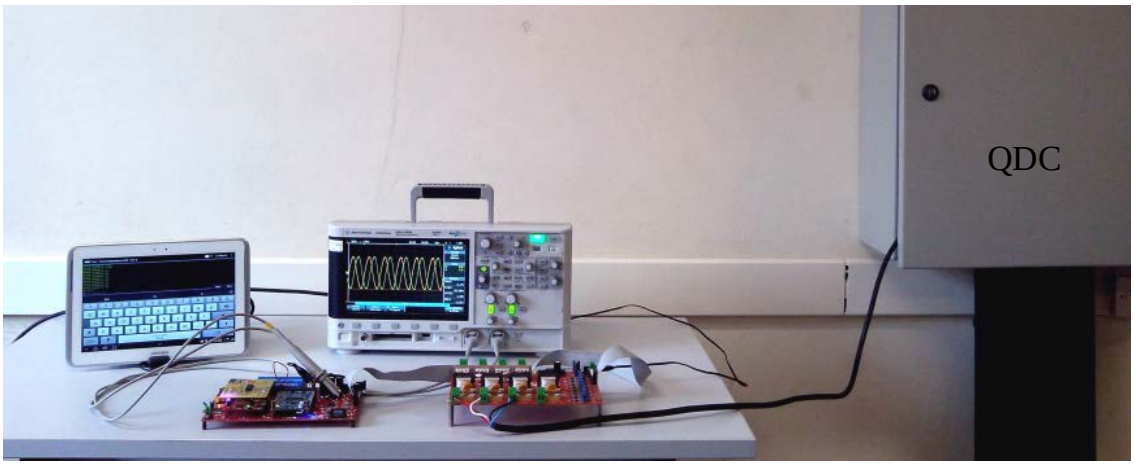


Figura 124: Estação de medição e testes do protótipo.

Primeiramente, foi realizada uma verificação da conversão das três fases em tempo real, juntamente com a estimação da frequência fundamental através do SignalTap II (Figura 125). O valor da frequência mostrado na tela do é o valor real em (Hz) multiplicado por $2\pi \times 50$, que é necessária para o funcionamento do processador. Portanto, para uma frequência de 60 Hz, deveria ser mostrado um valor próximo a 18849.

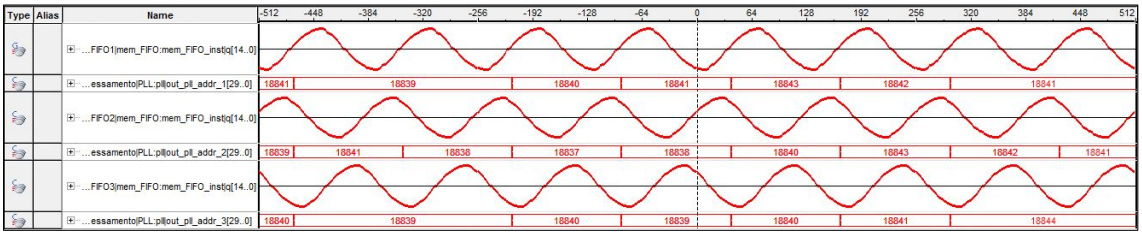


Figura 125: Verificação pelo SignalTap II.

- **Sensibilidade da detecção**

Após essa verificação registrou-se o sinal sem nenhuma detecção forçada pelo usuário através do dispositivo móvel, utilizando-se limiares de detecção mais sensíveis. Como a forma de onda da rede elétrica possuía uma distorção considerável, houve algumas detecções, devido ao bloco estimador de THD. Isso pode ser visto pela Figura 126, onde as partes mais escuras são as detecções. Nela, são mostrados 108 *frames*. O tamanho do arquivo é de 46.080 bytes. Considerando-se o armazenamento sem compressão, seria ocupado 331776 bytes. Tem-se agora uma taxa próxima a 7:1.

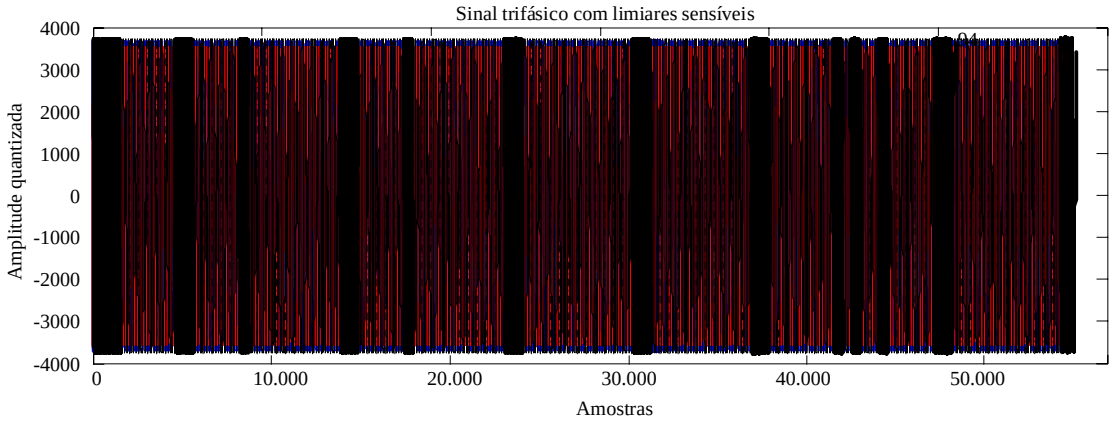


Figura 126: Detecções com limiares baixos.

Um outro registro pode ser visto pela Figura 127, agora, com limiares menos sensíveis. O tempo de duração foi pouco mais de um minuto, resultando em 1010 *frames*. Deu-se importância maior às extremidades do sinal onde houveram as novidades iniciais e finais, as quais foram ampliadas para se obter uma melhor visualização. Nota-se também que entre as mesmas não houve nenhuma detecção. Assim, a maior parte dos *frames* entre elas foi omitida. As linhas mais grossas são *frames* de novidade.

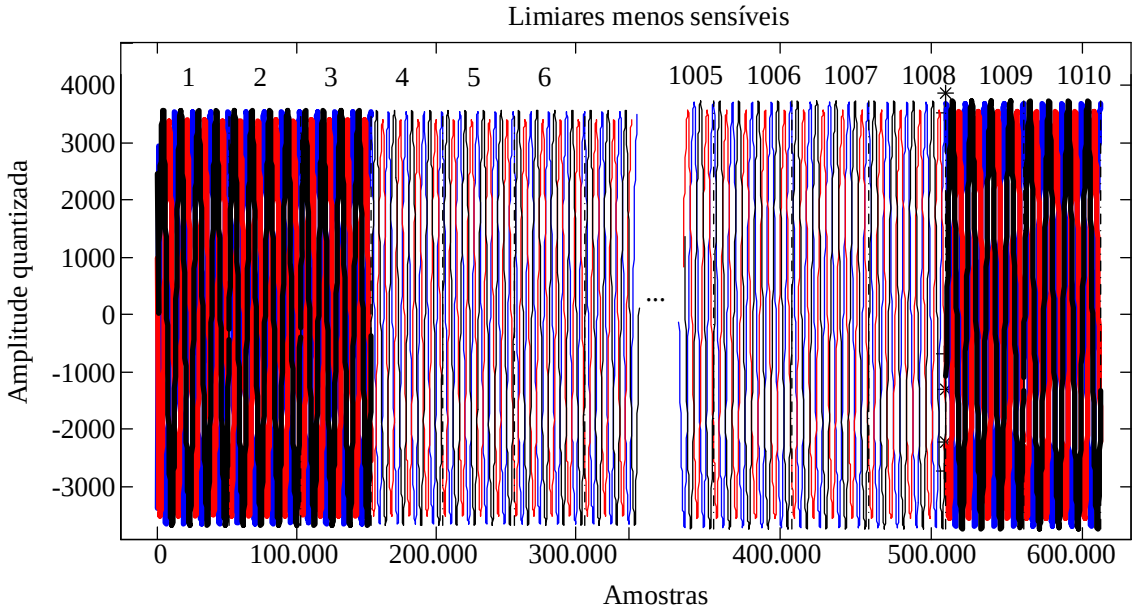


Figura 127: Limiares menos sensíveis. Parte final.

A Figura 128 apresenta uma ampliação ainda maior dessas extremidades e destaca os pontos `xn_1` e `xn_10` de cada fase, que ocorrem na transição de um *frame* sem novidade para um com novidade. Para este arquivo houve uma taxa de compressão de 80:1, visto que o arquivo compactado ocupa 38.400 bytes e só houveram detecções no início e no fim do sinal.

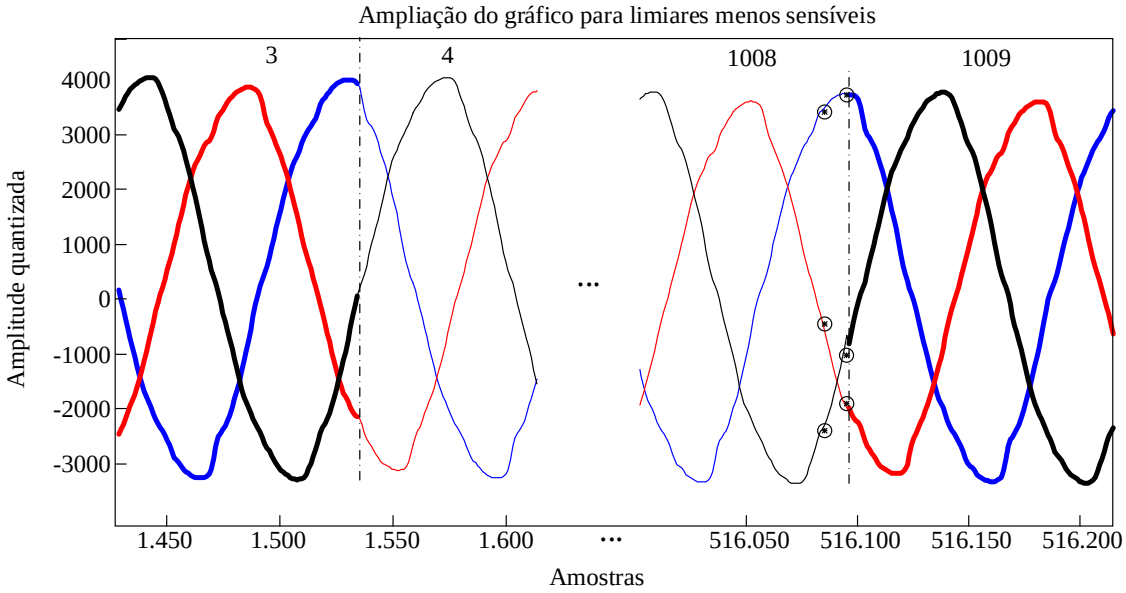


Figura 128: Transições de detecção (Ampliação da Figura 127).

- **Detecção de interrupção**

Para realizar esse tipo de teste, induziu-se um distúrbio do tipo interrupção no sinal aplicado. A interrupção durou aproximadamente 13 *frames* (aproximadamente 52 ciclos de 60 Hz $\cong$ 0,87 s). A Figura 129 mostra o impacto causado na tensão fornecida.

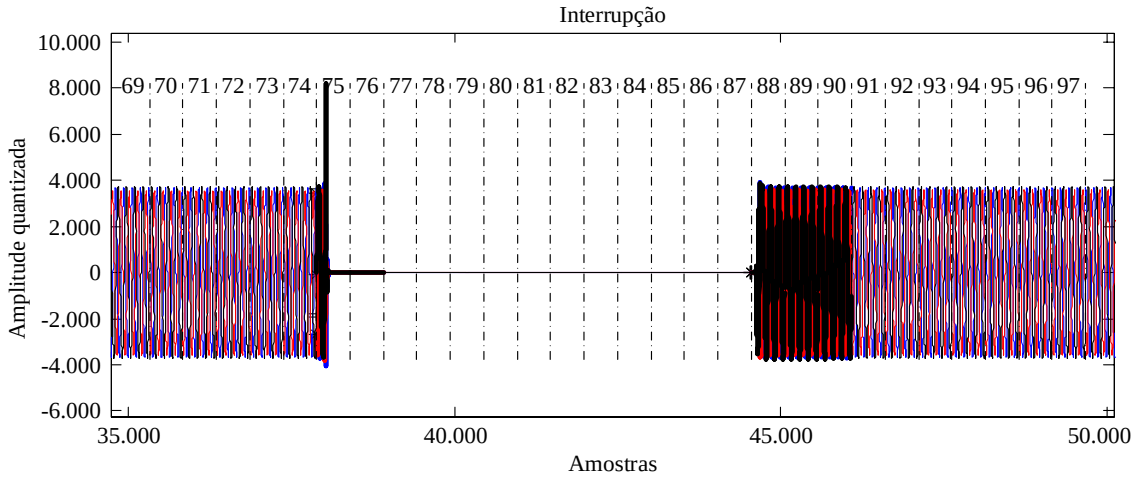


Figura 129: Detecção em um distúrbio do tipo interrupção.

Pode-se perceber que até o *frame* 74, não houve nenhuma detecção, excetuando-se as inicias (não mostradas na figura), que sempre ocorrem com a abertura de um novo arquivo. A partir do *frame* 74 até o *frame* 76 ocorreram novidades, pois os *frames* se diferem um do outro. Do *frame* 77 em diante até o *frame* 87, não ocorreram detecções. No *frame* 88 começa o restabelecimento da tensão e até o *frame* 90 são detectadas novidades, a partir do qual o sistema volta ao estado permanente. Assim, o SDCDE opera de forma inteligente, distinguindo as novidades presentes nos *frames*.

5.10. Conclusões do capítulo

Neste capítulo foram mostrados diversos testes, comprovando o funcionamento dos blocos que compõem o SDCDE. Os testes foram realizados visando avaliar as seguintes capacidades:

- Estimação: Avaliar a capacidade de estimação do algoritmo *Zero Crossing* implementado, frente às condições fornecidas pelo sistema; observar as estimações de energia dos *frames*, bem como a estimação da energia harmônica presente no sinal;

- Detecção: verificar a sensibilidade da detecção pelo detector baseado na energia do *frame*, detecções por variação da frequência fundamental e eventos com baixo THD;
- Compressão: quantificar a taxa de compressão total do arquivo compactado; avaliar o nível de perda da informação proveniente da aplicação de limiares nos coeficientes da *wavelet*.

Pelo que foi demonstrado através da análise dos resultados dos testes realizados com o SDCDE, pode-se concluir que o mesmo foi capaz de obter uma alta taxa de compressão, sem inserir grande perda de informação.

Outra conclusão que pode ser tirada é que a taxa de compressão é relativa e não pode ser facilmente deduzida, pois depende de alguns fatores, dentre eles:

- Dos limiares da curva de adaptação do patamar de novidade;
- Dos valores dos limiares de THD e de desvio de frequência escolhidos;
- E principalmente, do comportamento do sinal em questão.

6. CONCLUSÕES FINAIS

Esta dissertação propôs apresentar a implementação de um sistema inteligente, capaz de detectar e comprimir distúrbios, presentes em medições de parâmetros de sistemas elétricos de potência, o SDCDE. Inicialmente foram mostrados alguns métodos de compactação, dando-se ênfase nos que foram utilizados pelo sistema. Em seguida as implementações em FPGA e o desenvolvimento de um protótipo funcional foram descritos.

A metodologia proposta é baseada em detectar e salvar as novidades presentes no sinal, às quais são aplicadas técnicas de compressão com e sem perdas. Para isso, primeiramente submeteu-se o sinal a um seccionamento em *frames*. Utilizou-se uma detecção baseada na energia dos *frames*, auxiliada por outros detectores, um de desvio de frequência e outro, de estimação da energia do conteúdo harmônico do sinal.

A plataforma FPGA foi abordada dando-se enfoque no uso dos recursos de *hardware* pelos algoritmos de DSP implementados. Uma das técnicas em destaque na questão de otimização de lógica do DLP, foi o método utilizando o processador embarcado. Ele permitiu a flexibilidade de se implementar qualquer algoritmo utilizando o mesmo *hardware* e ainda permitiu a facilidade de ser programado utilizando-se um subconjunto da linguagem C.

Uma das vantagens, em termos de compressão, do SDCDE reside em aplicações nas quais os sinais sofrem pequenos e suaves desvios de frequência. Em tais situações não são detectadas novidades, exceto as iniciais e as finais. Isso permite uma alta taxa de compressão e ainda possibilita a reconstrução completa do sinal.

Em relação à sensibilidade do sistema, conclui-se que o mesmo pode ser tão sensível quanto se queira, bastando serem estabelecidos limites de comparação mais baixos. Com isso compromete-se a compressão, mas em contrapartida, pequenos eventos são detectados.

Quanto à taxa de compressão, verificou-se que a mesma é altamente dependente de vários fatores, dentre eles: (*i*) do comportamento dos sinais de entrada ao SDCDE; (*ii*) do nível de ruído e conteúdo harmônico presente nos mesmos; (*iii*) da parametrização configurada pelo usuário.

O protótipo funcional mostrou-se estável durante os testes. As leituras de tensões e correntes foram coerentes com os distúrbios aplicados. Além disso o dispositivo foi capaz de operar,

mesmo havendo queda ou falta de energia, revelando uma autonomia de bateria de aproximadamente 2 horas de funcionamento. A possibilidade de se armazenar os dados em um cartão de memória ou em um *pen drive*, permitiu a flexibilidade de transporte e reconstrução em um computador com um *software* adequado.

A estimação da frequência mostrou um mínimo de robustez perante ruído e injeção de harmônicos no sinal aplicado.

Enfim, conclui-se que o (SDCDE) foi capaz de cumprir com os objetivos que lhe foram propostos realizar. Mostrou-se robusto e eficaz, fornecendo altos níveis de compressão aos sinais medidos, e principalmente, mantendo o nível, arbitrado pelo usuário, de informação armazenada para posterior análise oscilográfica.

7. TRABALHOS FUTUROS

Prevê-se o desenvolvimento dos seguintes trabalhos futuros:

- Mudança de plataforma para o algoritmo LZW: atualmente, a compactação em termos de bit, com a implementação do algoritmo LZW, é realizada pelo processador ARM utilizado. Porém, deseja-se inserir esta etapa de compressão também na FPGA. Devido à quantidade reduzida de recursos presentes no *chip* atual, haja visto que foi utilizado o menor *kit* da ALTERA®, essa implementação comprometeria o resto do sistema. Entretanto, já foi iniciado o processo de migração de plataformas. Um código foi escrito para o processador embarcado na FPGA realizar a compressão, que foi comprovada através de simulações pelo Model*Sim*®. A próxima etapa é a gravação da FPGA e a verificação do funcionamento do algoritmo em tempo real, através da síntese em uma plataforma com mais recursos disponíveis.

- Criar um medidor de energia classe A, que atenda as normas de qualidade de energia (IEC 61000-4-30 ed3.0, 2015).

- Segmentação adaptativa: um dos grandes problemas para a reconstrução do sinal com fidelidade é a definição de *frame* de tamanho fixo. A reconstrução do sinal com *frame* fixo esbarra no problema da resolução, por exemplo, no caso atual em que 4 *frames* são armazenados a reprodução do sinal “estacionário”, através dos núcleos de reconstrução, limita-se a resolução de 15 Hz. Uma das possíveis soluções para este problema consiste trabalhar com segmentação adaptativa. Neste caso o tamanho do *frame* varia para conter as informações de baixa e alta frequência.

- Estimação do ruído: propõe-se desenvolver um bloco de *hardware* que seja capaz de estimar o ruído do sinal de entrada e adaptar o patamar para detecção de novidades de forma dinâmica, permitindo o estabelecimento do limiar do corte dos coeficientes de detalhe da *wavelet* de forma não-empírica. Essa função não anularia a configuração de um patamar fixo estabelecido pelo usuário do sistema, mas sim um acréscimo ao mesmo, podendo o operador escolher qual o tipo de limiar a ser utilizado.

- Detecção de novidade: além do método de comparação entre energias, propõe-se realizar uma comparação com outros métodos para detecção, a saber: filtro casado, redes neurais artificiais e estatísticas de ordem superior.

BIBLIOGRAFIA

AGILENT, K. T. **33120A Function/Arbitrary Waveform Generator**. Data Sheet: [s.n.]. 2015.

ALTERA®. **My First FPGA Design Tutorial**. San Jose: [s.n.], 2008. Disponível em: <http://www.altera.com/literature/tt/tt_my_first_fpga.pdf>. Acesso em: 17 fev. 2015.

ALTERA®. **Avalon Interface Specifications**. San Jose: [s.n.], 2015. Disponível em: <http://www.altera.com/literature/manual/mnl_avalon_spec.pdf>. Acesso em: 24 fev. 2015.

ALTERA®. DE0-Nano Development and Education Board. **www.altera.com**, 2015. Disponível em: <www.altera.com/education/univ/materials/boards/de0-nano/unv-de0-nano-board.html>. Acesso em: 02 fev. 2015.

ALTERA®. Stratix 10 FPGAs and SoCs: Delivering the Unimaginable. **www.altera.com**, 2015. Disponível em: <http://www.altera.com/devices/fpga/stratix-fpgas/stratix10/stx10-index.jsp>. Acesso em: 11 fev. 2015.

ANALOG DEVICES. **8-/6-/4-Channel DAS with 16-Bit, Bipolar Input, Simultaneous Sampling ADC - Data Sheet**. [S.l.]: [s.n.], 2012. Disponível em: <http://www.analog.com/media/en/technical-documentation/data-sheets/AD7606_7606-6_7606-4.pdf>. Acesso em: 01 mar. 2015.

AYODELE, K. P.; INYANG, I. A.; KEHINDE, L. O. An iLab for Teaching Advanced Logic Concepts With Hardware Descriptive Languages. **IEEE Transactions on Education**, Ile-Ife, Nigeria, v. PP, n. 99, p. 1, 4 fev. 2015. ISSN 0018-9359.

BETTA, G.; FERRIGNO, L.; LARACCA, M. Cost-Effective FPGA Instrument for Harmonic and Interharmonic Monitoring. **IEEE Transactions On Instrumentation And Measurement**, v. 62, n. 8, p. 2161-2171, ago. 2013. ISSN 0018-9456.

BHANDARI, S.; PUJARI, S. **Methodology for On the Fly Partial Reconfiguration For Computation Intensive Applications on FPGA**. International Conference on Computer Applications and Industrial Electronics (ICCAIE). Kuala Lumpur: [s.n.]. 5-7 dez. 2010. p. 597-601.

BOLZAN, M. J. A. Transformada em ondeleta: Uma necessidade. **Revista Brasileira De Ensino De Física**, v. 28, n. 4, p. 563-567, 2006. ISSN 1086-9126. Disponível em: <http://www.sbfisica.org.br/rbef/pdf/060309.pdf>. Acesso em: 26 jan. 2015.

CHAO, H. **Rate scalable video compression based on flexible block wavelet coding technique**. IEEE Fourth Workshop on Multimedia Signal Processing. Cannes: IEEE. 3-5 out. 2001. p. 415-420.

CHOONG, F. et al. **FPGA realization of power quality disturbance detection:** an approach with wavelet, ANN and fuzzy logic. IEEE International Joint Conference onNeural Networks (IJCNN) Proceedings. Montreal, Que.: IEEE. 31 jul. - 4 ago. 2005. p. 2613-2618.

CIOBOTARU, M.; TEODORESCU, R.; BLAABJERG, F. **A New Single-Phase PLL Structure Based on Second Order Generalized Integrator**. Power Electronics Specialists Conference, PESC. [S.l.]: [s.n.]. 18-22 jun. 2006. p. 1-6.

COSTA, F. B. Fault-Induced Transient Detection Based on Real-Time Analysis of the Wavelet Coefficient Energy. **IEEE Transactions On Power Delivery**, v. 29, n. 1, p. 140-153, fev. 2014. ISSN 0885-8977.

COVER, T. M.; THOMAS, J. A. **Elements of Information Theory**. 2. ed. Hoboken, New Jersey, USA: Wiley-Interscience, 2006.

CYCLONE II. **Device Handbook**. San Jose: ALTERA®, v. 1, 2008. Disponível em: <https://www.google.com.br/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=cyclone%20ii%20handbook%20volume%202>. Acesso em: 13 fev. 2015.

CYCLONE IV. Device Handbook. In: ALTERA® **Handbook**. [S.l.]: [s.n.], v. 1, 2010. Cap. 4. Embedded Multipliers in Cyclone IV Devices. Disponível em: <http://www.altera.com/literature/hb/cyclone-iv/cyiv-51004.pdf>. Acesso em: 15 fev. 2015.

CYCLONE IV. **Device Handbook**. San Jose: ALTERA®, v. 1, 2014. 4 p. Disponível em: <http://www.altera.com/literature/hb/cyclone-iv/cyclone4-handbook.pdf>. Acesso em: 11 fev. 2015.

DONNELLY, C.; STALLMAN, R. **Bison**. The YACC-Compatible Parser Generator: [s.n.], 2015. Disponível em: <http://www.gnu.org/software/bison/>. Acesso em: 25 fev. 2015.

DU, J.; LUO, Z.; WANG, J. **High-speed real-time signal processing system design and implementation based on FPGA**. 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC). Deng Leng: IEEE. 8-10 ago. 2011. p. 7096-7099.

DUQUE, C. A. **O Processador Digital de Sinais - Família TMS320FXX**. Juiz de Fora: [s.n.], 2004.

DUQUE, C. A. et al. Power quality event detection based on the divide and conquer principle and innovation concept. **IEEE Transactions on Power Delivery**, v. 20, n. 4, p. 2361-2369, out. 2005. ISSN 0885-8977.

FEMINE, A. D. et al. Power-Quality Monitoring Instrument With FPGA Transducer Compensation. **IEEE Transactions on Instrumentation and Measurement**, v. 58, n. 9, p. 3149-3158, 30 jun. 2009. ISSN 0018-9456.

FERNANDES,. 1. Conceitos básicos sobre VHDL. **www.vhdl.com.br**, 17 mai. 2014. Disponível em: <http://vhdl.com.br/site/vhdl/conceitos-basicos/148-conceitos-b%C3%A1sicos-sobre-vhdl>. Acesso em: 17 fev. 2015.

FERNANDES,. Conceitos básicos de um FPGA. **www.vhdl.com.br**, 15 mai. 2014. Disponível em: <http://vhdl.com.br/site/fpga/conceitos-basicos>. Acesso em: 17 fev. 2015.

FERRIGNO, L.; LANDI, C.; LARACCA, M. **FPGA-based Measurement Instrument for Power Quality Monitoring according to IEC Standards**. IEEE Instrumentation and Measurement Technology Conference Proceedings, IMTC. Victoria, BC: [s.n.]. 12-15 mai. 2008. p. 906-9011.

FINAMORE, W. A.; RIBEIRO, M. V. **Teoria da Informação**. UFJF. Juiz de Fora. 2013. (notas de aula).

FINKER, R. et al. **An intelligent embedded system for real-time adaptive extreme learning machine**. IEEE Symposium on Intelligent Embedded Systems (IES). Orlando, FL: [s.n.]. 9-12 dez. 2014. p. 61-69.

FUGAL, D. L. **Conceptual Wavelets**. Natick: Space & Signal Technical, 2009.

FUSCO, D. **Data Compression Of Biological Signals**. Engineering in Medicine and Biology Society, 1990., Proceedings of the Twelfth Annual International Conference of the IEEE. [S.l.]: IEEE. 1-4 nov. 1990. p. 891-892.

GREAVES, D.; SINGH, S. **Kiwi - Synthesis of FPGA Circuits from Parallel Programs**. 16th International Symposium on Field-Programmable Custom Computing Machines (FCCM). Palo Alto, CA: IEEE. 14-15 abr. 2008. p. 3-12.

HAIBING, Y. **An Efficient Lossless Image Compression Algorithm for External Memory Bandwidth Saving**. Data Compression Conference (DCC). Snowbird, UT: IEEE. 26-28 mar. 2014. p. 435.

HAMAD, M. et al. A High-Speed FPGA Implementation of an RSD-Based ECC Processor. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. PP, n. 99, p. 1, 29 jan. 2015. ISSN 1063-8210. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7027194&isnumber=4359553>. Acesso em: 04 fev. 2015.

HUANG, S.-J.; YANG, T.-M.; HUANG, J.-T. FPGA Realization of Wavelet Transform for Detection of Electric Power System Disturbances. **Transactions On Power Delivery**, v. 17, n. 2, p. 388-394, 2002. ISSN 0885-8977. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=997905&tag=1>. Acesso em: 27 dec. 2015.

IEEE STD 1800™-2012. **IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language**. New York,: IEEE, 2013. ISBN 978-0-7381-8110-3. Disponível em: <http://ieeexplore.ieee.org/servlet/opac?punumber=6469138>. Acesso em: 18 fev. 2015.

JOHNSON, J. List and comparison of FPGA companies. **www.fpgadeveloper.com**, 15 jul. 2011. Disponível em: <http://www.fpgadeveloper.com/2011/07/list-and-comparison-of-fpga-companies.html>. Acesso em: 11 fev. 2015.

JOOST , R.; SALOMON , R. **Advantages of FPGA-based multiprocessor systems in industrial applications**. 31st Annual Conference of IEEE Industrial Electronics Society (IECON). [S.l.]: IEEE. 6-10 nov. 2005.

KAPISCH, E. B. et al. **An electrical signal disturbance detector and compressor based on FPGA platform**. International Conference on Harmonics and Quality of Power (ICHQP). Bucharest: IEEE. 25-28 mai. 2014. p. 278-282.

KAPISCH, E. B. et al. **Implementação em FPGA de Transformada Wavelet para Compactaçõ de Sinais Elétricos de Sistemas de Potência Utilizando Processador Embarcado**. Congresso Brasileiro de Automática. Belo Horizonte: [s.n.]. 2014. p. 195-202.

KIEFFER, J. C.; YANG, E.-H. **Lossless data compression algorithms based on substitution tables**. IEEE Canadian Conference on Electrical and Computer Engineering. Waterloo, Ont.: IEEE. 24-28 mai. 1998. p. 629-632.

L. NANJING QINHENG ELECTRONICS. File manage and control chip CH376. **wch-ic.com/**, 2013. Disponível em: <http://wch-ic.com/product/usb/ch376.asp>. Acesso em: 02 mar. 2015.

LAI, C. S. **Compression Of Power System Signals With Wavelets**. International Conference on Wavelet Analysis and Pattern Recognition (ICWAPR). Lanzhou, China: IEEE. 13-16 jul. 2014. p. 109-115.

LEMPEL, A.; ZIV, J. A universal algorithm for sequential data compression. **IEEE Transactions on Information Theory**, v. 23, n. 3, p. 337-343, 1977. ISSN 0018-9448.

LIVANI, H.; EVRENOSOGLU, C. Y. A Machine Learning and Wavelet-Based Fault Location Method for Hybrid Transmission Lines. **IEEE Transactions On Smart Grid**, v. 5, n. 1, p. 51-59, jan. 2014. ISSN 1949-3053.

LOZANO , H.; ITO ,. **A Deeply Embedded Processor For Smart Devices**. International Conference on Smart Computing Workshops (SMARTCOMP Workshops). Hong Kong: IEEE. 5 nov. 2014. p. 79-86.

MARINKOVIC, B.; GILLETTE, M.; NING, T. **FPGA implementation of respiration signal classification using a soft-core processor**. Proceedings of the IEEE 31st Annual Northeast Bioengineering Conference. [S.l.]: [s.n.]. 2-3 abr. 2005. p. 54-55.

MCNAMARA,. IEEE P1364-2005, IEEE Standard Verilog Hardware Description Language. **www.verilog.com**, 2012. Disponível em: <http://www.verilog.com/>. Acesso em: 17 fev. 2015.

MEYER-BAESE, U. **Digital Signal Processing with Field Programmable Gate Array**. 3. ed. [S.l.]: Springer, 2013.

MINAEI, S.; YUCE, E. **High-order current-mode low-pass, high-pass and band-pass filter responses employing CCCIIs**. 6th International Conference on Information, Communications & Signal Processing. Cingapura: IEEE. 10-13 dez. 2007.

MITRA,. **Digital Signal Processing:** A Computer-based Approach. 4. ed. Califórnia: McGraw-Hill, 2011.

MORAES, F. G.; CALAZANS, N. L. V. **Parte 1 – Introdução à Simulação em VHDL**. Porto Alegre: [s.n.], 2013. Disponível em: <https://www.inf.pucrs.br/moraes/laborg/docs/laborg_parte1.pdf>. Acesso em: 17 fev. 2015.

MORAES, F. G.; CALAZANS, N. L. V. **Parte 2 - Introdução a FPGAs e Prototipação de Hardware**. Porto Alegre: [s.n.], 2014. Disponível em: <https://www.inf.pucrs.br/moraes/laborg/docs/laborg_parte2.pdf>. Acesso em: 12 fev. 2015.

NANDI, U.; MANDAL, J. J. **A Compression Technique Based On Optimality Of LZW Code (OLZW)**. 2012 Third International Conference on Computer and Communication Technology. Allahabad: IEEE. 2012. p. 166-170.

NEBEKER, F. **Signal Processing,The Emergence of a Discipline - 1948 to 1998**. [S.l.]: IEEE, 1998.

NIOS II - ALTERA®. **Nios II Processor Reference Handbook**. San Jose: [s.n.], 2014. Disponível em: <http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf?GSA_pos=1&WT.oss_r=1&WT.oss=nios%20ii%20handbook>. Acesso em: 25 fev. 2015.

PARSEH, R. et al. **Real-time compression of measurements in distribution grids**. IEEE Third International Conference on Smart Grid Communications (SmartGridComm). Tainan: IEEE. 5-8 nov. 2012. p. 223-228.

PAXSON, V. **Flex, version 2.5**. A Fast Scanner Generator: GNU, 1995. Disponível em: <https://www.gnu.org/software/flex/>. Acesso em: 25 fev. 2015.

PUTNAM, A. et al. **A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services**. 41st Annual International Symposium on Computer Architecture (ISCA). Minneapolis, MN: IEEE. 14-18 jun. 2014. p. 13-24.

QIONG, C.; ZHAO-HUI, W. **Research and Design of Portable Fault Recorder Based on FPGA**. The Proceedings of International Conference on Smart Grid and Clean Energy Technologies (ICSGCE). [S.l.]: [s.n.]. 27-30 set. 2011. p. 686-692.

QUARESMA, P. **Frequency Analysis of the Portuguese Language**. Coimbra: Centre for Informatics and Systems of the University of Coimbra, 2008. ISBN 0874-338X.

RIBEIRO, P. F. et al. **Power Systems Signal Processing for Smart Grids**. 1. ed. [S.l.]: Wiley, 2014.

ROMANO, J. M. T. **Algoritmo LZW (Lempel-Ziv-Welch)**. UNICAMP - Faculdade de Engenharia Elétrica e Computação. Campinas. 2001.

ROSE, J. et al. **The VTR Project - Architecture and CAD for FPGAs from Verilog to Routing**. Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays. New York, NY: [s.n.]. 2012. p. 77-86.

SARAVANAN, S. **Enhancing the performance of video compression on display devices using macro block classification and H.264 AVC**. IEEE International Conference on Computational Intelligence and Computing Research (ICCIC). Enathi: IEEE. 26-28 dez. 2013. p. 1-4.

SCHOEBERL, M. **Leros, A Tiny Microcontroller for FPGAs**. International Conference on Field Programmable Logic and Applications (FPL). Chania: [s.n.]. 5-7 set. 2011. p. 10-14.

SHANNON, C. E. A mathematical theory of communication. **The Bell System Technical Journal**, v. 27, n. 3, p. 379-423, jul. 1948. ISSN 0005-8580.

SHANNON, C. E. The Bandwagon. **IRE Transactions on Information Theory**, v. 2, n. 1, p. 3, 1956.

SILVA, C. E. M. D. A História da informática (Parte 6: Sistemas embarcados e supercomputadores). **Guia do Hardware.net**, 23 ago. 2011. Disponível em: <http://www.hardware.com.br/guias/historia-informatica/fpgas.html>. Acesso em: 04 fev. 2015.

SILVA, S. A. O. D.; NOVOCHADLO, R.; MODESTO, R. A. **Single-Phase PLL Structure Using Modified p-q Theory for Utility Connected Systems**. Power Electronics Specialists Conference, PESC IEEE. Rhodes: IEEE. 15-19 jun. 2008. p. 4706-4711.

SMITH, S. W. **Digital Signal Processing**. Burlington: Newnes, 2003. 650 p. Disponível em: <http://www.newnespress.com>.

STRATIX V. **Device Handbook**. San Jose: ALTERA, v. 1, 2015. Disponível em: <http://www.altera.com/literature/hb/stratix-v/stratix5_handbook.pdf>. Acesso em: 11 fev. 2015.

SUN, W.; WIRTHLIN, M. J.; NEUENDORFFER, S. FPGA Pipeline Synthesis Design Exploration Using Module Selection and Resource Sharing. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 26, n. 2, p. 254-265, jan. 2007. ISSN 0278-0070.

TEXAS INSTRUMENTS. **Tiva TM4C123GXL Microcontroller Datasheet**. [S.l.]: [s.n.], 2013.

TOCCI, R. J.; WIDMER, N. S.; MOSS, G. L. **Sistemas Digitais**. 10. ed. São Paulo: Prentice Hall, 2007.

TOMAR, A. Texas Instruments: EK-TM4C123GXL Tiva™ C Series LaunchPad Evaluation Kit. **elemnt14COMUNiTY**, 17 set. 2013. Disponível em: <http://www.element14.com/community/servlet/JiveServlet/showImage/102-55621-3-178408/EK-TM4C123GXL_2.PNG>. Acesso em: 02 mar. 2015.

VIRTEX-6. **Family Overview**. [S.l.]: XILINX®, 2012. Disponível em: <http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf>. Acesso em: 11 fev. 2015.

WANG, ; XIANG, X.; JINGMING, K. A novel multichannel audio signal compression method based on tensor representation and decomposition. **China Communications**, v. 11, n. 3, p. 80-90, mar. 2014. ISSN 1673-5447.

WANG, W. et al. Real-time High-quality Stereo Vision System in FPGA. **IEEE Transactions on Circuits and Systems for Video Technology**, v. PP, n. 99, p. 1, 03 jan. 2015. ISSN 1051-8215.

WYLIE, F. Digital audio data compression. **Electronics & Communication Engineering Journal**, v. 7, n. 1, p. 5-10, fev. 1995.

XU, Y.; SHUANG, K. **Implementation of high order matched filter on a FPGA chip**. 4th International Congress on Image and Signal Processing (CISP). Shanghai: IEEE. 15-17 out. 2011. p. 2526-2530.

YEUNG, R. W. **Information Theory and Network Code**. [S.l.]: [s.n.], 2007. 564 p.

YILDIZ, N. et al. Architecture of a Fully Pipelined Real-Time Cellular Neural Network Emulator. **IEEE Transactions on Circuits And Systems**, v. 62, n. 1, p. 130-138, 01 out. 2014. ISSN 1549-8328.

YIN, J. et al. **Large-Scale Data Challenges in Future Power Grids**. IEEE 7th International Symposium on Service Oriented System Engineering (SOSE). Redwood City, Califórnia, EUA: IEEE. 25-28 mar. 2013. p. 324-328.

ZHANG, M.; LI, K.; HU, Y. A High Efficient Compression Method for Power Quality Applications. **IEEE Transactions On Instrumentation And Measurement**, v. 60, n. 6, jul. 2011.

APÊNDICE A – VERILOG

Este apêndice tem por objetivo apresentar alguns tópicos da linguagem Verilog. Uma consideração importante a ser feita é que serão abordados apenas conceitos básicos sobre a linguagem. Um conteúdo muito mais aprofundado e completo pode ser encontrado na norma do grupo IEEE de padronização para Verilog (IEEE Std 1800™-2012, 2013).

A.1. SINTAXE

A linguagem Verilog possui algumas características de sintaxe, dentre as quais podem ser citadas:

- *Case-sensitive*: diferencia-se letras maiúsculas de minúsculas;
- Todas as palavras-chaves são formatadas em minúsculo: comandos e declarações;
- Espaços são utilizados apenas para legibilidade;
- Ponto e vírgula (`;`) é usado para o término de instruções (*statements*);
- Comentário de uma linha: `// ...`
- Comentário de diversas linhas: `/* ... */`

A.2. MÓDULOS

Um projeto em Verilog consiste em uma hierarquia modular. Módulos em níveis mais altos (*Top-Level*) encapsulam outros módulos em níveis inferiores e se comunicam com eles através de um conjunto declarado de portas (Figura 130). Cada módulo é projetado para executar uma parte da tarefa.

Em Verilog sempre se inicia um módulo com o comando `module` e termina-se o mesmo com o comando `endmodule`. Logo após o comando de início de um módulo, acrescenta-se o nome dele. Esse nome não pode ser o mesmo de nenhuma palavra-chave.

Para a criação do corpo de um módulo, com suas entradas, saídas e elementos internos procede-se da forma como é mostrada na Tabela 13.

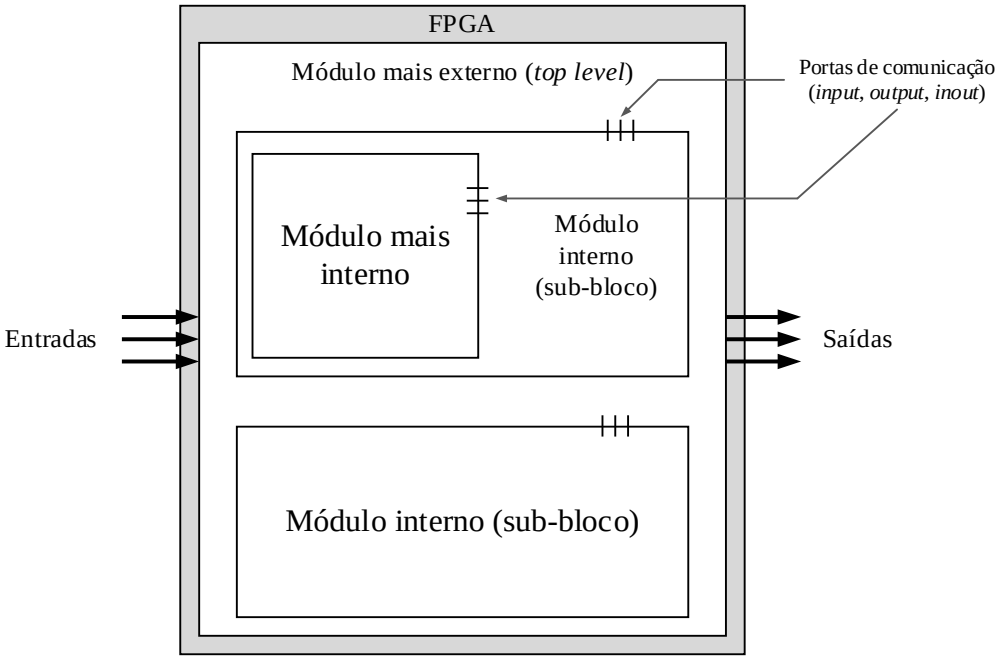


Figura 130: Hierarquia modular e comunicação através de portas numa FPGA.

Tabela 13: Regras para a criação do corpo de um módulo.

```
module nome (
             especificação e lista de portas,
             separadas por vírgula,
             exceto a última
            );
// especificação de tipos de dados
// funcionalidade do circuito
endmodule
```

- **Portas**

A lista de portas é situada após o nome do módulo e deve conter a especificação e a largura de bits. As portas podem ser dos tipos `input` – porta de entrada, `output` – porta de saída e `inout` – porta bidirecional. Portanto, a declaração de portas segue a regra:

```
<tipo_da_porta> [tamanho]<nome_da_porta>;
```

Um exemplo de declaração de portas é mostrado na Tabela 14. Nele, duas entradas de 8 bits são declaradas (`ina` e `inb`), uma entrada de *clock* (`clk`), uma de *clear* (`clr`). E umas saída de 16 bits (`out`).

Um exemplo de um módulo é dado através da Tabela 15. Nele, apenas as portas entradas e saídas do módulo *body* são definidas.

Tabela 14: Declaração de portas em Verilog.

```
input [7:0] ina, inb;
input clk, clr;
output [15:0] out;
```

Tabela 15: Exemplo de um módulo contendo apenas portas.

```
module body (
             input in1,
             input in2,
             output out
             );

endmodule
```

Através de uma ferramenta de visualização do arquivo *netlist*, (RTL *Viewer– Register Transfer Level Viewer*), é possível observar como as conexões e portas do circuito digital estão dispostas. A Figura 131 mostra uma representação para o *hardware* que seria gerado, caso o código da Tabela 15 fosse gravado numa FPGA.

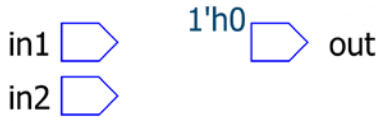


Figura 131: Representação das portas do código da Tabela 15.

- **Dados**

Existem dois tipos de dados fundamentais em Verilog, denominados *net* e *register*. O tipo *net* representa uma conexão física. O tipo *register* representa uma variável que será guardada temporariamente.

- ***Net***

Dentro do tipo de dado *net*, têm-se dois subtipos: o subtipo `wire`, que representa uma interconexão simples e o subtipo `tri`, que representa uma interconexão *tri-state* (portas do *Top-Level*).

A declaração de barramento de dados pode ser realizada seguindo-se a regra contida na Tabela 16. Um exemplo de declaração de barramentos, subtipo `wire`, pode ser visto na Tabela 17. Nela, a primeira linha declara um barramento de interconexão `wire` com 1 bit de largura para *clock*.

A segunda linha declara dois barramentos de interconexão com 16 bits cada um, o primeiro é denominado `multiplicador_out` e o segundo `somador_out`.

Tabela 16: Regra para declaração de dados.

```
<tipo_de_dado> [MSB : LSB] <nome_do_sinal> ;
```

Tabela 17: Exemplo de declaração de dados de interconexão do tipo *wire*.

```
wire clk ; // ligação de 1 bit para clock
wire [15:0] multiplicador_out, somador_out; // ligação de 16 bits
```

- ***Register***

Os dados do tipo *register* podem ser de dois subtipos: `reg` – agrupamento de bits de qualquer largura e `integer` – agrupamento de bits com largura de 32 bits. A declaração de barramentos do tipo *register* segue a mesma regra da Tabela 16. Um exemplo de declaração desses dados pode ser visto na Tabela 18.

Tabela 18: Exemplo de declaração de dados, tipo *register*.

```
reg valid_out;       //dado do tipo register de 1 bit
integer data_in;     //32 bits sinalizado
reg [15:0] data_out; // 16 bits
```

- ***Memória***

Para se descrever uma memória, é fundamental a utilização de dados do tipo *register*. Eles podem ser agrupados como mostra a Tabela 19. A Tabela 19 mostra um exemplo de declaração de memória. A implementação de uma memória ROM assíncrona, por exemplo, utiliza uma estrutura parecida com a apresentada.

Tabela 19: Exemplo de declaração de uma memória e mil posições de 32 bits.

```
reg [31:0]mem[0:1023]; // 1Kx32
reg [31:0] instr;
instr = mem[2];
```

- ***Parâmetro***

A declaração de um parâmetro, atribui um valor a um símbolo. A Tabela 20 mostra um exemplo de parâmetro, no qual se atribui o valor 8 ao parâmetro `size`. Na segunda linha, declara-se outras duas variáveis `a` e `b` com a largura dependente de `size`. Este exemplo tem a finalidade de mostrar como a largura de um barramento pode ser parametrizável.

Tabela 20: Exemplo de declaração e utilização de parâmetros.

```
parameter size = 8;  // atribui o valor 8 ao dado size
reg [size-1:0] a, b; // declara a e b como regs de 1 byte cada
```

- ***Atribuição de valores – números***

Especifica-se um número como é mostrado na Tabela 21.

Tabela 21: Especificação de um número em Verilog.

```
<sinal><número de bits>'<base><número>
```

As bases possíveis são decimal (`'d` ou `'D`), binaria (`'b` ou `'B`), hexadecimal (`'h` ou `'H`), octal (`'o` ou `'O`). Os padrões para base, largura de bits e sinal, quando não informados, são decimal, 32 bits e positivo, respectivamente.

Caso se queira declarar um número com um valor não definido (*don't care*), basta colocar o símbolo `x` ou `X` depois da base. Esta prática pode ser utilizada, por exemplo, na criação de máquinas de estado e multiplexadores. Nesses casos o compilador se encarrega de escolher um valor de tal maneira que melhore a eficiência para a síntese.

Quando se deseja atribuir um valor de alta impedância, utiliza-se o símbolo `z` ou `Z`, depois da base. A Tabela 22 mostra alguns exemplos de declaração de números.

Tabela 22: Exemplos de declarações de números.

```
3'b010  // número binário de 3 bits
b010    // 32'b010 → número binário de 32 bits (32 é o padrão)
10      // 32'd10 → número decimal de 32 bits (decimal é o padrão)
2'd7    // 2'd3 pois: 7 → 1 1 1 (o último bit e truncado)
-8'd3   // número 3, negativo, com 8 bits de largura
12'h12x // número hexadecimal de 12 bits com o (digito menos
```

```
        // significativo) desconhecido.
1'bz    // número de alta impedância de 1 bit.
```

- **Operações aritméticas**

As operações aritméticas básicas suportadas pelo Verilog estão descritas na Tabela 23.

Tabela 23: Operações aritméticas básicas da linguagem Verilog.

Símbolo	Função	Exemplo (a = 5; b = 10; c = 2'b01;)
+	Adição	b + c = 11
–	Subtração, negativo	b – c = 9; -b = -10
*	Multiplicação	a * b = 50
/	Divisão	b / a = 2
%	Modulus	b % a = 0

Caso algum operando tenha algum bit em alta impedância (z) ou seja desconhecido (x), o resultado é desconhecido (x). Se os operandos e o resultado são da mesma largura de bits, o bit de *carry* é perdido.

- **Operadores bit a bit**

As operações bit a bit estão descritas na Tabela 24. O resultado possui o número de bits do maior operando.

Tabela 24: Operadores bit a bit.

Símbolo	Função	Exemplo (a = 3'b101; b = 3'b110; c = 3'b01X)
~	Inverte cada bit	~a = 3'b010
&	AND bit a bit	a & b = 3'b100; b & c = 3'd010
|	OR bit a bit	a | b = 3'b111
^	XOR bit a bit	a ^ b = 3'b011
^~ ou ~^	XNOR bit a bit	a ^~b = 3'b100

- **Operadores de redução**

Estes operadores reduzem um vetor a um único bit. Eles estão expressos na Tabela 25.

Tabela 25: Operadores de redução.

Símbolo	Função	Exemplo (a = 5'b10101; b = 4'b0011; c = 3'bZ00; d = 4'bX011)
&	AND de todos os bits	&a = 1'b0 ; &d = 1'b0
~&	NAND de todos os bits	~&a = 1'b1
|	OR de todos os bits	|a = 1'b1; |c = 1'bX
~|	NOR de todos os bits	~|a = 1'b0
^	XOR de todos os bits	^a = 1'b1
^~ ou ~^	XNOR de todos os bits	~^a = 1'b0

Em alguns casos, mesmo sendo `x` e `z` considerados desconhecidos, o resultado pode ser um valor conhecido. Retirando-se o valor `d` da Tabela 25, a operação `&d = 1'b0` é um exemplo disso.

- **Operadores lógicos, relacionais e de igualdade**

São usados para comparar valores. Ao serem aplicados, retornam 1 bit como resposta: 0 para falso e 1 para verdadeiro. Se algum operando for `x` ou `z`, o resultado dá desconhecido. A Tabela 26 descreve esses operadores.

Tabela 26: Operadores relacionais e de igualdade.

	Símbolo	Função	Exemplo (a = 3'b010; b = 3'b100; c = 3'b111; d = 3'b01z; e = 3'b01x; f = 3'b101; g = 3'b000)
Relacionais	>	Maior	a > b resulta falso (1'b0)
	<	Menor	a < b resulta verdade (1'b1)
	>=	Maior ou Igual	a >= d resulta desconhecido (1'bX)
	<=	Menor ou Igual	a <= e resulta desconhecido (1'bX)
De igualdade	==	Igualdade	a == c resulta falso (1'b0)
	!=	Desigualdade	e != e resulta desconhecido (1'bX)
Lógicos	!	Não verdade	!a resulta em falso (1'b0)
	&&	Duas expressões verdadeiras	f && g resulta em falso (1'b0)
	||	Uma ou duas expressões verdadeiras	f || g resulta em verdade (1'b1)

- **Operadores de *shift***

Esses operadores são utilizados para deslocar um vetor para a direita ou para a esquerda, de acordo com a direção e o número de bits especificados. Os bits deslocados para fora da palavra são perdidos. A descrição desses operadores pode ser vista na Tabela 27.

Tabela 27: Operadores de *shift*.

	Símbolo	Função	Exemplo (`a = 4'b1010, b = 4'b10X0`)
Completa com zeros	>>	Shift para direita	`b >> 1;` resulta em `4'b010X`
	<<	Shift para esquerda	`a << 2;` resulta em `4'b1000`
Mantém o MSB (complemento de 2)	>>>	Shift para direita	`b >>> 1;` resulta em `4'b110X`
	<<<	Shift para esquerda	`a <<< 1;` resulta em `4'b0100`

- **Operadores diversos**

Outros operadores auxiliam no desenvolvimento das estruturas de *hardware*. Eles são mostrados na Tabela 28.

Tabela 28: Operadores diversos.

Símbolo	Função	Exemplo (`a = 3'b010, b = 4'b1100`)
`?:`	Condicional	`(condição) ? valor_verdadeiro : valor_falso;`
`{ }`	Concatenação	`{a,b}` resulta em `7'b0101100`
`{{ }}`	Replicação	`{3{2'b10}}` resulta em `6'b101010`

- **Ordem de precedência das operações**

Como em C, as operações também possuem precedência umas sobre outras (Figura 132). É aconselhável, porém, utilizar os parênteses “( )”, para se evitar problemas.

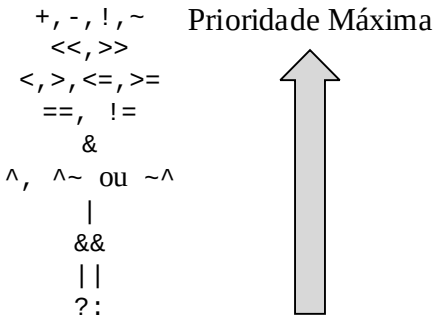


Figura 132: Ordem de prioridade para as operações em Verilog.

- **Modelagem da funcionalidade**

A partir de agora será mostrado como aplicar os conceitos vistos até aqui em uma modelagem de *hardware*. Em Verilog, existem duas formas para se descrever a funcionalidade de um circuito: a atribuição contínua (*continuous assignment*) e os blocos de processos (*procedural blocks*).

- ***Continuous assignment***

Na atribuição contínua a funcionalidade não varia. Um exemplo disso é um ganho aplicado a um sinal. Ela é utilizada para descrever circuitos combinacionais, nos quais a saída é modificada com a modificação das entradas. A variável que recebe o valor deve ser do tipo *net*, e o valor atribuído pode ser de qualquer tipo, *net* ou *register*. A atribuição pode ser declarada no momento da declaração da variável ou no decorrer do corpo do código. A Tabela 29 mostra exemplos desse tipo desta modelagem funcional.

Tabela 29: *Continuous assignment*.

```
1 wire[15:0] adder_out = in_A + in_B; //atribui o valor na
2                                     //declaração
3 wire[15:0] adder_out;               //declaração
4 assign adder_out = in_A + in_B      //atribuição
```

A Tabela 30, a Figura 133 e a Figura 134 mostram um exemplo de uma porta *and* com duas entradas de 1 bit. O resultado da lógica é atribuído continuamente à saída `out`.

Outro exemplo de atribuição contínua pode ser visto pela Tabela 31, pela Figura 135 e pela Figura 136. Nele é mostrada uma atribuição de multiplicação com duas entradas de 4 bits a uma saída de 8 bits.

Tabela 30: Atribuição contínua de lógica “and”.

```
1  module myand (
2                input ina, inb,
3                output wire out
4                );
5
6  assign out = ina & inb;
7
8  endmodule
```

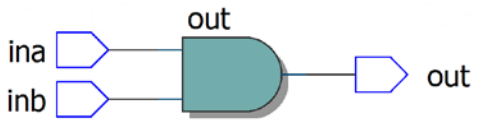


Figura 133: Representação do *hardware* descrito pelo código em Verilog descrito na Tabela 30.

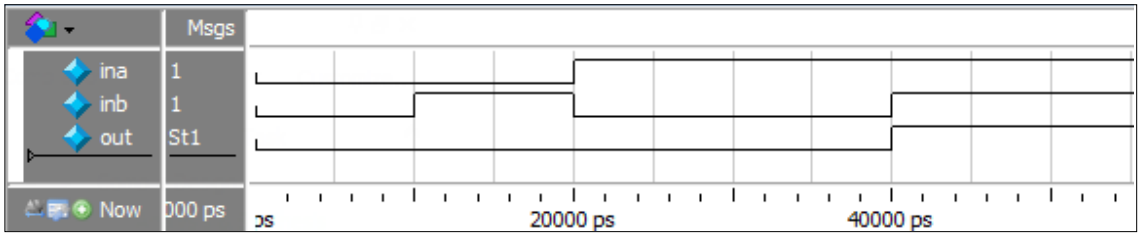


Figura 134: Diagrama temporal das entradas e saída do *hardware* da Figura 133.

Tabela 31 Atribuição contínua de multiplicação:

```
1  module mymult (
2                input signed [3:0] ina, inb,
3                output wire signed [7:0] out
4                );
5
6  assign out = ina * inb;
7
8  endmodule
```

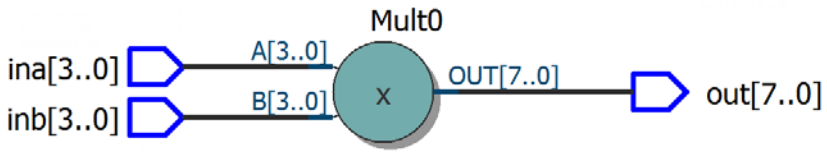


Figura 135: Representação do *hardware* descrito pelo código em Verilog descrito na Tabela 31.

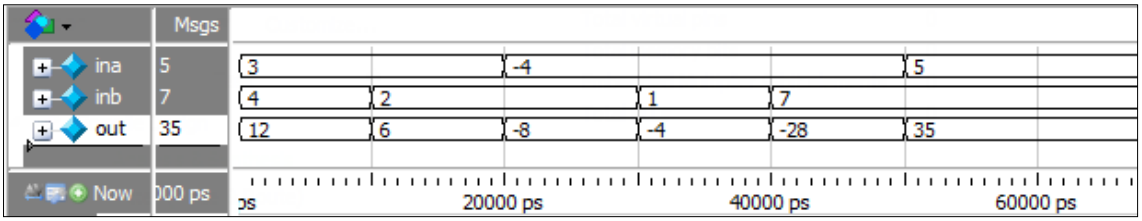


Figura 136: Diagrama temporal das entradas e saída do *hardware* da Figura 135.

- ***Procedural blocks***

Nessa modelagem é criado um processo que definirá uma ação, em que a mesma, pode possuir um comportamento variável mediante entrada, saídas e condições internas.

Por exemplo, blocos que dependam da variação de um determinado sinal para a atualização da saída, ou blocos sincronizados com a borda de um outro determinado sinal (*clock*). Tem-se dois tipos de *procedural blocks*: o bloco `always` e o bloco `initial`. Cada `always` ou `initial` representa um processo independente e são executados em paralelo.

Bloco *always*

O bloco `always` é sintetizável, ou seja, um código descritivo que o utiliza, pode ser implementado num *hardware*. Cada `always` criado no código representa a descrição de um comportamento que se repete com o tempo, resultando em vários processos que são executados em paralelo. Todos os processos começam ao mesmo tempo, no entanto, o desenrolar de cada um dependerá da lógica inferida no código. Nas operações realizadas dentro deste bloco, as variáveis que recebem valores devem ser, necessariamente, do tipo *register* e os valores recebidos podem ser quaisquer, inclusive uma parte de um barramento.

A Tabela 32 mostra um exemplo de bloco um bloco `always` dentro de um módulo. Neste exemplo é descrito um registrador de 16 bits. Toda borda positiva do sinal de *clock* (`clk`), o registrador carrega o valor colocado em sua entrada (`in`) e o coloca em sua saída (`out`). Caso o sinal de *clear* (`clr_n`) seja ativado (em zero), o registrador é limpo assincronamente.

As declarações entre parênteses após o `always`, formam a lista de sensibilidade e as declarações `if` e `else` dentro do bloco são as declarações de comportamento. O *hardware* descrito pelo código da Tabela 32 é mostrado na Figura 137.

Tabela 32: Descrição de um registrador.

```
 1 module regist (
 2                input clk, clr_n, enable,
 3                input signed [15:0] in,
 4                output signed [15:0] out
 5                );
 6
 7 reg signed [15:0] registra;
 8
 9 always @ (posedge clk or negedge clr_n)
10 begin
11    if (clr_n == 1'b0) registra <= 16'd0;
12    else if (enable) registra <= in;
13 end
14
15 assign out = registra;
16
17 endmodule
```

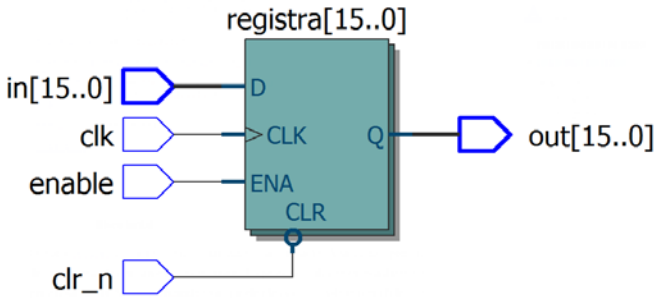


Figura 137: Representação do *hardware* descrito pelo código da Tabela 32.

Outra declaração de comportamento importante é o `case`. Com ela é possível criar estruturas de multiplexação, muito utilizadas em aplicações de DSP. Nessas estruturas, a lista de sensibilidade não é ativada por bordas crescentes ou decrescentes (*clocked process*), como no caso do registrador, mas sim, com a mudança de estado de qualquer variável da lista (*combinatorial process*). Em códigos cuja lista de sensibilidade é muito grande se torna mais fácil a substituição de todo o seu conteúdo por um asterisco "*".

A Tabela 33 mostra um exemplo de um multiplexador, utilizando a declaração de comportamento `case`. A Figura 138 mostra uma representação desse *hardware*.

Tabela 33: Descrição de um multiplexador.

```
 1  module mux_teste(
 2                   input a, b, c, d, e,
 3                   input [2:0] sel,
 4                   output reg out
 5                   );
 6
 7  always @(*)
 8  begin
 9     case (sel)
10        3'd1: out <= a;
11        3'd2: out <= b;
12        3'd3: out <= c;
13        3'd4: out <= d;
14        3'd5: out <= e;
15        default: out <= 1'bx;
16     endcase
17  end
18
19  endmodule
```

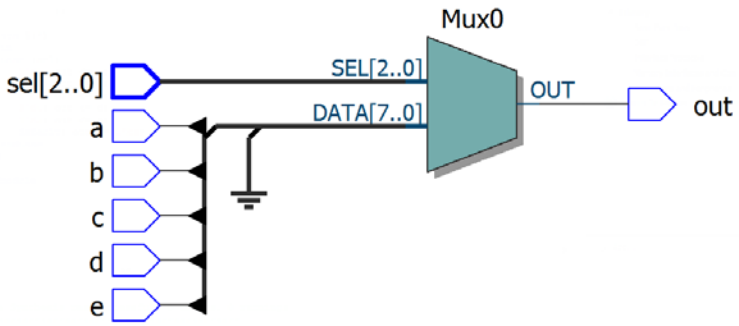


Figura 138: Representação do *hardware* gerado pelo código da Tabela 33.

Bloco *initial*

O bloco `initial` é não sintetizável e é utilizado para descrição de códigos de simulação, denominados *testbenches*. Através desse bloco pode-se definir as condições iniciais e os valores assumidos pelos estímulos de entrada de um circuito durante o período da simulação. Ele não possui lista de sensibilidade, isto se deve ao fato dele não repetir em nenhum momento. Cada bloco `initial` começa sua execução no tempo de simulação zero.

Os comandos de início e fechamento deste bloco podem ser `begin` e `end`, como no bloco always, ou `fork` e `join`. No primeiro caso, tem-se uma referência de tempo relativa ao último comando, já no segundo a referência é fixa no instante zero.

A Tabela 34 mostra um caso em que são utilizados os comandos `begin` e `end`. A Figura 139 é o resultado na simulação. A Tabela 35 e a Figura 140 mostram a utilização dos comandos `fork` e `join`.

Tabela 34: Utilização dos comandos *begin* e *end* para simulação.

```
1  module teste_tb ();
2  reg clr_n, enable;
3  initial
4  begin
5     clr_n <= 1'b0;
6     enable <= 1'b0;
7     #23 clr_n <= 1'b1;
8     #35 enable <= 1'b1;
9  end
10 endmodule
```

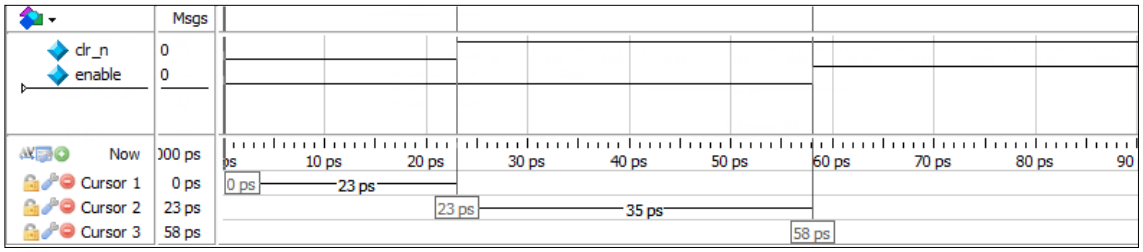


Figura 139: Diagrama temporal de simulação utilizando os comandos *begin* e *end* no bloco initial.

Tabela 35: Utilização dos comandos *fork* e *join* para simulação.

```
1  module teste_tb ();
2  reg clr_n, enable;
3
4  initial
5  fork
6     clr_n <= 1'b0;
7     enable <= 1'b0;
8     #23 clr_n <= 1'b1;
9     #35 enable <= 1'b1;
10 join
11 endmodule
```

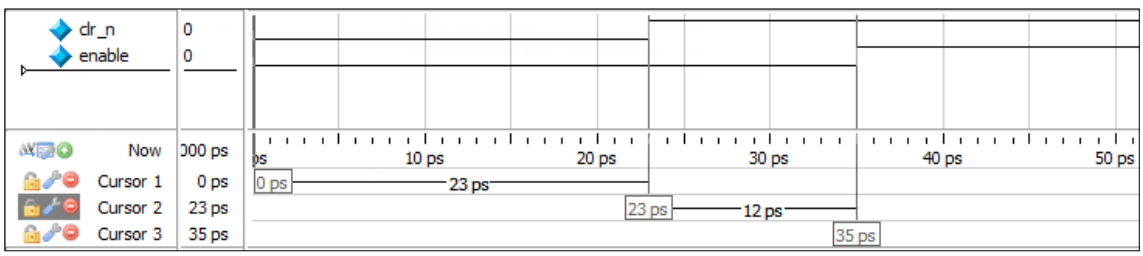


Figura 140: Diagrama temporal de simulação utilizando os comandos *fork* e *join* no bloco initial.

A.3. LIGAÇÃO DE MÓDULOS – INSTÂNCIAS

Como visto na Seção A.2 deste apêndice, pela Figura 130, a linguagem Verilog é uma linguagem modular. Os módulos permitem uma melhor organização do código e divisão de tarefas. A forma usada para colocar um bloco dentro de outro, ou mesmo, vários dentro de um único maior, é a instância. A instância é uma forma de "chamar" um bloco e conectá-lo com os devidos sinais em suas entradas e saídas. Pode-se instanciar o mesmo bloco de *hardware* quantas vezes necessário, mesmo sendo ele descrito apenas uma vez.

Para se instanciar um módulo, basta colocar o nome dele, o nome da instância e em seguida as conexões com as portas existentes nele. Existem duas formas de se instanciar um módulo, quanto à conexão de suas portas: pela ordem delas ou pelos seus nomes. A primeira é mais resumida, mas requere mais atenção, pois é mais suscetível a erros. A segunda é mais trabalhosa, porém, mais segura.

A Tabela 36 mostra duas instâncias do mesmo bloco *and* da Tabela 30, dentro de um mesmo módulo. Nela, os módulos `myand` são instanciados pelos nomes de suas portas. Já a Tabela 37 mostra novamente o mesmo caso, porém com as instâncias feitas pela ordem das portas. Em ambos os casos o *hardware* gerado é o mesmo e pode ser observado através da Figura 141. É importante notar a redução do texto utilizado na forma que utiliza ordem das portas.

Tabela 36: Instâncias pelos nomes das portas.

```
 1 module duas_myands_nome(
 2                         input a, b, c, d,
 3                         output out1, out2
 4                         );
 5
 6 myand myand1(                          //primeira instância
 7              .ina(a),
 8              .inb(b),
 9              .out(out1)
10              );
11
12 myand myand2(                          //segunda instância
13              .ina(c),
14              .inb(d),
15              .out(out2)
16              );
17
18 endmodule
```

Tabela 37: Instâncias pela ordem das portas.

```
1  module duas_myands_ordem(
2                           input a, b, c, d,
3                           output out1, out2
4                           );
5
6  myand myand1(a,b,out1);              //primeira instância
7
8  myand myand2(c,d,out2);             //segunda instância
9
10 endmodule
```

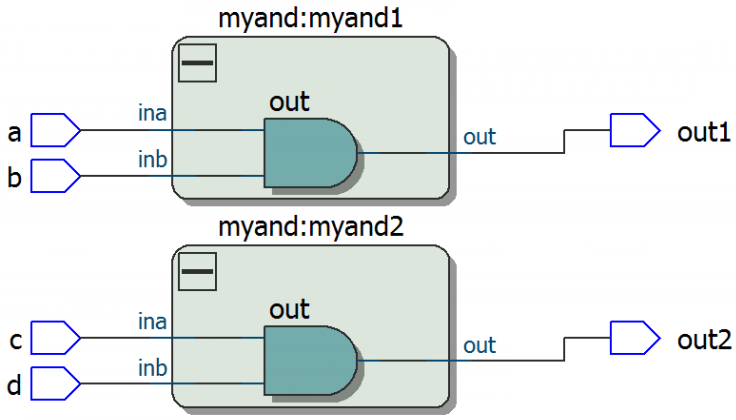


Figura 141: *Hardware* correspondente aos códigos da Tabela 36 ou da Tabela 37.

Através das instâncias e dos blocos, é possível serem formadas estruturas mais complexas, como filtros digitais, e outros elementos de DSP.

A.4. CÓDIGOS DE IMPLEMENTAÇÃO

Nesta seção são mostrados em tabelas, os códigos de algumas das implementações em FPGA descritas na Seção 3.5.

Tabela 38: Código descritivo de *hardware* para o filtro passa-altas da *wavelet* Daubechies 3 na representação direta transversal.

```
1  module filtro_transversal(
2                            input clk, en_filt, clr_n,
3                            input signed [15:0] ent,
4                            output wire signed [35:0] out
5                            );
6
7  parameter signed [15:0] a0 = -16'd10900, a1 = 16'd26440,
8                           a2 = -16'd15069, a3 = -16'd4424,
9                           a4 =  16'd2799,  a5 =  16'd1154;
10
11 reg signed [15:0] x1, x2, x3, x4, x5;
```

```
12
13 always @ (posedge clk or negedge clr_n)
14 begin
15    if (clr_n == 1'b0)
16    begin
17       x1  <= 16'd0;
18       x2  <= 16'd0;
19       x3  <= 16'd0;
20       x4  <= 16'd0;
21       x5  <= 16'd0;
22    end
23    else
24    begin
25       if (en_filt)
26       begin
27          x1  <= ent ;
28          x2  <= x1 ;
29          x3  <= x2 ;
30          x4  <= x3 ;
31          x5  <= x4 ;
32       end
33    end
34 end
35
36 assign out = ent*a0 + x1*a1 + x2*a2 + x3*a3 + x4*a4 + x5*a5;
37
38 endmodule
```

Tabela 39: Código descritivo de *hardware* para o filtro passa-altas da *wavelet* Daubechies 3 na representação transposta.

```
 1 module filtro_transversal(
 2                           input clk, en_filt, clr_n,
 3                           input signed [15:0] ent,
 4                           output wire signed [35:0] out
 5                           );
 6
 7 parameter signed [15:0] a0 = -16'd10900, a1 = 16'd26440,
 8                         a2 = -16'd15069, a3 = -16'd4424,
 9                         a4 =  16'd2799,  a5 =  16'd1154;
10
11 reg signed [15:0] reg0, reg1, reg2, reg3, reg4;
12
13 always @ (posedge clk or negedge clr_n)
14 begin
15    if (clr_n == 1'b0)
16    begin
17       reg4  <= 16'd0;
18       reg3  <= 16'd0;
19       reg2  <= 16'd0;
20       reg1  <= 16'd0;
21       reg0  <= 16'd0;
22    end
23    else
24    begin
25       if (en_filt)
26       begin
27          reg4 <= ent*a1 + reg3;
28          reg3 <= ent*a2 + reg2;
29          reg2 <= ent*a3 + reg1;
```

```
30          reg1 <= ent*a4 + reg0;
31          reg0 <= ent*a5;
32       end
33    end
34 end
35
36 assign out = ent*a0 + reg4;
37
38 endmodule
```

Tabela 40: Código em Verilog para o atrasador de cinco amostras.

```
 1 module atrasador_5amostras(
 2                             input wire unsigned clr_n,
 3                             input wire unsigned en_atr,
 4                             input wire unsigned clk,
 5                             input wire signed [15:0] x,
 6                             output wire signed [15:0] x0, x1, x2, x3,
 7 x4, x5
 8                             );
 9
10 reg signed [15:0] xn_1, xn_2, xn_3, xn_4, xn_5;
11
12 always @ ( posedge clk or negedge clr_n )
13 begin
14    if (clr_n == 1'b0)
15    begin
16       xn_1 <= 16'd0;
17       xn_2 <= 16'd0;
18       xn_3 <= 16'd0;
19       xn_4 <= 16'd0;
20       xn_5 <= 16'd0;
21    end
22    else
23    begin
24       if (en_atr)
25       begin
26          xn_1 <= x;
27          xn_2 <= xn_1;
28          xn_3 <= xn_2;
29          xn_4 <= xn_3;
30          xn_5 <= xn_4;
31       end
32    end
33 end
34
35 assign x0 = x;
36 assign x1 = xn_1;
37 assign x2 = xn_2;
38 assign x3 = xn_3;
39 assign x4 = xn_4;
40 assign x5 = xn_5;
41
42 endmodule
```

Tabela 41: Código da memória de coeficientes.

```
1 module memoria_HP(
2                    output reg signed [15:0] data = 16'd0,
```

```
                    input wire clr_n, clk,
                    input wire unsigned [2:0] add
                    );
wire signed [15:0] data_wire;

wire  signed  [15:0] mem[0:5];//06x16bits

parameter signed a0 = -16'd10900, a1 = 16'd26440,
a2 = -16'd15069, a3 = -16'd4424, a4 = 16'd2799,
a5 = 16'd1154; // coeficientes do filtro HP

assign mem[0] = a0;
assign mem[1] = a1;
assign mem[2] = a2;
assign mem[3] = a3;
assign mem[4] = a4;
assign mem[5] = a5;

assign data_wire = mem[add];

always @ (posedge clk or negedge clr_n)
begin
   if(clr_n == 1'b0)    data <= 16'd0;
   else data <= data_wire;
end

endmodule
```

Tabela 42: Código de descrição para o contador de coeficientes e amostras.

```
module cont_6_coef_am(
                      input wire unsigned clk, clr_n, en_cnt,
                      output reg unsigned [2:0] cnt
                      );

always @ (posedge clk or negedge clr_n) begin
      if (clr_n == 1'b0) cnt <= 3'd0;
      else if (en_cnt == 1'b1) cnt <= (cnt == 3'd5) ? 3'd0 : cnt +
3'd1;
end

endmodule
```

Tabela 43: Código do multiplexador.

```
module mux_6_entradas(
                    input wire unsigned [2:0] chn,
                    input wire clk, clr_n,
                    output reg signed [15:0] out_mux,
                    input wire signed [15:0] x0, x1, x2, x3, x4, x5
                    );

always @ (posedge clk or negedge clr_n)
      begin
            if(clr_n == 1'b0) out_mux <= 16'd0;
            else
            begin
                  case(chn)
                  4'd0: out_mux <= x0;
```

```
                    4'd1: out_mux <= x1;
                    4'd2: out_mux <= x2;
                    4'd3: out_mux <= x3;
                    4'd4: out_mux <= x4;
                    4'd5: out_mux <= x5;
                    default:    out_mux <= 16'dx;
                    endcase
            end
      end

endmodule
```

Tabela 44: Código descritivo do multiplicador.

```
module mult_HP(
                input wire clk, en_mult, clr_n,
                input wire signed [15:0] ent1,
                input wire signed [15:0] ent2,
                output reg signed [31:0] out_mult
                );

wire signed [31:0] out_mult_wire;

assign out_mult_wire = ent1 * ent2;

always @ (posedge clk or negedge clr_n)
begin
   if (clr_n == 1'b0) out_mult <= 32'd0;
   else if (en_mult == 1'b1) out_mult <= out_mult_wire;
end

endmodule
```

Tabela 45: Código descritivo para o acumulador.

```
module acc_HP(
              input wire clk, en_acc, clr_n, reset_acc_n,
              input wire signed [31:0] ent,
              output reg signed [34:0] out_acc
              );
wire signed [34:0] out_acc_wire;

assign out_acc_wire = ent + out_acc;

always @ (posedge clk or negedge reset_acc_n or negedge clr_n)
begin
   if (clr_n == 1'b0 | reset_acc_n == 1'b0) out_acc <= 32'd0;
else if (en_acc == 1'b1) out_acc <= out_acc_wire;
end

endmodule
```

Tabela 46: Código descritivo da máquina de estados.

```
module maq_HP(
              input wire valid_in,
              input wire clk, clr_n,
              input wire unsigned [2:0] cnt_reg,
```

```
5                output reg reset_acc_n en_atr, next, en_mult,
6                en_acc
7                );
8
9  // Declaração do registro de estados:
10 (* syn_encoding = "safe" *) reg [2:0] state;
11 // Declaração dos estados:
12 parameter S0 = 3'd0, S1 = 3'd1, S2 = 3'd2, S3 = 3'd3, S4 = 3'd4, S5
13 = 3'd5, S6 = 3'd6;
14
15 // Saída que dependem apenas do estado:
16 always @ (state, valid_in) begin
17    case (state)
18       S0:
19       begin
20          en_atr <= 1'b0;
21          next <= 1'b0;
22          en_mult <= 1'b0;
23          en_acc <= 1'b0;
24          reset_acc_n <= 1'b1;
25       end
26       S1:
27       begin
28          en_atr <= 1'b0;
29          next <= 1'b0;
30          en_mult <= 1'b0;
31          en_acc <= 1'b0;
32          reset_acc_n <= 1'b0;
33       end
34       S2:
35       begin
36          en_atr <= 1'b0;
37          next <= 1'b1;
38          en_mult <= 1'b0;
39          en_acc <= 1'b0;
40          reset_acc_n <= 1'b1;
41       end
42       S3:
43       begin
44          en_atr <= 1'b0;
45          next <= 1'b0;
46          en_mult <= 1'b1;
47          en_acc <= 1'b0;
48          reset_acc_n <= 1'b1;
49       end
50       S4:
51       begin
52          en_atr <= 1'b0;
53          next <= 1'b1;
54          en_mult <= 1'b0;
55          en_acc <= 1'b1;
56          reset_acc_n <= 1'b1;
57       end
58       S5:
59       begin
60          en_atr <= 1'b0;
61          next <= 1'b0;
62          en_mult <= 1'b0;
63          en_acc <= 1'b1;
64          reset_acc_n <= 1'b1;
65
```

```
66        end
67        S6:
68        begin
69           en_atr <= 1'b1;
70           next <= 1'b0;
71           en_mult <= 1'b0;
72           en_acc <= 1'b0;
73           reset_acc_n <= 1'b1;
74        end
75        default:
76        begin
77           en_atr <= 1'bx;
78           next <= 1'bx;
79           en_mult <= 1'bx;
80           en_acc <= 1'bx;
81           reset_acc_n <= 1'bx;
82        end
83     endcase
84  end
85  // Determinação do próximo estado
86  always @ (posedge clk)
87  begin
88     if (clr_n == 1'b0)
89     state <= S0;
90     else
91     begin
92        case (state)
93           S0:
94           begin
95              if (valid_in == 1'b1)
96              state <= S1;
97              else
98              state <= state;
99           end
100          S1: state <= S2;
101          S2: state <= S3;
102          S3:
103          begin
104             if (cnt_reg <= 3'd4) state <= S4;
105             else state <= S5;
106          end
107          S4: state <= S3;
108          S5: state <= S6;
109          S6: state <= S0;
110          default: state <= S0;
111       endcase
112    end
113 end
114 endmodule
```

Tabela 47: Código descritivo para o agrupamento da memória com o contador.

```
1  module memoria_HP_cont(
2                         output wire signed [15:0] coef,
3                         output wire unsigned [2:0] cnt,
4                         input wire clk, clr_n, next
5                         );
6
7  cont_6_coef_am contador (
8                           .clk(clk),
```

```
                          .reset_n(clr_n),
                          .en_cnt(next),
                          .cnt(cnt)
                          );

memoria_HP memoria(
                    .clk(clk),
                    .clr_n(clr_n),
                    .data(coef),
                    .add(cnt)
                    );


endmodule
```

Tabela 48: Código do MAC.

```
module mac_HP(
               input wire clk, clr_n, reset_acc_n, en_mult,
               en_acc,
               input wire signed [15:0] coef,
               input wire signed [15:0] out_mux,
               output wire signed [34:0] out_acc,
               );

wire signed [31:0] out_mult;


mult_HP multip(
                .clk(clk),
                .en_mult(en_mult),
                .clr_n(clr_n),
                .ent1(coef),
                .ent2(out_mux),
                .out_mult(out_mult)
                );

acc_HP accum(
              .clk(clk),
              .clr_n(clr_n),
              .en_acc(en_acc),
              .reset_acc_n(reset_acc_n),
              .ent(out_mult),
              .out_acc(out_acc)
              );

endmodule
```

Tabela 49: Código em Verilog do *Top Level* `filtro_HP`

```
module filtro_HP(
                  input wire valid_in,
                  input wire clk, clr_n,
                  input wire signed [15:0] x,
                  output wire signed [31:0] out_acc,
                  output wire valid_out
                  );
wire signed [15:0] x0, x1, x2, x3, x4, x5;
wire signed [15:0] out_mux;
```

```
10  wire signed [31:0] out_mult;
11
12  wire next, en_atr, reset_acc_n, en_mult, en_acc;
13  wire unsigned [2:0] cnt;
14  wire signed [15:0] coef;
15
16  reg unsigned [2:0] cnt_reg;
17
18
19  always @ (posedge clk)
20  cnt_reg <= cnt;
21
22  atrasador_5amostras atr(
23                          .clr_n(clr_n),
24                          .en_atr(en_atr),
25                          .clk(clk),
26                          .x(x),
27                          .x0(x0),
28                          .x1(x1),
29                          .x2(x2),
30                          .x3(x3),
31                          .x4(x4),
32                          .x5(x5)
33                           );
34
35  memoria_HP_cont mem(
36                      .clk(clk),
37                      .next(next),
38                      .clr_n(clr_n),
39                      .cnt(cnt),
40                      .coef(coef)
41                      );
42
43  mux_6_entradas multiplex(
44                           .clk(clk),
45                           .clr_n(clr_n),
46                           .chn(cnt),
47                           .out_mux(out_mux),
48                           .x0(x0),
49                           .x1(x1),
50                           .x2(x2),
51                           .x3(x3),
52                           .x4(x4),
53                           .x5(x5)
54                           );
55
56  mac_HP mac(
57              .clk(clk),
58              .clr_n(clr_n),
59              .reset_acc_n(reset_acc_n),
60              .en_mult(en_mult),
61              .en_acc(en_acc),
62              .coef(coef),
63              .out_mux(out_mux),
64              .out_mult(out_mult),
65              .out_acc(out_acc)
66              );
67
68  maq_HP m(
69           .valid_in(valid_in),
70           .clk(clk),
```

```
71          .clr_n(clr_n),
72          .cnt_reg(cnt_reg),
73          .en_atr(en_atr),
74          .next(next),
75          .en_mult(en_mult),
76          .en_acc(en_acc),
77          .reset_acc_n(reset_acc_n)
78          );
79
80  assign valid_out = en_atr;
81
82  endmodule
```

APÊNDICE B – CONVERSOR AD

B.1. INTRODUÇÃO

Este apêndice visa mostrar detalhes de funcionamento do conversor AD e fornecer informações extras que podem ser utilizadas para auxiliar no entendimento da Subseção 4.3.4.

Para este trabalho, foi utilizado o conversor AD7606™ da empresa fabricante ANALOG DEVICES®. As informações contidas neste apêndice foram baseadas *data sheet* do dispositivo (ANALOG DEVICES, 2012), onde podem ser encontradas ainda mais detalhes sobre o funcionamento do mesmo. Assim, o que será descrito nesta parte, são configurações direcionadas à aplicação em questão.

B.2. DESCRIÇÃO GERAL E CARACTERÍSTICAS

O conversor AD7606™ é um conversor que possui até 8 canais de 16 bits, *single-ended*, que podem ser convertidos simultaneamente. Cada canal contém um grampo de proteção contra elevadas tensões na entrada e um filtro *antialiasing* de segunda ordem do tipo *Butterworth*. O método de conversão utilizado pelo conversor AD7606™ é o de aproximações sucessivas, que tem por características o fato de gastar o mesmo tempo de conversão independentemente do tamanho da amplitude do sinal analógico de entrada (Tocci, Widmer, & Moss, 2007). Ele possui capacidade de leitura paralela em alta velocidade. A tensão de alimentação do circuito do conversor (denominada AV_{CC}) é de 5 V e ele admite sinais analógicos de entrada bipolares com amplitudes de ±5 V e ±10 V, dependendo do range de entrada que se escolha. A taxa de amostragem do sinal de entrada pode ir até 200 mil amostras por segundo (200 kSPS). O grampo de proteção tolera tensões de até ±16,5 V na entrada, independente do range que se escolha. Possui impedância de entrada de 1 MΩ independente da frequência de amostragem. A saída digitalizada do conversor AD7606™ é em complemento de 2. É fabricado apenas no formato TQFP (*Thin Quad Flatpack Package*) com 64 pinos ao todo (ANALOG DEVICES, 2012). Uma representação da vista superior do *chip* do AD7606™ pode ser observada através da Figura 142[6].

[6] Para visualizar corretamente o diagrama de pinos, a impressão desta parte do documento dever ser realizada com tinta colorida, caso contrário não será possível discernir a diferenciação das cores na legenda.

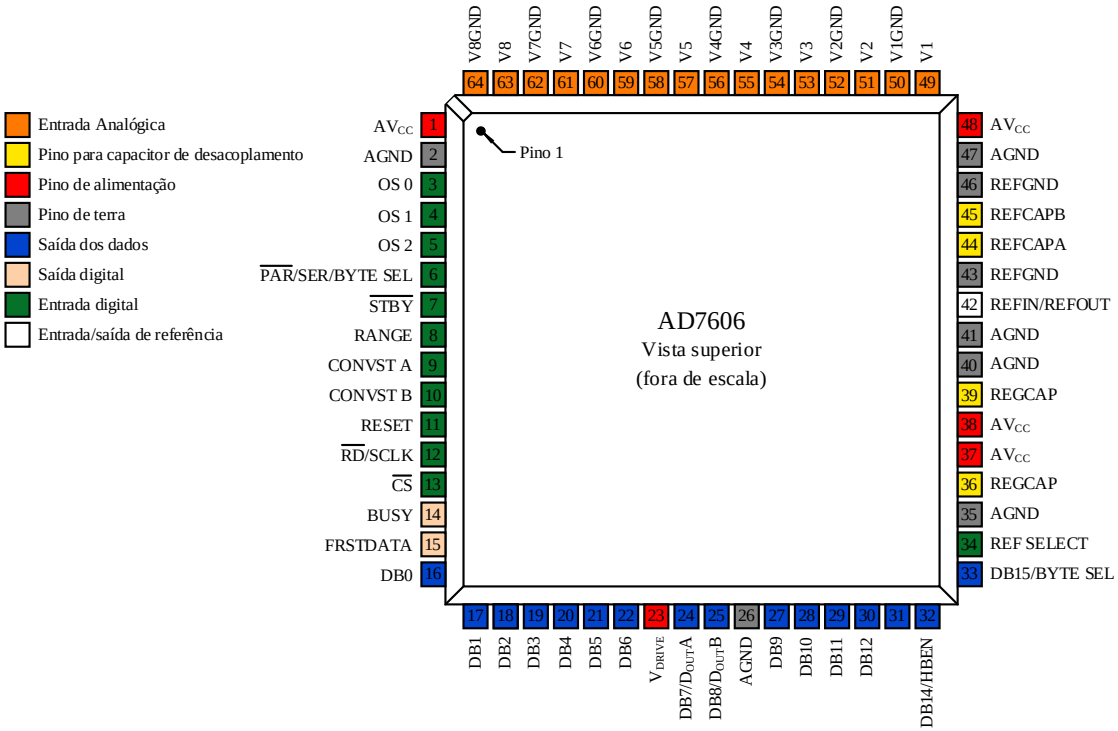


Figura 142: Os pinos do AD7606™ e suas designações.

B.3. FUNÇÕES DOS PINOS E SUAS CONFIGURAÇÕES

A Tabela 50 descreve as funções dos pinos do conversor.

Tabela 50: Funções dos pinos do AD7606™.

Nome	Número(s)	Função
AV_{CC}	1, 37, 38, 48	Pinos de alimentação de 5 V.
AGND	2, 26, 35, 40, 41, 47	Pinos de terra.
OS[2:0]	5, 4, 3	Pinos para modo de taxa de sobreamostragem.
$\overline{\text{PAR}}$ /SER/ BYTE SEL	6	Seletor de leitura serial ou paralela. Quando em 0, a leitura é paralela.
$\overline{\text{STBY}}$	7	Coloca o *chip* em modo *standby* quando ligado a 0.
RANGE	8	Define o nível de tensão analógica de entrada. Se for 0, é ±5 V.
CONVSTA CONVSTB	9, 10	Quando ligados juntos, a borda crescente sinal digital nesses pinos, solicita a conversão dos 8 canais simultaneamente.
RESET	11	A borda crescente do sinal nesse pino reinicializa o *chip*. Aborta a conversão caso esteja em andamento, e zera os registradores de saída.
$\overline{\text{RD}}$ /SCLK	12	Após o barramento de saída da conversão ser retirado de *three-state* com $\overline{\text{CS}}$ ligado em 0, as bordas de descida nesse pino expõem as

		conversões no barramento, na ordem dos canais: de V1 a V8.
$\overline{CS}$	13	Pino de *chip select*, ativo em baixo, tira o barramento de saída das conversões (DB[15:0]) do *three-state*.
BUSY	14	Enquanto estiver em nível lógico alto, a conversão está sendo efetuada. Ao cair, os dados já podem ser lidos.
FRSTDATA	15	Vai para nível alto indicando a presença do resultado da conversão do canal V1 no modo paralelo. Depois que o canal V1 é lido, volta ao zero.
DB[15:0]	16 - 22, 24, 25, 27 - 33	Barramento de saída de 16 bits da conversão.
V_{DRIVE}	23	Pino de tensão lógica de alimentação (3,3V).
REF SELECT	34	Se for ligado a 1, a referência de tensão interna é habilitada. Caso seja ligado em 0, aplicar tensão de referência no pino 42.
REGCAP	36, 39	Ligar um capacitor de 1µF entre cada um deles e o plano de terra
REFIN/ REFOUT	42	Caso seja escolhida referência interna, pelo pino REF SELECT, uma tensão de 2,5 V fica disponível nesse pino. Caso contrário, a tensão deve ser aplicada nesse pino. Em ambos os casos deve-se ligar um capacitor de 10µF entre ele e o plano de terra.
REFGND	43, 46	Devem ser ligados ao terra.
REGCAPA/ REGCAPB	44, 45	Devem ser curto-circuitados e desacoplados do terra através de um capacitor cerâmico de 10µF com baixo ESR (Resistência Equivalente em Série).
V*i* (*i* = 1...8)	49, 51, 53, 55, 57, 59, 61, 63	São as entradas analógicas dos 8 canais.
V*i*GND (*i* = 1...8)	50, 52, 54, 56, 58, 60, 62, 64	Pinos de terra de cada canal. Devem ser ligados ao plano de terra do circuito.

A faixa de frequência de amostragem do sinal analógico de entrada pode ser alterada de acordo com as opções mostradas na Tabela 51, a configuração da faixa adequada se dá através dos pinos de entrada digital 3, 4 e 5 (OS[2:0]).

Tabela 51: Codificação para diferentes frequências de amostragem.

OS[2:0]	Relação de *Ovesampling*	Máxima frequência de amostragem (CONVST kHz)
000	Desativado	200
001	Por 2	100
010	Por 4	50
011	Por 8	25
100	Por 16	12,5
101	Por 32	6,25
110	Por 64	3,125
111	Inválido	———

Para a aplicação deste trabalho, a taxa de amostragem é de 128 pontos por ciclo de 60 Hz, o que resulta em uma frequência de amostragem (F_s) de 7,68 kHz. Portanto, pode-se escolher uma relação por 16, que permite uma frequência de amostragem de até 12,5 kHz. Para isso, o sinal de entrada digital OS[2:0] = [100]. A opção está destacada na Tabela 51.

Como o modo de leitura escolhido foi leitura paralela, o sinal no pino 6 de entrada digital ($\overline{\text{PAR}}$ /SER/BYTE SEL) deve ser 0.

Os modos de baixo consumo estão relacionados com as combinações das configurações dos pinos 7 e 8 ($\overline{\text{STBY}}$ e RANGE, respectivamente) e podem ser visualizados através da Tabela 52.

Tabela 52: Modos de baixo consumo.

Modo de baixo consumo	$\overline{\text{STBY}}$	RANGE
Desativado	1	x
Standby	0	1
Desligado	0	0

Portanto, para manter o *chip* sempre ativo, aplica-se um nível lógico alto de forma permanente no pino 7.

O range de tensão escolhida para aplicação é de ±5 V, sendo assim necessária a aplicação de nível lógico baixo no pino 8.

Como a leitura é feita de forma paralela, optou-se por ligar os dois pinos de CONVST (CONVSTA e CONVSTB) na a mesma entrada do *clock* de amostragem (7,68 kHz). Doravante,

denominar-se-á CONVSTA e CONVSTB com o único termo CONVST devido a estarem em curto-circuito.

B.4. MODO DE OPERAÇÃO

Um típico diagrama de conexão é mostrado na Figura 143. O bloco responsável por controlar o conversor AD é o controlador do AD (ou Interface AD). Este bloco fornece os sinais de controle, recebe os sinais de resposta do AD e coleta as conversões nos momentos corretos.

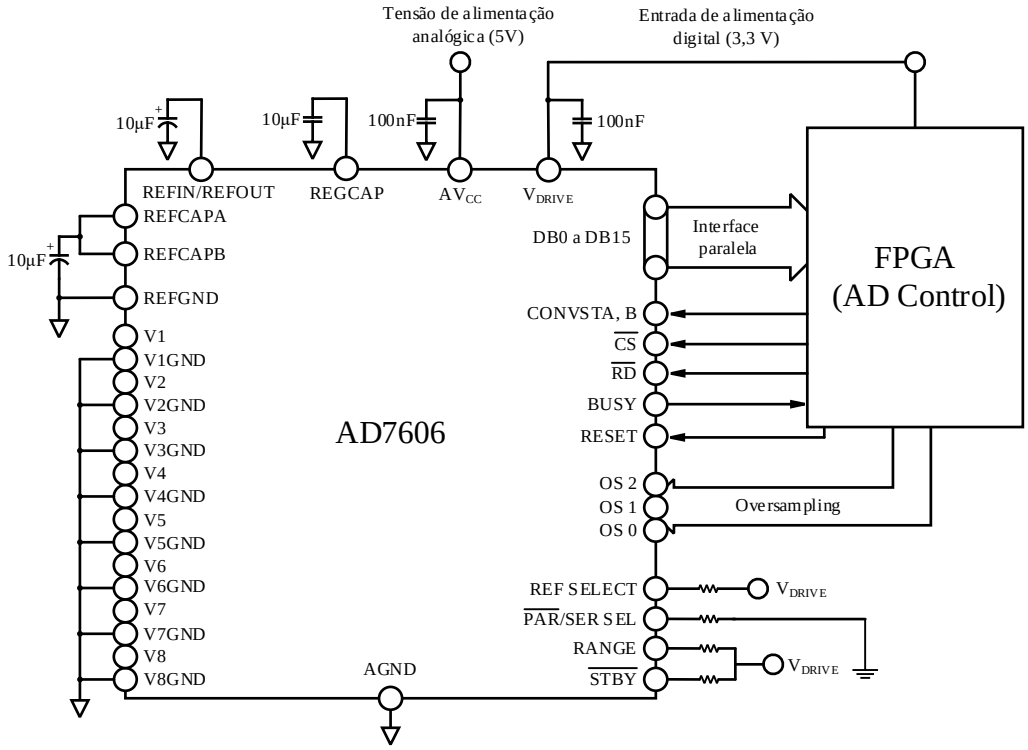


Figura 143: Conexão típica para a leitura paralela.

O diagrama temporal da operação do conversor pode ser visto na Figura 144.

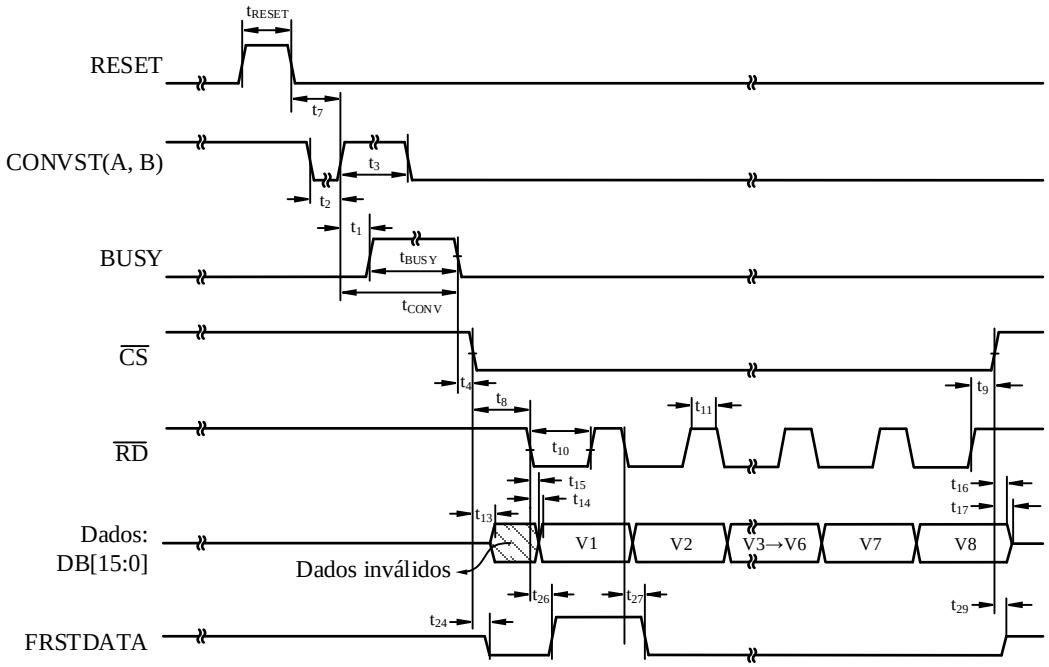


Figura 144: Diagrama temporal do conversor AD7606™ para leitura paralela.

Inicialmente um pulso de RESET pode ser aplicado, dentre outras, com as seguintes finalidades:

- Garantir que o conversor esteja configurado para o modo de operação correto;
- Para voltar de um modo de baixo consumo, caso seja a questão;
- Depois de mudar a configuração de referência de tensão (interna ou externa);
- Depois de ligado, para garantir que as entradas analógicas estão configuradas com o range de tensão selecionado.

Portanto, mediante qualquer alteração em configuração é essencial executar o RESET. Para que ele tenha adequado efeito, deve-se permanecer com o pulso em nível lógico alto por no mínimo t_{RESET} = 50 ns.

A partir do momento em que os pinos de configuração estão todos estáveis e o pulso de RESET foi dado, é possível começar a operação de conversão através do sinal de CONVST (A e B, borda crescente). Uma vez aplicado um sinal de CONVST, a borda crescente faz com que o circuito *front-end* de *track-and-hold* seja configurado para *hold*. Deve haver uma distância temporal de no mínimo t_7 = 25 ns entre a queda do RESET e a subida de CONVST. O tempo mínimo que CONVST deve permanecer em baixo é t_2 = 25 ns. O tempo mínimo que deve permanecer em alto é t_3 = 25 ns. Neste trabalho, como a taxa de amostragem é de F_s = 7,68

kHz, o período é de aproximadamente 130 μs, garantindo t_2 e t_3 muito maiores do que o mínimo permitido.

Depois da borda crescente de CONVST, a conversão foi solicitada, então um pulso positivo de BUSY é dado como a resposta do conversor. O tempo de demora entre a subida de CONVST e a subida de BUSY é de no máximo t_1 = 45 ns.

O tempo de conversão t_{CONV} de todos os canais é considerado como a soma de t_1 com t_{BUSY} e depende da taxa de sobreamostragem (ou *oversampling*). Para o caso deste, com modo de sobreamostragem com relação por 16 (OS[2:0] = 100), o tempo máximo para t_{CONV} é de 78μs, maior do que t_2 ou t_3.

Após BUSY cair $\overline{CS}$ pode também ir para zero instantaneamente, portanto t_4 poder ser 0 ns. A queda de $\overline{CS}$ retira o barramento DB[15:0] do estado de alta impedância (pinos *three-state*) em no máximo t_{13} = 37 ns.

Depois que $\overline{CS}$ cai e o barramento está com o estado de alta impedância desabilitado, os dados estão prontos para serem colocados no barramento. O comando para isso é dado por $\overline{RD}$, cuja primeira borda decrescente colocará o resultado da conversão do canal V1 no barramento de saída para ser lido. Depois de uma borda decrescente de $\overline{RD}$ os dados anteriores ainda se mantém no barramento por t_{15} = 6 ns, quando então são alterados para os novos dados, que ficam disponíveis em no máximo t_{14} = 37 ns. As bordas decrescentes subsequentes de $\overline{RD}$ farão com que as conversões dos outros canais sejam expostas. $\overline{RD}$ precisa ficar em nível baixo por t_{10} = 37 ns e em nível alto por t_{11} = 15 ns.

A queda de $\overline{CS}$ retira também o pino FRSTDATA do estado de alta impedância em no máximo t_{24} = 35 ns. O pino FRSTDATA indica quando $\overline{RD}$ baixou pela primeira vez, expressando esse fato por uma borda crescente. Depois que $\overline{RD}$ foi para 0, FRSTDATA demora no máximo t_{26} = 35 ns para subir para 1. Na segunda queda de $\overline{RD}$ FRSTDATA volta a 0 em no máximo t_{27} = 29 ns.

Depois da última borda de subida de $\overline{RD}$, $\overline{CS}$ pode subir instantaneamente, portanto, t_9 pode ser 0 ns.

Após $\overline{CS}$ subir depois da última leitura, os dados ainda continuam no barramento por t_{16} = 6 ns, para então voltar ao estado de alta impedância (*three-state* habilitado) em no máximo t_{17} = 22 ns. FRSTDATA também volta para o estado de alta impedância depois de t_{29} = 29 ns.

Ao se analisar o funcionamento do *chip* na configuração de conversão paralela, pode-se prever quais respostas o bloco controlador do AD necessita gerenciar e como coletar as respostas.

APÊNDICE C – PRODUÇÃO BIBLIOGRÁFICA

C.1. ARTIGOS EM CONGRESSOS NACIONAIS

KAPISCH, Eder Barboza; Silva, Leandro R. M.; Martins, Carlos H. N.; Filho, Luciano M. de A.; Duque, Carlos A., Implementação em FPGA de Transformada *Wavelet* para Compactação De Sinais Elétricos de Sistemas de Potência Utilizando Processador Embarcado. *Anais do XX Congresso Brasileiro de Automática (CBA)*, 2014, Belo Horizonte, Minas Gerais.

Resumo: Este trabalho tem como objetivo apresentar uma implementação da Transformada *Wavelet* em FPGA (*Field Programmable Gate Array*). Através dessa ferramenta, é possessível a compactação rápida de sinais elétricos, suavizando ou realçando os detalhes dos mesmos. Além disso, compara-se essa ferramenta embarcada em processador com uma implementação direta, denominada de método paralelo, cujos diversos blocos lógicos compõem a estrutura não-processada. Essa comparação é feita considerando-se a quantidade de blocos lógicos utilizada e a velocidade despendida em cada caso. Um protótipo desenvolvido foi capaz de implementar os dois métodos e de realizar esta tarefa de compressão, na qual mais de 87% das amostras adquiridas a uma taxa de 7,68 kHz puderam ser substituídas por zero, mantendo-se mais de 99,9% da energia do sinal original. O erro na reconstrução limitou-se à ordem de 10^{-4}%.

C.2. ARTIGOS EM CONGRESSOS INTERNACIONAIS

KAPISCH, Eder Barboza; Silva, Leandro R. M.; Martins, Carlos H. N.; Barbosa, Alexander S.; Duque, Carlos A.; Filho, Luciano M. de A.; Cerqueira, Augusto S., An Electrical Signal Disturbance Detector and Compressor Based on FPGA Platform. *Proceedings of the 16th IEEE International Conference on Harmonics and Quality of Power (ICHQP)*, 2014, Bucareste, Romênia.

Abstract: This paper presents a prototype system used to detect and compress disturbances commonly found in electrical signal, on both voltage and current. The algorithm for detecting and compressing the disturbances are synthesized in FPGA platform. The compression algorithm operates in three different stages: the innovation stage, the wavelet stage and the bit compression stage. These three stages provide efficient compression rate. Beside the algorithm

description, the paper presents details of the *hardware* implementation and results obtained from synthetized and real signals.

C.3. ARTIGOS SUBMETIDOS A PERIÓDICOS

KAPISCH, Eder Barboza; Silva, Leandro R. M.; Martins, Carlos H. N.; Barbosa, Alexander S.; Duque, Carlos A.; Filho, Luciano M. de A.; Tavil, Andres E.; de Souza, Luiz A. R., An Implementation of a Power System Smart Data Recorder using FPGA and ARM cores. *Journal of the International Measurement Confederation (IMEKO)*, 2015, Praga, República Tcheca. ISSN 0263-2241

Abstract: In this paper, the design and the prototype implementation of a power-quality (PQ) disturbance detector and compressor are described. This instrument, named Power System Smart Data Recorder (PSSDR), is able to acquire the samples of the power system signals and process them in order to detect the disturbances, compress the data and store it in a micro SD card, from which it can be reconstructed and analyzed offline with a suitable computer application. It uses, among other devices, Field Programmable Gate Array (FPGA) and ARM platforms to work with the electrical power system signals in a smart way. Algorithms of Digital Signal Processing with a high level of complexity are implemented in a low cost and small FPGA chip due the utilization of an embedded processor, developed for this purpose, which can be programmed by customized compilers that use Assembly language and an subset of the C language.

C.4. PATENTES

KAPISCH, Eder Barboza; Silva, Leandro R. M.; Martins, Carlos H. N.; Duque, Carlos A.; Filho, Luciano M. de A.; Cerqueira, Augusto S., *Registrador Eficiente De Distúrbios Elétricos*, 2014, Brasil.

Resumo: A presente invenção diz respeito à descrição de um sistema utilizado para detectar distúrbios em sinais elétricos (tanto na corrente, quanto na tensão) e compactar de maneira eficiente estes distúrbios. O sistema utiliza técnicas de detecção, compactação de distúrbio e *hardware* reconfigurável para implementar os algoritmos de processamento digital de sinais. Além disso, aplica simultaneamente três conceitos fundamentais, o que lhe confere eficiência para registrar e compactar os distúrbios de sinais elétricos em tempo real: (a) detecção

utilizando o conceito de “novidade” e limiares adaptativos; (b) Compactação do sinal em 3 níveis; (c) utilização de *hardware* reconfigurável em FPGA (*Field-Programmable Gate Array*), onde além de sintetizar máquinas de estados de dimensão finitas, são implementados processadores customizados para tarefas específicas do sistema.