

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ  
DEPARTAMENTO ACADÊMICO DE ELÉTRICA  
CURSO DE ENGENHARIA ELÉTRICA

CALLEBE SOARES BARBOSA

**IMPLEMENTAÇÃO DO ALGORITMO RADIX-2 PARA  
CÁLCULO DA FFT EM FPGA**

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO

2018

CALLEBE SOARES BARBOSA

## **IMPLEMENTAÇÃO DO ALGORITMO RADIX-2 PARA CÁLCULO DA FFT EM FPGA**

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de Trabalho de Conclusão de Curso 2, do Curso de Engenharia Elétrica da Coordenação de Engenharia Elétrica - CO-ELT - da Universidade Tecnológica Federal do Paraná - UTFPR, Câmpus Pato Branco, como requisito parcial para obtenção do título de Engenheiro Eletricista.

Orientador: Prof. Dr. Fábio Luiz Bertotti

PATO BRANCO

2018

## **TERMO DE APROVAÇÃO**

O Trabalho de Conclusão de Curso intitulado **IMPLEMENTAÇÃO DO ALGORITMO RADIX-2 PARA CÁLCULO DA FFT EM FPGA** do acadêmico **Callebe Soares Barbosa** foi considerado **APROVADO** de acordo com a ata da banca examinadora **Nº 123 de 2018.**

Fizeram parte da banca examinadora os professores:

**Prof. Dr. Fábio Luiz Bertotti**

**Prof. Dr. Giovanni Alfredo Guarneri**

**Dr. Jean Patric da Costa**



*Se vi mais longe foi por estar de pé sobre ombros  
de gigantes.*

Isaac Newton

## **AGRADECIMENTOS**

Agradeço primeiramente a Deus, por prover a mim os meios pelos quais fui, e sou sustentado todos os dias. E por me conceder a oportunidade, e a capacitação para realização deste trabalho.

Agradeço a estrutura que formou meu caráter, e me deu suporte para a realização de meus mais distantes sonhos; a minha querida família, sem o qual jamais teria chegado até aqui. Ao meu pai José Geral, a minha mãe Maria Rosa Barbosa e ao meu irmão Victor E. Soares Barbosa.

Agradeço também ao meu professor orientador Dr. Fábio Luiz Bertotti, pela paciência, dedicação e respeito com que me direcionou e ensinou ao longo da realização deste trabalho. Deixo também aqui registrado minha gratidão e reconhecimento a toda a Universidade Tecnológica Federal do Paraná (UTFPR). A esta instituição que me recebeu como aluno, e me deu a oportunidade de estudar junto a um grupo docente de excelência, e que me preparou para o nobre exercício da engenharia.

Por fim, agradeço aos meus queridos amigos, que surgiram ao longo de toda a minha jornada, e que de maneira extraordinária contribuíram para meu crescimento, celebrando com minhas vitórias e me sustentando em minha dificuldades.

## **RESUMO**

BARBOSA, Callebe Soares. Implementação do Algoritmo Radix-2 para o Cálculo da FFT em FPGA. 2018. 104 f. Monografia (Trabalho de Conclusão de Curso) - Curso de Engenharia Elétrica, Universidade Tecnológica Federal do Paraná. Pato Branco, 2018.

O presente trabalho aborda o desenvolvimento de um *hardware* dedicado ao cálculo da Transformada Rápida de Fourier (FFT), a partir da implementação do algoritmo Radix-2 em uma FPGA, conferindo paralelismo a fim aumentar a eficiência no cômputo da FFT. Para isso, são introduzidos os conceitos principais sobre a FFT, o algoritmo Radix-2, o algoritmo CORDIC, e o projeto de parâmetros que maximizam o desempenho desses algoritmos. O dispositivo escolhido para implementação é o *ZynqBerry - TE0726*, o qual é equipado com a FPGA da família Zynq-700. Neste trabalho são implementadas duas arquiteturas de FFT. A primeira FFT possui 16 pontos, e é computada com apenas 12 ciclos de *clock*, atingindo um desempenho de SQNR de 52dB. A segunda FFT possui 1024 pontos, e é computada com 1728 ciclos *clock*, atingindo um desempenho SQNR de 41dB. Ao final deste trabalho é possível compreender como é projetado e implementado uma FFT de bom desempenho, em um ambiente vantajoso como a FPGA.

**Palavras-chave:** FFT, CORDIC, RADIX-2, FPGA, ZynqBerry.

## ABSTRACT

BARBOSA, Callebe Soares. Implementation of the Radix-2 Algorithm for the Calculation of FFT in FPGA. 2018. 104 p. Undergraduate Thesis - Electrical Engineering Course, Universidade Tecnológica Federal do Paraná. Pato Branco, 2018.

The objective of this work is the development of a specific hardware for the calculation of Fast Fourier Transform (FFT), based on the implementation of the Radix-2 algorithm in FPGA, using parallelism to increase the computational efficiency. This work introduce the main concepts about FFT, the Radix-2 algorithm, the CORDIC algorithm, and the project of parameters that maximize the performance of these algorithms. The device chosen for FFT implementation is the *ZynqBerry - TE0726*, which has an FPGA of the Zynq-700 family. Two FFT architectures are implemented in this work; the first FFT has 16 points, and is computed with only 12 cycles of textit clock, achieving a SQNR performance of 52dB. The second FFT has 1024 points, and is computed with 1728 clock cycles, achieving a SQNR performance of 41dB. At the end it is possible to understand how an FFT is designed and implemented in an advantageous environment such as the FPGA.

**Keywords:** FFT, CORDIC, RADIX-2, FPGA, ZynqBerry.

## LISTA DE FIGURAS

Figura 1:	Batimento Cardíaco em 3D . . . . .	18
Figura 2:	Índice BM & FBOVESPA, São Paulo - Brasil . . . . .	18
Figura 3:	Temperatura de um Volume de Água em um Ebulidor Controlado	19
Figura 4:	Modos Normais de uma Corda Vibrante . . . . .	20
Figura 5:	Ilustração da Propriedade de Autofunção de Sistemas Lineares	22
Figura 6:	Aproximação do Sinal de Onda Quadrada por um Termo Senoidal	23
Figura 7:	Aproximação do Sinal de Onda Quadrada por Soma de 3 Termos Senoidais . . . . .	24
Figura 8:	Aproximação do Sinal de Onda Quadrada por Soma de 8 Termos Senoidais . . . . .	24
Figura 9:	Expectro de Fourier do Sinal $x(t) = \sin(2\pi 15t) + \sin(2\pi 40t)$ . . . . .	27
Figura 10:	Butterfly do Fluxo do Sinal . . . . .	32
Figura 11:	FFT de 8 Pontos . . . . .	33
Figura 12:	FFT DIF de 8 Pontos . . . . .	35
Figura 13:	Rotação de $H_r$ pelo ângulo de $2\pi r/N_0$ . . . . .	37
Figura 14:	Ângulos Elementares - Cordinic Tradicional com N=4 . . . . .	40
Figura 15:	EEAS com N=2 e S=4 . . . . .	43
Figura 16:	Exemplo de execução do Algoritmo TBS . . . . .	46
Figura 17:	Exemplo de execução do Algoritmo TBS . . . . .	47
Figura 18:	MSR com I=2, J=1 e N=2 . . . . .	50
Figura 19:	MSR com I=2, J=1 e N=2 . . . . .	51
Figura 20:	Relação entre o $P_{upper}$ e o SQNR para MSR-CORDIC, como $N_{spt} = 3$ e $N = 3$ . . . . .	54
Figura 21:	Arquitetura Tipica de uma FPGA . . . . .	57
Figura 22:	Arquitetura de uma CLB com 4 BLEs . . . . .	58
Figura 23:	Arquitetura de uma BLE ( <i>Basic Logic Element</i> ) . . . . .	59

Figura 24:	Diagrama Lógico Full Adder 4 Bits - ISE Design Suite 14	60
Figura 25:	CPU MicroBlaze e Coprocessador para FFT com Interface UART - Vivado 2017.4	63
Figura 26:	Arquitetura Simplificada - Zynq-7000	65
Figura 27:	Arquitetura Simplificada de um Sistema Digital	65
Figura 28:	Arquitetura Simplificada do um Sistema Digital Mapeado para o Zynq	66
Figura 29:	Visão Geral da Arquitetura - Zynq 7000	68
Figura 30:	Diagrama Geral do Sistema Implementado	69
Figura 31:	SoC ZynqBery - TE0726	70
Figura 32:	Relação entre $S$ e o SQNR Médio para MSR Cordic Modo Normal	75
Figura 33:	Arquitetura da Iteração MSR Cordic Modo Normal $N_{spt} = 3$	77
Figura 34:	Arquitetura Switch 2x2	77
Figura 35:	Arquitetura Implementada FFT de 16 Pontos	79
Figura 36:	Arquitetura FFT 1024 Pontos Implementada	81
Figura 37:	Sinal com componentes em 213Hz, 410Hz, e 1403Hz - FFT de 16 Pontos	85
Figura 38:	Sinal de Entrada da FFT de 16 Pontos Amostrado a 3200 Hz	86
Figura 39:	Espectro de Fourier proveniente da FFT de 16 Pontos	86
Figura 40:	Espectro de Fourier proveniente do <i>Matlab</i> para 16 Pontos	87
Figura 41:	Erro entre o resultado obtido pela FFT de 16 Pontos Implementada e o <i>Matlab</i>	87
Figura 42:	Sinal com Componentes em 213Hz, 410Hz, e 1403Hz - FFT de 1024 Pontos	90
Figura 43:	Sinal de Entrada da FFT de 1024 Pontos Amostrado a 3200 Hz	90
Figura 44:	Espectro de Fourier proveniente da FFT de 1024 Pontos	91
Figura 45:	Espectro de Fourier proveniente do <i>Matlab</i> para 1024 Pontos	91
Figura 46:	Erro entre o resultado obtido pela FFT de 1024 Pontos Implementada e o <i>Matlab</i>	92
Figura 47:	Criação de um Novo Bloco de Design	99
Figura 48:	Criação de um Novo Bloco de Design	100

Figura 49: Bloco de Design para FFT 16 Pontos . . . . .	101
Figura 50: Endereço de Memória da Interface AXI FFT . . . . .	102
Figura 51: Caminho até a ferramenta: <i>Tools: Create and Package New IP</i> .	104
Figura 52: Ferramenta de Criação e Empacotamento de Novas IPs . . . . .	104
Figura 53: Escolha da Opção: <i>Create a New Peripheral</i> . . . . .	105
Figura 54: Configurações da nova IP . . . . .	105

## **LISTA DE TABELAS**

Tabela 1:	Relação entre propriedade do tempo de um sinal e a representação de Fourier adequada . . . . .	25
Tabela 2:	Matriz $\phi$ para o dado Exemplo . . . . .	48
Tabela 3:	Nível SQNR entre o Modo Generalizado e Normal, $N_{SPT} = 3$ . .	76
Tabela 4:	Resultado de Síntese FFT 16 Pontos . . . . .	83
Tabela 5:	Comparativo de Síntese - FFT 16 Pontos . . . . .	83
Tabela 6:	Comparativo Nível de SNQR para FFT de 16 Pontos . . . . .	84
Tabela 7:	Resultado de Síntese FFT 1024 Pontos . . . . .	88
Tabela 8:	Comparativo de Síntese - FFT 1024 Pontos . . . . .	88
Tabela 9:	Comparativo Nível de SNQR para FFT de 1024 Pontos . . . . .	89

## SUMÁRIO

<b>1 INTRODUÇÃO .....</b>	<b>13</b>
1.1 OBJETIVOS .....	15
1.1.1 Objetivo Geral .....	15
1.1.2 Objetivos Específicos .....	15
1.2 ORGANIZAÇÃO DO TRABALHO .....	16
<b>2 REVISÃO DE LITERATURA .....</b>	<b>17</b>
2.1 REPRESENTAÇÃO DE FOURIER PARA SINAIS .....	19
2.1.1 Resposta do Sistemas LTI a Entrada Senoidal .....	21
2.1.2 Série de Fourier .....	25
2.1.3 Espectro de Fourier .....	27
2.2 SÉRIE DE FOURIER EM TEMPO DISCRETO .....	28
2.3 TRANSFORMADA RÁPIDA DE FOURIER .....	30
2.4 ALGORITMO CORDIC .....	35
2.4.1 CORDIC Tradicional .....	37
2.4.2 EEAS-CORDIC .....	39
2.4.2.1 Algoritmo TBS .....	43
2.4.3 MSR-CORDIC .....	48
2.4.3.1 Análise do Erro .....	52
2.5 FPGA .....	55
2.5.1 Aspectos Construtivos da FPGA .....	56
2.5.2 Programação na FPGA .....	60
2.5.2.1 Zynq-7000 .....	64
<b>3 MATERIAIS E MÉTODOS .....</b>	<b>69</b>
3.1 ZYNQBERRY TE0726-03M .....	70
3.2 IMPLEMENTANDO O PROCESSADOR CORDIC .....	71
3.2.1 Projeto dos Parâmetros Cordic .....	72
3.2.2 Arquitetura Cordic Implementada .....	76

3.3 IMPLEMENTANDO A FFT 16 PONTOS .....	78
3.4 IMPLEMENTANDO A FFT 1024 PONTOS .....	80
<b>4 RESULTADOS E DISCUSSÃO .....</b>	<b>83</b>
4.1 FFT DE 16 PONTOS .....	83
4.2 FFT DE 1024 PONTOS .....	87
<b>5 CONCLUSÃO .....</b>	<b>93</b>
<b>APÊNDICE A - PROGRAMANDO O ZYNQBERRY .....</b>	<b>99</b>
<b>APÊNDICE B - IMPLEMENTANDO A INTERFACE AXI .....</b>	<b>103</b>

## 1 INTRODUÇÃO

A Transformada Discreta de Fourier ou (DFT) (*Discrete Fourier transform*), segundo Bingham (1990), é um recurso amplamente utilizado em aplicações como processamento de imagens, comunicação em redes locais sem-fio ou WLAN (*Wireless Local Area Network*), multiplexação por divisão de frequências ortogonais ou OFDM (*Orthogonal Frequency Division Multiplexing*), e medições de diferentes espectros.

O cálculo da DFT é uma tarefa que exige muitos recursos computacionais e que requer um projeto preciso para uma implementação eficiente (WANG *et al.*, 2010). A DFT tem como base a própria série trigonométrica de Fourier discretizada. Segundo Lathi (2007, p. 719), graças a um algoritmo chamado (FFT) (*Fast Fourier transform*), desenvolvido por Cooley e Tukey (1965) o número de cálculos para executar uma DFT foi drasticamente reduzido, possibilitando um tempo de execução aceitável.

Segundo Zhou *et al.* (2009), os dispositivos conhecidos como FPGA (*Field Programmable Gate Array*) são cada vez mais usados em implementação de *hardware* para telecomunicações, por exemplo, devido a sua capacidade de alcançar um elevado desempenho, aliado a sua flexibilidade de configuração. Desta forma, o uso de FPGA para implementar um *hardware* específico para o calculo da FFT torna-se relevante.

Como afirma He e Guo (2008), o algoritmo da FFT é aplicado em larga escala como componente chave em sistemas de processamentos de sinais. De tal forma que a FFT não é somente usada em aplicações de telecomunicações e processamento de dados audiovisuais (BINGHAM, 1990), mas e o algoritmo numérico mais comum em diversas áreas, como engenharia, medicina, física e matemática (VANMATHI *et al.*, 2014).

No campo da engenharia biomédica, a FFT tem sido empregada na avaliação de parâmetros biológicos, como a bioimpedância (AMARAL *et al.*, 2011). Segundo Martinsen e Grimnes (2011), a análise da bioimpedância já é usada para monitorar atividades bioelétricas cerebrais, diagnosticar câncer de pele, dermatite, hiperidrose, avaliar a composição corporal, avaliar condição nutricional, detectar processo de rejeição de órgãos transplantado e ainda monitorar recém-nascidos através tomografia de impedância elétrica ou EIT (*Electrical Impedance Tomography*). Ainda, a EIT é a única técnica de obtenção de imagens que não afeta o sistema imunológico de

recém-nascidos (TRIANTIS *et al.*, 2011).

Tendo em vista a importância da bioimpedância na área biomédica, juntamente com a intenção de contribuir com a pesquisa nesta área dentro da Universidade Tecnológica Federal do Paraná (UTFPR), o presente trabalho propõe o desenvolvimento de uma ferramenta essencial que possa ser utilizada na aquisição de dados de bioimpedância.

Para se obter os dados de bioimpedância de um dado Objeto Sob Análise (OSA), em uma análise mais abrangente, é necessário obter a impedância na forma complexa, em determinada faixa de frequência. Isto pode ser realizado a partir de medições do módulo e da fase da tensão e da corrente no OSA, e relacionando estas grandezas. A faixa de frequências está relacionada com a aplicação e com quais fenômenos ou eventos deseja-se observar.

Uma das formas de encontrar o módulo e fase ou as partes real e imaginária da bioimpedância consiste em aplicar a DFT nos sinais digitais referentes a tensão e corrente adquiridos no OSA, os quais são obtidos, basicamente, a partir da conversão dos respectivos sinais analógicos em palavras digitais. O cálculo da DFT dependendo do número de pontos a analisar e da faixa de frequências de interesse, a demanda computacional pode ser bastante significativa, exigindo computadores com elevada capacidade de processamento e memória, inviabilizando sistemas portáteis e de baixo custo para a aquisição da bioimpedância.

Uma das formas de realizar o cálculo da DFT de forma mais eficiente consiste em dividir o cálculo em partes, sendo executados em paralelo por dispositivos de hardware específicos para esta finalidade. Isto pode ser conseguido a partir de dispositivos que permitem construir elementos de hardware de forma configurável, como as FPGAs.

A FPGA é uma boa opção para a implementação do algoritmo da FFT devido a grande variedade de recursos de *hardware* sintetizáveis, além de possuir recursos de programação paralela que permite o processamento paralelo de sinais, conferindo assim uma maior rapidez na execução do algoritmo. Portanto, implementar a FFT em uma plataforma dotada de FPGA é mais uma fator motivador para a realização deste trabalho (IBRAHIM *et al.*, 2016).

## 1.1 OBJETIVOS

### 1.1.1 OBJETIVO GERAL

Utilizando a Linguagem de Descrição de *hardware* VHSIC (VHDL), implementar um circuito lógico que, dado um conjunto de dados digitais de entrada, seja capaz de processar a Transformada Rápida de Fourier (FFT), utilizando a metodologia do algoritmo Radix-2. Buscar na implementação alcançar o melhor desempenho em termos de ciclos de *clock* necessários para que completar a tarefa de processamento da FFT, tendo como restrição os recursos disponíveis na FPGA utilizada.

### 1.1.2 OBJETIVOS ESPECÍFICOS

Ao longo deste trabalho serão buscados os seguintes objetivos específicos:

- Identificar a melhor variação do Algoritmo Cordic para a Implementação do cálculo da FFT. Tendo em vista as restrições de recursos da FPGA utilizada, encontrar o algoritmo que implemente a operação de rotação vetorial, essencial no cálculo da FFT, que reduza o erro da aproximação e maximize o desempenho da operação.
- Projetar os valores dos parâmetros Cordic que maximizem a operação de rotação vetorial. Dado a variação do Algoritmo Cordic escolhido, encontrar um método de otimização para a relação erro de aproximação x desempenho. A partir do método escolhido obter o conjunto otimizante de parâmetros para cada ângulo de rotação a ser realizado para o calculo da FFT.
- Implementar o *hardware* do Processador Cordic a ser utilizado. Com base na variação do algoritmo Cordic desenvolver em VHDL o *hardware* necessário para realizar a operação de rotação vetorial, também visando a otimização de recursos.
- Implementa os demais componentes de *hardware* necessários para a montagem do *FFT* de 16 pontos, tais como memórias RAM e ROM, Mux, Demux e o Módulo de Controle.
- Implementar a FFT de 16 pontos, e a partir de um conjunto de sinais de entrada digital realizar os devidos testes de desempenho.

- Implementar o *hardware* da FFT de 1024 pontos. A partir da FFT de 32 pontos expandir a implementação para um *hardware* com maior resolução. Realizar os devidos testes para a comparação de desempenho.

## 1.2 ORGANIZAÇÃO DO TRABALHO

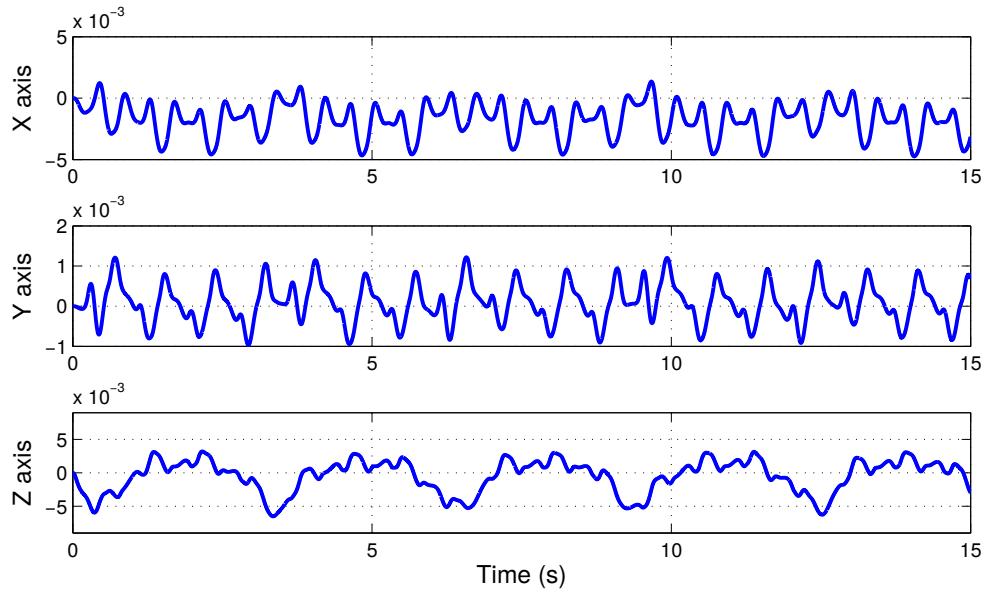
A fim de se apresentar de forma clara e organizada, o presente trabalho se encontra estruturado de forma que, no Capítulo 2 é abordado toda o embasamento teórico por traz do desenvolvimento e da implementação do trabalho. Em seguida, o Capítulo 3 trata das metodologias e procedimentos realizados para desenvolver a implementação do projeto. No Capítulo 4 são apresentados os resultados obtidos a partir da implementação e teste do projeto. E por ultimo, o Capítulo 5 trata das conclusões obtidas com o desenvolvimento deste trabalho.

## 2 REVISÃO DE LITERATURA

Nos mais diversos campos da ciência os conceitos de sinais e sistemas possuem um papel importante, sendo utilizados nas aplicações mais variadas como acústica, sismologia, projeto de circuitos, sistemas de geração e distribuição de energia, controle de processos químicos, e engenharia biomédica, entre outros. Indiferentemente da natureza física de uma aplicação, os sinais e sistemas que surgem destas possuem duas características básicas; os sinais contém informações sobre a natureza ou comportamento de um fenômeno, e os sistemas respondem a um sinal específico produzindo um comportamento desejado (OPPENHEIM; WILLSKY, 2010, Prólogo).

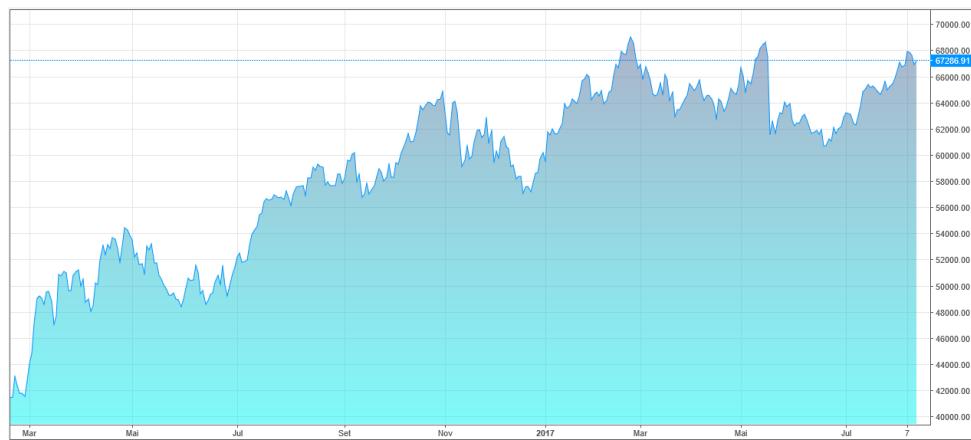
Um sinal é um conjunto de dados ou informações, que podem descrever diversos tipos de fenômenos, como o batimento cardíaco (Figura 1), a temperatura de um volume de aguá em um ebulidor controlado (Figura 3), o registro de vendas de uma empresa ou ainda os valores de fechamento de uma bolsa de valores (Figura 2). Já os sistemas são entidades que processam conjuntos de sinais (ditos entrada), e geram outro conjunto de sinais como resposta (ditos saídas). Os sistemas podem ser constituídos de componentes físicos, elétricos, mecânicos, hidráulicos ou apenas lógicos (LATHI, 2007, p. 75). O filtro de áudio, que modifica o sinal de entrada gerando um sinal de áudio com as características desejadas, é um exemplo de sistema. Assim como o ebulidor controlado que a partir de um sinal de referência (entrada) gera um sinal de controle para o atuador da resistência de aquecimento, de modo a elevar a temperatura do líquido até a referência.

Segundo Oppenheim e Willsky (2010, p. 1), existe uma linguagem adequada para descrever sinais e um conjunto poderoso de ferramentas para analisá-los, capaz de se aplicar a problemas oriundos de diversos domínios. A série de Fourier e a Transformada Rápida de Fourier são ambas exemplos destas ferramentas, e serão apresentadas nos próximos capítulos.



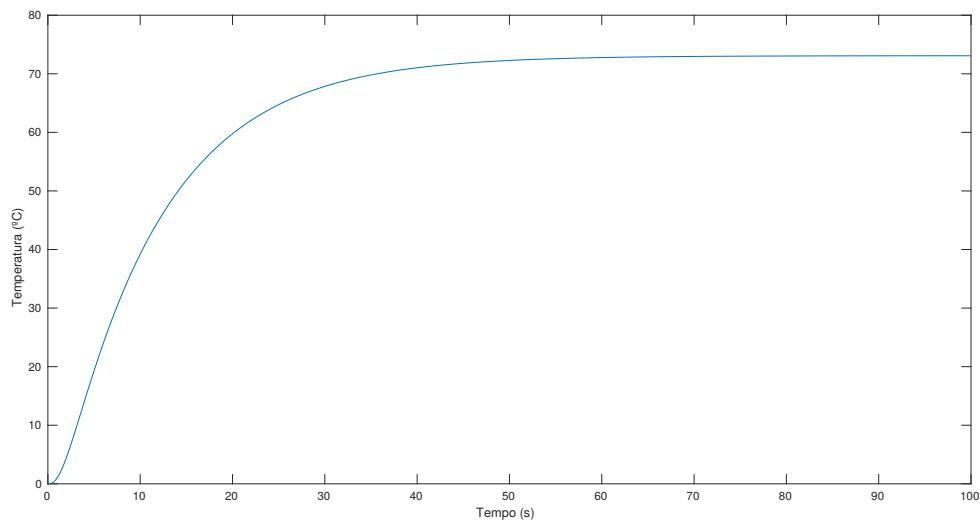
**Figura 1: Batimento Cardíaco em 3D**

Fonte: Liu *et al.* (2011)



**Figura 2: Índice BM & FBOVESPA, São Paulo - Brasil**

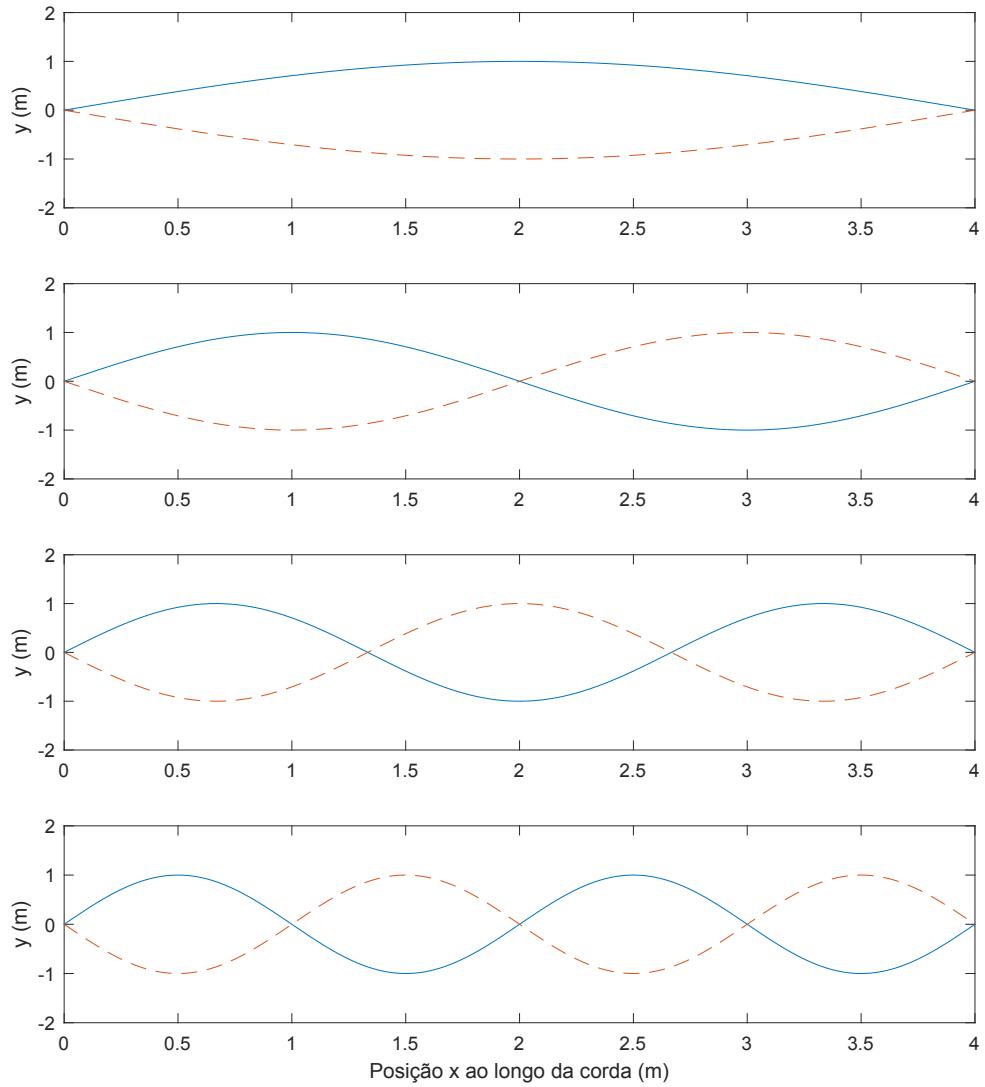
Fonte: TradingView (2017)



**Figura 3: Temperatura de um Volume de Água em um Ebulidior Controlado**  
Fonte: Autoria Própria

## 2.1 REPRESENTAÇÃO DE FOURIER PARA SINAIS

A série de Fourier tem como princípio os estudos das somas trigonométricas de senos e cossenos harmonicamente relacionados, com o intuito de descrever fenômenos periódicos. Tal estudo possui uma longa história, que data pelo menos da época dos babilônicos, e que envolve o estudo de diferentes fenômenos físicos. Mas o marco moderno neste tema ocorre em 1748 com o matemático e físico suíço Leonhard Paul Euler (OPPENHEIM; WILLSKY, 2010, p. 104). Euler em seu estudo sobre ondas estacionárias, utilizando cordas vibrantes, observou que se a configuração da posição vertical  $y_0$  de um ponto horizontal  $x$  em uma onda estacionaria no tempo  $t_0$  for uma combinação linear dos modos normais da onda, o mesmo acontece com a configuração em qualquer valor de tempo  $t_s$  subsequente, como pode ser observado na Figura (4). Com base nesse estudo, Euler demonstrou que é possível calcular diretamente os coeficientes da combinação linear em tempos futuros usando os coeficientes em tempos anteriores. (OPPENHEIM; WILLSKY, 2010, p. 104).



**Figura 4: Modos Normais de uma Corda Vibrante**  
Fonte: Adaptado de Oppenheim e Willsky (2010, p. 105)

O estudo de Euler se torna ainda mais importante quando aplicado a sinal e a sistemas (LIT) (*Linear Invariante no Tempo*). Segundo HAYKIN e Veen (2001, p. 163), se a entrada de um sistema LIT for expressa por uma combinação linear ponderada de senóides ou exponenciais complexas, a saída do sistema será expressa como uma combinação linear ponderada da resposta do sistema a cada senóide ou exponencial complexa. Expressar sinal em termos de senoides ou exponenciais complexas não apenas leva a uma expressão alternativa útil para o comportamento da entrada e saída de um sistema LTI, como também fornece uma caracterização muito criteriosa dos sinal e sistemas.

Segundo Oppenheim e Willsky (2010, p. 105), meio século depois da divulgação do trabalho de Euler, o físico e matemático francês Jean-Baptiste Joseph Fourier (1768

- 1830), havia se envolvido no estudo sobre séries trigonométricas, com a motivação física de estudar o fenômeno da propagação e difusão de calor. Fourier conclui que séries senoidais harmonicamente relacionadas eram úteis na representação da distribuição de temperatura em um corpo, e que qualquer sinal periódico poderia ser representado por tal série. Fourier ainda apresentou uma representação para sinais aperiódicos, não através de somas ponderadas de senoides harmonicamente relacionadas, mas como integrais ponderadas de senoides que não são necessariamente harmonicamente relacionadas(HAYKIN; VEEN, 2001, p. 163).

Como afirma Oppenheim e Willsky (2010, p. 106), muitas das ideias básicas por trás das contribuições de Fourier já eram conhecidas, e as condições precisas sob as quais a representação de sinais proposta era válida só foram apresentadas por P.L. Dirichlet em 1829. Porém foi Fourier que teve a clara percepção do potencial pra essa representação, e até certo ponto foi o seu trabalho e suas afirmações que estimularam grande parte do trabalho subsequente. Logo, em sua homenagem, o estudo de sinais e sistemas, usando representações senoidais, é denominado análise de Fourier. E as séries pelo qual é realizada a representação de sinais na forma de somas de senoides complexas é denominada série de Fourier.

### 2.1.1 RESPOSTA DO SISTEMAS LTI A ENTRADA SENOIDAL

Na análise de Fourier, os sinais de entrada senoidais são comumente usados para caracterizar a resposta de um sistema Linear e Invariante no Tempo (LTI). A resposta senoidal em estado estacionário de um sistema LTI é obtido pela convolução entre a entrada senoidal e o sinal de impulso(HAYKIN; VEEN, 2001, p. 163).

Ao aplicar um sinal impulso ( $\delta(t)$ ) a entrada de um sistema LTI, é gerado um sinal de saída conhecido como resposta ao impulso  $\delta(t)$ . Através da resposta ao impulso é possível caracterizar de maneira completa o comportamento de um sistema. A resposta ao impulso também possibilita conhecer a resposta do sistema LTI a qualquer sinal de entrada, através da convolução deste sinal ao impulso (HAYKIN; VEEN, 2001, p. 108).

Assim, realizando a convolução do impulso ao sinal senoidal, segundo HAYKIN e Veen (2001, p. 164), a saída de um sistema LTI dado uma entrada senoidal complexa  $x(t)$ , na forma exponencial  $e^{j\omega t}$ , é dada por:

$$y(t) = H(j\omega)e^{j\omega t}, \quad (1)$$

Em que  $H(j\omega)$  é a resposta em frequência, definida em termos de resposta ao impulso  $\delta(t)$ . Assim;

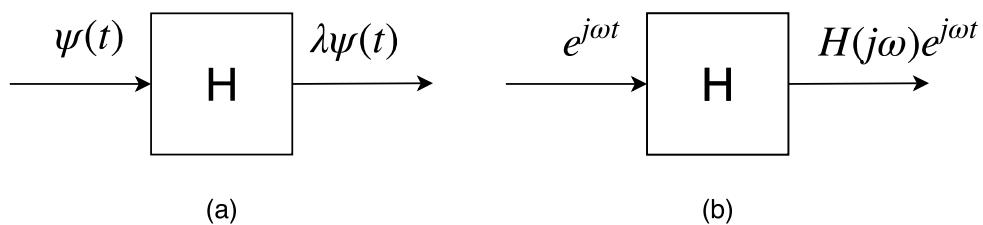
$$H(j\omega) = \int_{-\infty}^{\infty} \delta(t)e^{-j\omega t} dt. \quad (2)$$

Logo, a entrada senoidal complexa em um sistema LTI gera uma saída igual a entrada senoidal multiplicada apenas pela resposta em frequência do sistema  $H(j\omega)$ .

As equações (1) e (2) apenas consideram como entrada um sinal senoidal. Porém, é de interesse obter uma expressão para a resposta do sistema LTI a quaisquer sinais arbitrários. Para tal, HAYKIN e Veen (2001, p. 164) consideram a senoide complexa  $\psi = e^{j\omega t}$  como uma autofunção do sistema  $H$  associando com o autovalor  $\lambda = H(j\omega)$ , de modo a satisfazer:

$$H\|\psi\| = \lambda\psi(t). \quad (3)$$

Como pode ser visto na Figura (5), a saída de um sistema, dada a entrada de uma autofunção, é o produto da entrada por um número complexo. Se  $e_k$  for um autovetor de uma matriz  $A$ , e  $\lambda_k$  os autovalores associados a esta matriz, a autorrelação do problema tradicional do autovalor matricial é aplicável, se  $Ae_k = \lambda_k e_k$  (HAYKIN; VEEN, 2001, p. 164).



**Figura 5: Ilustração da Propriedade de Autofunção de Sistemas Lineares**

(a) **Autofunção geral**  $\psi(t)$  e **autovalor**  $\lambda$ .

(b) **Autofunção senoidal complexa**  $e^{j\omega t}$  e **autovalor**  $H(j\omega)$ .

**Fonte:** Adaptado de HAYKIN e Veen (2001, p. 164)

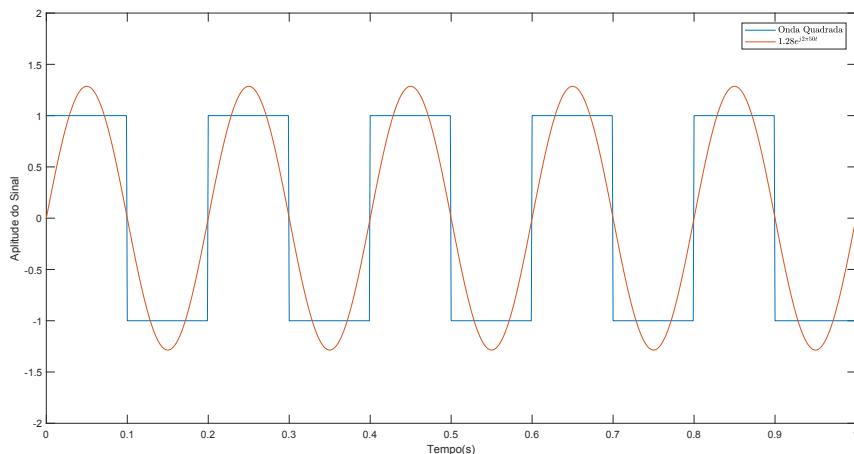
A ideia principal aqui é utilizar a superposição ponderada de autofunções para representar um único sinal periódico. Para esse efeito, HAYKIN e Veen (2001, p. 164) expressam a entrada de um sistema LTI como uma soma de  $N$  senoides

complexas ponderadas, na forma:

$$x(t) = \sum_{k=1}^N a_k e^{j\omega_k t}, \quad (4)$$

ou seja na forma de série de Fourier.

Expressar a entrada de um sistema LTI a partir de uma soma ponderada de senoides complexas possui a intenção de realizar uma aproximação coerente do sinal de entrada, utilizando uma composição de funções básicas já bem conhecidas. Por exemplo, considere o sinal de onda quadrada presente na Figura (6), o qual deseja-se aproximar utilizando uma soma de senoides complexas. Tomando uma senoide de amplitude 1.286 e uma frequência de  $50Hz$  é possível realizar uma aproximação groseira, porém em fase com este sinal.

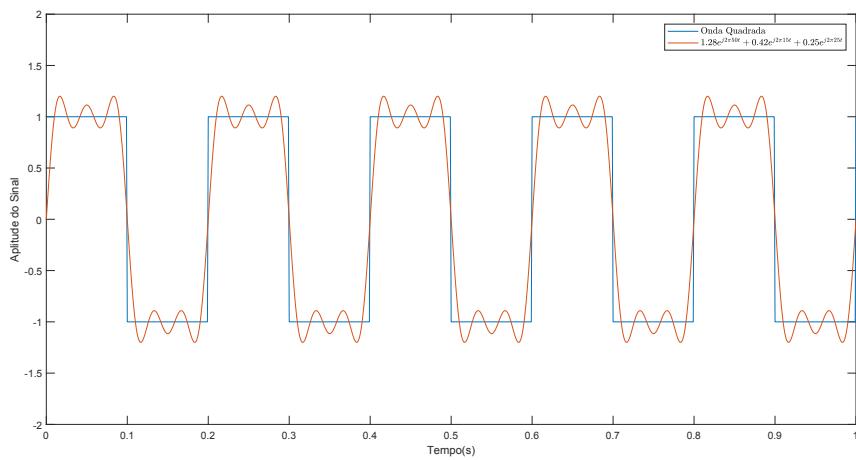


**Figura 6: Aproximação do Sinal de Onda Quadrada por um Termo Senoidal**  
Fonte: Autoria Própria

Para melhorar a aproximação da representação em relação ao sinal de onda quadrada é necessário adicionado mais termos senoidais ao somatório. Como pode ser visto nas Figura (7) e (8) as aproximações ficam mais precisas ao adicionar mais 2 ou mais 5 termos, respectivamente. Quanto maior o número de termos senoidais complexas ponderados presentes no somatório da representação maior é a aproximação, sendo no limite perfeita. Logo, se  $e^{j\omega_k t}$  for uma autofunção e  $H(j\omega_k)$  for o autovalor do sistema, aplicando-se a autorrelação apresentada anteriormente, o sinal de saída do sistema pode ser expresso como:

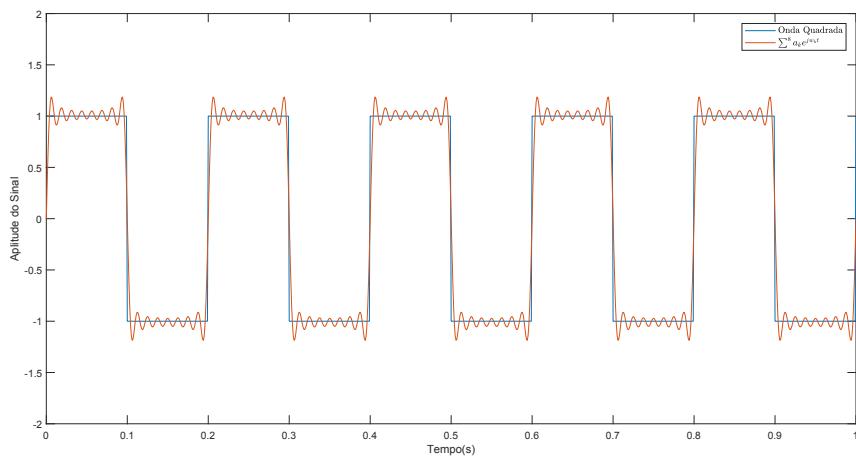
$$y(t) = \sum_{k=1}^N a_k H(j\omega_k) e^{j\omega_k t}, \quad (5)$$

que nada mais é do que a soma ponderada das senoides complexas da entrada, sendo os pesos  $a_k$  ponderados pela resposta em frequência  $H(j\omega_k)$ . Por meio destes resultados, é possível transformar a operação de convolução em uma operação de multiplicação dos termos  $a_k H(j\omega_k)$ . Segundo Oppenheim e Willsky (2010, p. 164), foi Euler que descobriu que, se a entrada de um sistema LTI for dado como combinação linear de senoides complexas, a saída deste sistema também será combinação de exponenciais complexas, relacionadas com a entrada. Esta descoberta acabou por motivar Fourier e os outros matemáticos após ele no estudo da extensão de classes de sinais que poderiam ser representados na forma de somatórios de exponenciais complexas ponderadas.



**Figura 7: Aproximação do Sinal de Onda Quadrada por Soma de 3 Termos Senoidais**

**Fonte:** Autoria Própria



**Figura 8: Aproximação do Sinal de Onda Quadrada por Soma de 8 Termos Senoidais**

**Fonte:** Autoria Própria

Além de tornar mais prático o cálculo da convolução de sinais, a representação em somais senoidais complexas ponderadas fornece uma interpretação alternativa para sinais e sistemas HAYKIN e Veen (2001, p. 166). Por meio da análise dos pesos  $a_k$  ponderados, é possível descrever um sinal em função da frequência, ao invés do tempo.

A representação de sinais por séries de Fourier pode ser aplicada para diferentes tipos de sinais, com diferentes características. Há quatro classes de representações de Fourier, divididas de acordo com a sua periodicidade e sua continuidade, como poder ser visto na Tabela (1).

Para sinais periódicos a representação é feita como séries, sendo que para sinais de tempo contínuo é aplicado a Série de Fourier (FS), e para sinais de tempo discreto é usada as séries de Fourier de tempo discreto (DFS).

Quando os sinais não são periódicos a representação é denominada como transformada, para o caso do sinal ser contínuo, a representação é feita pela transformada de Fourier (FT), e no caso discreto é a transformada de Fourier de tempo discreto (DFT).

Propriedade do Tempo	Periódico	Não Periódico
Continuo	Série de Fourier (FS)	Transformada de Fourier (FT)
Discreto	Série de Fourier de Tempo Discreto (DFS)	Transformada de Fourier de Tempo Discreto (DFT)

**Tabela 1: Relação entre propriedade do tempo de um sinal e a representação de Fourier adequada**

Fonte: Adaptado de Oppenheim e Willsky (2010, p. 166)

A representação de Fourier utilizada no desenvolvimento deste trabalho é a DFT, sendo a mais importante a ser abordada. Tendo em vista que, neste trabalho, a entrada ao qual se deseja aplicar a transformada de Fourier é um conjunto de dados binários, que podem representar sinais não periódicos, amostrados e quantizados por um Conversor Analógico-Digital (ADC).

### 2.1.2 SÉRIE DE FOURIER

Segundo Lathi (2007, p. 530), "Um sinal periódico  $x(t)$  com período  $T_0$  pode ser descrito como a soma de senoides de frequência  $f_0$  e todas as suas harmônicas", conforme apresentado na seguinte expressão:

$$x(t) = a_0 + \sum_{n=1}^{\infty} a_n \cos(n\omega_0 t) + b_n \sin(n\omega_0 t) \quad (6)$$

Esta é a chamada série de Fourier para sinais periódicos, ou apenas série de Fourier. Na expressão da Equação (6), a série de Fourier está na forma trigonométrica, onde  $\omega_0$  é a frequência fundamental de  $x(t)$ , e  $a_0$ ,  $a_n$  e  $b_n$  são os coeficientes de amplitude das harmônicas que compõe  $x(t)$ , sendo  $a_0$  a harmônica zero (nível CC).

Os coeficientes  $a_0$ ,  $a_n$  e  $b_n$  de (6) são determinados pelas seguintes equações:

$$a_0 = \frac{1}{T_0} \int_{T_0} x(t) dt, \quad (7)$$

$$a_n = \frac{2}{T_0} \int_{T_0} x(t) \cos(n\omega_0 t) dt, \quad (8)$$

$$b_n = \frac{2}{T_0} \int_{T_0} x(t) \sin(n\omega_0 t) dt, \quad (9)$$

em que  $T_0$  representa o período relativo a frequência fundamental  $f_0$ .

A série de Fourier, além da forma trigonométrica, também pode ser apresentada na forma exponencial, em termos de  $e^{j\omega_0 t}$ , como apresentado na seção anterior. A forma exponencial, segundo Lathi (2007, p. 533), é dada pela expressão:

$$x(t) = \sum_{-\infty}^{\infty} C_n e^{jn\omega_0 t}, \quad (10)$$

em que o coeficiente  $C_n$  é análogo aos coeficientes  $a_n$  e  $b_n$  da série trigonométrica, sendo obtido por:

$$C_n = \frac{1}{T_0} \int_{T_0} x(t) e^{jn\omega_0 t} dt. \quad (11)$$

Tanto a forma trigonométrica quanto a exponencial da série de Fourier consideram  $x(t)$  como sendo uma função qualquer, real ou complexa. Entretanto na maioria das aplicações  $x(t)$  é real, como é o caso dos sinais neste trabalho. Segundo Lathi (2007, p. 533), se o sinal de entrada do sistema LTI  $x(t)$  é real, isso significa que  $a_n$  e  $b_n$  também são reais para todos os valores de  $n$ , assim a série de Fourier pode ser representada na forma compacta por:

$$x(t) = D_0 + \sum_{n=1}^{\infty} D_n \cos(n\omega_0 t + \theta_n), \quad (12)$$

sendo:

$$D_0 = a_0, \quad (13)$$

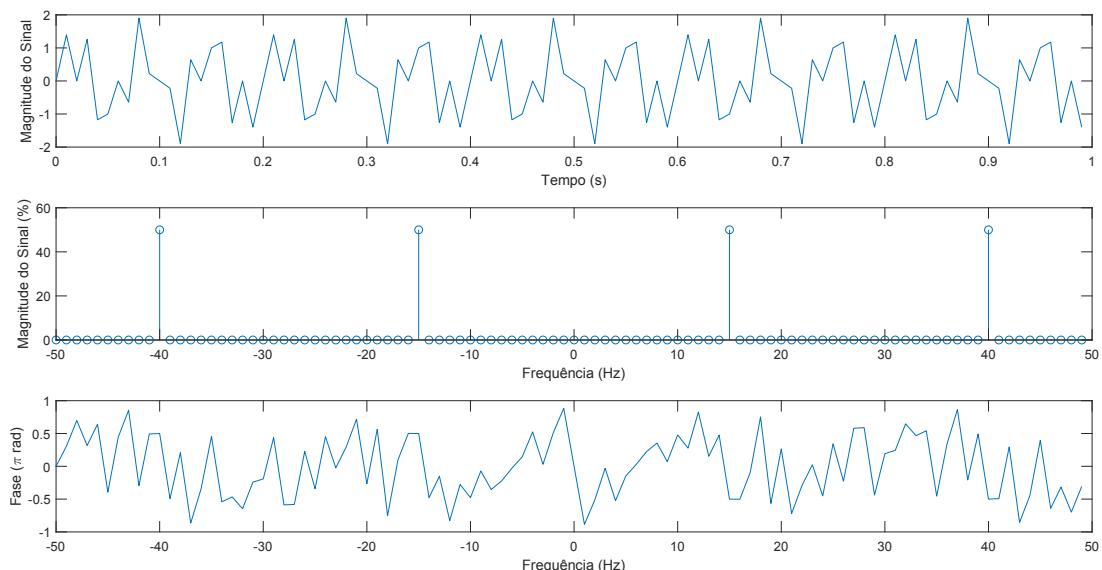
$$D_n = \sqrt{a_n^2 + b_n^2}, \quad (14)$$

$$\theta_n = \tan^{-1} \frac{-b_n}{a_n}. \quad (15)$$

### 2.1.3 ESPECTRO DE FOURIER

Por meio da série de Fourier na forma compacta, conforme (12), conclui-se que um sinal real periódico  $x(t)$  pode ser descrito como uma soma de senoides de frequências  $n\omega_0$  e amplitudes  $D_n$  e fases  $\theta_n$ . Segundo Lathi (2007, p. 533), o espectro exponencial de Fourier é traçado a partir de  $D_n$  e  $\theta_n$  em função das frequências  $n\omega_0$ . Portanto, são traçados dois gráficos para o espectro exponencial de Fourier, um que relaciona  $D_n$  com  $n\omega_0$ , chamado espectro de magnitude, e outro que relaciona  $\theta_n$  com  $n\omega_0$ , chamado de espectro de fase.

Tomando como exemplo um sinal  $x(t) = \sin(2\pi 15t) + \sin(2\pi 40t)$ , expresso em um período igual a 1 segundo, como mostrado na Figura 9. Para este sinal é expresso o espectro de Fourier na mesma Figura (9), com o espectro de magnitude e fase.



**Figura 9: Expectro de Fourier do Sinal**  $x(t) = \sin(2\pi 15t) + \sin(2\pi 40t)$   
**Fonte:** Autoria Própria

Nota-se que o espectro da Figura (9) aparecem frequências negativas, as quais dividem a magnitude com suas frequências simétricas. Isso ocorre devido a

simetria que o ângulo  $n\omega_0 t$  possui no cálculo dos coeficientes da série de Fourier. Para resolver este problema, basta considerar apenas a parte positiva do espectro e multiplicar por 2 a magnitude das frequências no espectro de magnitude.

Para Lathi (2007, p. 533), os dois gráficos de magnitude e fase juntos formam o espectro de frequência, o qual revela os conteúdos de frequência do sinal  $x(t)$ , com suas amplitudes e fase. Conhecendo-se este espectro, não só é possível analisar o sinal  $x(t)$  dentro do domínio da frequência, como também reconstruí-lo de forma fácil.

## 2.2 SÉRIE DE FOURIER EM TEMPO DISCRETO

Ate aqui foi apresentada a forma continua da série de Fourier, porém para ser útil em uma aplicação computacional é necessário encontrar sua forma discreta, ou DFT. Segundo HAYKIN e Veen (2001, p. 314), a DFT é a única representação de Fourier que pode ser calculada por um computador, sendo amplamente usada para manipular sinais.

O primeiro passo para se obter uma DFT é considerar o teorema da Amostragem. Tal teorema afirma que um sinal real  $x(t)$ , cujo o espectro é limitado em  $\phi$  Hz, pode ser reconstruído a partir de suas amostras tomadas uniformemente a uma taxa  $f_s > 2\phi$  (LATHI, 2007, p. 679). Em seguida, a amostragem de  $x(t)$ , feita a uma frequência  $f_s$ , pode ser obtida pela multiplicação de  $x(t)$  por um trem de impulsos  $\delta(t)$ . Sendo tais impulsos unitários e periódicos, repetidos a cada  $T = 1/f_s$  segundos, por um numero total de amostras  $N_0$ , a amostragem pode ser definida por:

$$\bar{x}(t) = x(t)\delta_T(t) = \sum_{n=0}^{N_0-1} x(nT)\delta(t - nT) \quad (16)$$

Por conveniência, deseja-se obter um espectro do sinal amostrado  $x(t)$  em função de  $\omega$  ou expresso em termos de frequência. Para tal, segundo Lathi (2007, p. 681), o trem de impulsos  $\delta(t)$  é um sinal periódico que pode ser descrito pela série trigonométrica de Fourier da seguinte forma:

$$\delta_T(t) = \frac{1}{T}[1 + 2\cos(\omega_s t) + 2\cos(2\omega_s t) + 2\cos(3\omega_s t) + \dots]. \quad (17)$$

Logo, multiplicando  $x(t)$  por  $\delta_T(t)$ , obtém-se:

$$\bar{x}(t) = x(t)\delta_T(t) = \frac{1}{T}[x(t) + 2x(t)\cos(\omega_s t) + 2x(t)\cos(2\omega_s t) + 2x(t)\cos(3\omega_s t) + \dots] \quad (18)$$

Segundo Lathi (2007, p. 679), a transformada de Fourier do primeiro termo  $x(t)$ , em (18), é  $X(\omega)$ . Já a transformada de Fourier do segundo termo  $2x(t)\cos(\omega_s t)$  é  $X(\omega - \omega_s) + X(\omega + \omega_s)$ , e do terceiro termo  $2x(t)\cos(2\omega_s t)$  é  $X(\omega - 2\omega_s) + X(\omega + 2\omega_s)$ . E assim, semelhantemente a transformada de Fourier dos demais termos da série que descreve (18), representam o espectro  $X(\omega)$  deslocado em  $n\omega_s$  e  $-n\omega_s$ . Assim,

$$\bar{X}(\omega) = \frac{1}{T} \sum_{-\infty}^{\infty} X(\omega - n\omega_s) \quad (19)$$

Desde que a frequência de amostragem  $f_s$  garanta o critério do teorema da Amostragem, o sinal  $\bar{X}$  será constituído de repetições não sobrepostas de  $x(\omega_0)$ , a um intervalo de tempo  $T = 1/f_s$ . Tanto  $\bar{X}(\omega)$ , quanto  $\bar{x}(t)$  são periódicas e equivalentes, porém com representações distintas do espectro amostrado Oppenheim e Willsky (2010, p. 125). Sendo assim, usando a propriedade de deslocamento no tempo (20) e a expressão da amostragem (16), obtém-se:

$$\delta(t - nT) \longleftrightarrow e^{-jn\omega T}, \quad (20)$$

$$\bar{x}(t) = \sum_{n=0}^{N_0-1} x(nT)e^{-jn\omega T}. \quad (21)$$

Segundo Lathi (2007, p. 705), a transformada de  $\bar{x}(t)$  pode ser aproximada, considerando um certo *aliasing* negligenciável, para  $X(\omega)/T$ . Portanto:

$$X(\omega) = T \sum_{n=0}^{N_0-1} x(nT)e^{jn\omega T} \quad |\omega| \leq \frac{\omega_s}{2} \quad (22)$$

Analizando a propriedade periódica de  $x(t)$  e  $X(\omega)$ , e considerando  $x(nT)$  e  $X(r\omega_0)$  a  $n$ -ésima e  $r$ -ésima amostra de  $x(t)$  e  $X(\omega)$ , respectivamente, são definidas as seguintes variáveis:

$$x_n = Tx(nT), \quad (23)$$

$$x_n = \frac{T_0}{N_0} x(nT), \quad (24)$$

$$X_r = X(\omega), \quad (25)$$

$$\omega = r\omega_0, \quad (26)$$

$$X_r = X(r\omega_0), \quad (27)$$

$$\omega_0 = 2\pi f_0 = \frac{2\pi}{T_0}. \quad (28)$$

Assim, substituindo (27) e (24) em (22), e fazendo  $\omega_0 T = \Omega_0 = 2\pi/N_0$ , se obtém a seguinte expressão para a transformada discreta de Fourier (OPPENHEIM; WILLSKY, 2010, p. 125):

$$X_r = \sum_{n=0}^{N_0-1} x_n e^{j\omega_0 nr}, \quad (29)$$

onde:

$$\Omega_0 = \frac{2\pi}{N_0}. \quad (30)$$

Para compactar a expressão de (29), se faz a substituição da expressão exponencial pela variável  $W$ , de modo que  $W_{N_0} = e^{-2\pi/N_0} = e^{-j\Omega_0}$  (MEYER-BAESE, 2007, p. 344). Logo a expressão para DFT é dada por:

$$X_r = \sum_{n=0}^{N_0-1} x_n e^{j\omega_0 nr}, \quad (31)$$

onde:

$$0 \leq k \leq N_0 - 1. \quad (32)$$

## 2.3 TRANSFORMADA RÁPIDA DE FOURIER

Para se calcular uma DFT de  $N_0$  valores usando apenas (31), é necessário realizar um total de  $N_0^2$  multiplicações e  $N_0(N_0 - 1)$  somas, utilizando números complexos. Deste modo, quando  $N_0$  assume um valor elevado, muitos recursos computacionais são necessários, até chegar ao ponto de que esse algoritmo se torna impraticável.

A redução do numero de operações matemáticas necessárias para calcular

a DFT é possível a partir do algoritmo criado por J.W. Cooley e John Tukey, conhecido como Transformada Rápida de Fourier ou FFT (*Fast Fourier Transform*) Lathi (2007, p. 719). Para reduzir o numero de cálculos, a FFT se utiliza da propriedade linear da transformada de Fourier. Segundo Oppenheim e Willsky (2010, p. 119), a transformada de Fourier de um sinal pode ser dada pela combinação linear da transformada de Fourier de segmentos menores do mesmo sinal. Logo, é possível aplicar a DFT o paradigma da Divisão e Conquista, o qual é um recurso muito utilizado em algoritmos de ordenação.

Segundo Cormen *et al.* (2002, p. 21), um algoritmo de Divisão e Conquista realiza o desmembramento de um problema em vários subproblemas que são idênticos ao original, porém menores em sua faixa de ação, o que os torna mais simples de resolver. Em seguida, resolvem-se os subproblemas recursivamente e combinam-se essas soluções de modo a obter a solução para o problema original.

De modo muito semelhante, o algoritmo da FFT prevê uma divisão recursiva da DFT em dois blocos: bloco par e o bloco ímpar, como mostrado abaixo(CHU; GEORGE, 1999, p. 35):

$$X_r = \underbrace{\sum_{n=0}^{\frac{N_0}{2}-1} x_{2n} W_{N_0}^{2nr}}_{\text{Parcela Par}} + \underbrace{\sum_{n=0}^{\frac{N_0}{2}-1} x_{2n+1} W_{N_0}^{(2n+1)r}}_{\text{Parcela Ímpar}}. \quad (33)$$

Nesta mesma equação os limites dos somatórios de ambas as parcelas ímpar e par foram redefinidas para englobar apenas metade dos  $N_0$  pontos, bem como os expoentes de  $W$  foram ajustados.

Utilizando algumas das propriedades geométricas de  $W$ , já que este representa um numero complexo, pode-se realizar simplificações importantes em 34. Primeiro, nota-se que  $W_{N_0/2} = W_{N_0}^2$ , logo:

$$X_r = \underbrace{\sum_{n=0}^{\frac{N_0}{2}-1} x_{2n} W_{N_0}^{2nr}}_{G_r} + \underbrace{\sum_{n=0}^{\frac{N_0}{2}-1} x_{2n+1} W_{N_0}^{(2n+1)r}}_{H_r}. \quad (34)$$

Como  $G_r$  e  $H_r$  são DFTs com  $N_0/2$  pontos cada, então ambos possuem um período de  $N_0/2$ . Com base na propriedade periódica destas DFTs, pode-se utilizar as simplificações (35) e (36) para reduzir o número de cálculos na DFT (LATHI, 2007, p. 721).

$$G_{r+\frac{N_0}{2}} = G_r, \quad (35)$$

$$H_{r+\frac{N_0}{2}} = H_r, \quad (36)$$

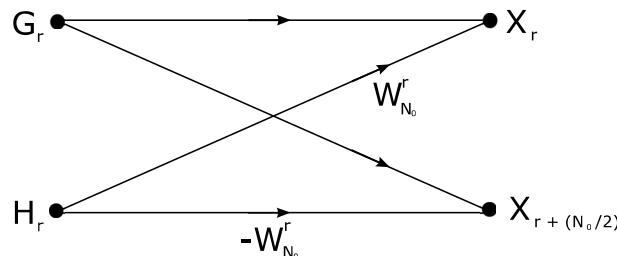
$$W_{N_0}^{r+\frac{N_0}{2}} = W_{N_0}^{\frac{N_0}{2}} = e^{-j\pi} W_{N_0} = -W_{N_0}^r. \quad (37)$$

Além disso, a expressão em (37) pode ser assumida para se reduzir o número de cálculos da FFT. Portanto, usando (38) e (39) se obtém, respectivamente, os primeiros  $N_0/2$  pontos e os últimos  $N_0/2$  pontos da FFT:

$$W_{N_0}^{r+\frac{N_0}{2}} = W_{N_0}^{\frac{N_0}{2}} = e^{-j\pi} W_{N_0} = -W_{N_0}^r, \quad (38)$$

$$W_{N_0}^{r+\frac{N_0}{2}} = W_{N_0}^{\frac{N_0}{2}} = e^{-j\pi} W_{N_0} = -W_{N_0}^r, \quad (39)$$

Portanto, uma DFT pode ser calculada combinando duas DFTs de  $N_0/2$ , tal como mostrado em (38) e (39). É comum na literatura representar este processo de cálculo de DFT feito pelo algoritmo da FFT pelo diagrama da Figura (10). Este diagrama é conhecido como *Butterfly* de Fluxo do Sinal (CHU; GEORGE, 1999, p. 36).



**Figura 10: Butterfly do Fluxo do Sinal**  
Fonte: Lathi (2007, p. 721)

Aliando o conceito de divisão em conquista ao método de cálculo da DFT, usando o diagrama *Butterfly*, a representação do algoritmo da FFT pode ser facilmente representado pelo diagrama da Figura (11) (LATHI, 2007, p. 722). Nesta figura, a FFT é feita para apenas 8 amostras de sinal  $X$ .

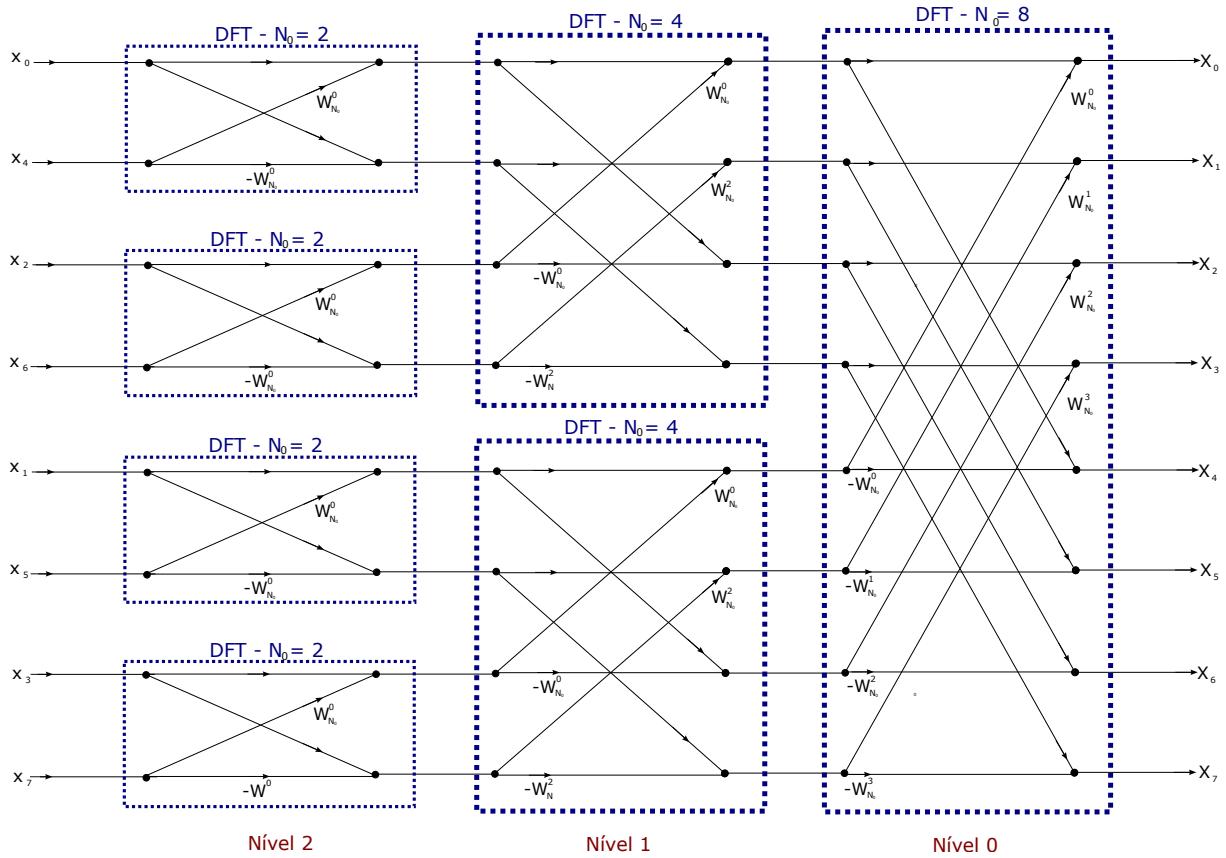


Figura 11: FFT de 8 Pontos

Fonte: Lathi (2007, p. 722)

Após se dividir uma DFT de tamanho  $N_0$  em duas DTFs de tamanho  $N_0/2$ , e subdividindo cada uma das DFTs de tamanho  $N_0/2$  em duas  $N_0/4$  (LATHI, 2007, p. 721). E assim o procedimento continua até que se atinja um nível em que as DFTs tenham tamanho  $N_0/2^n = 2$ , ou seja, quando se atinge DFTs que possuem um custo de cálculo mínimo.

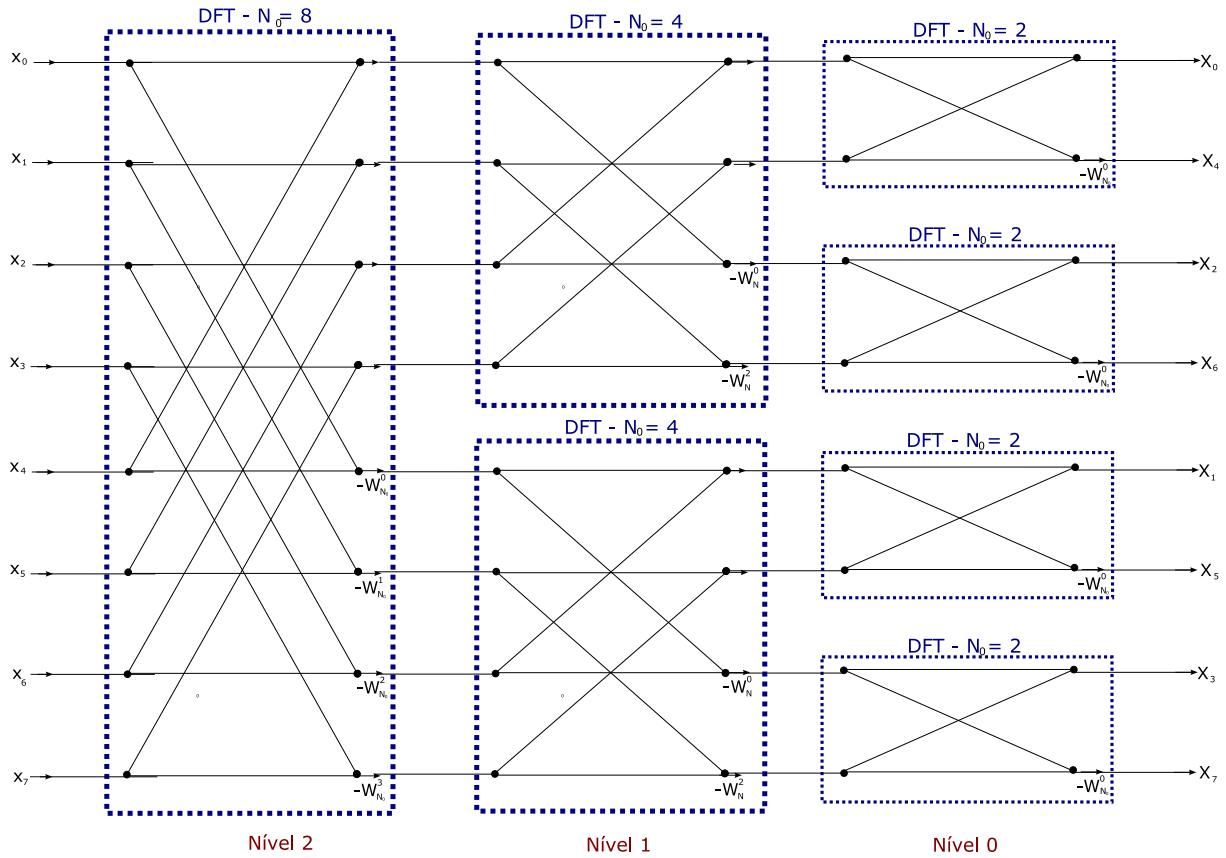
Um fato importante sobre o algoritmo da FFT é que o valor  $N_0$  pode ser escolhido segundo a relação  $N_0 = r^n$ , onde  $n$  é o numero de níveis necessários para calcular a FFT, e  $r$  é o mínimo tamanho DFT. Os algoritmos de FFT mais usados possuem a base  $r$  igual a 2 ou a 4. Consequentemente, estes algoritmos são conhecidos como Radix-2 e Radix-4, respectivamente (MEYER-BAESE, 2007, p. 365). Neste trabalho o foco será o algoritmo com a base  $r$  igual a 2, ou seja, o Radix-2.

Na Figura (11) as DFTs estão agrupadas por níveis, onde cada nível abrange as DFTs de mesmo tamanho  $N_0/n$ , e no último nível a esquerda há quatro DFTs de tamanho 2. O número de níveis necessários em uma FFT de  $N_0$  pontos é  $\log_2 N_0$ . Os valores de  $X$ , na Figura (11) à direita estão ordenados de forma crescente, porém os valores de  $x$  à esquerda estão ordenados de forma diferente. Esta ordenação é

conhecida como *Bit-Reverse* (CHU; GEORGE, 1999, p. 51).

Segundo Chu e George (1999, p. 51), quando se divide o processo de cálculo de uma DFT em duas, sendo uma responsável pelos valores pares e a outra pelo valores ímpares, conforme as conexões entre os níveis vão ocorrendo, há permutações entre os sinais. O processo de *Bit-Reverse* prevê estas permutações, sendo possível saber qual a ordem adequada dos sinais na entrada da FFT. Para aplicar o conceito de *Bit-Reverse*, basta considerar um elemento  $x_k$  de ordem  $n$  e escrevê-lo na base binária com  $\log_2 N_0$  bits. Em seguida, para determinar onde  $x_k$  será ocupado, basta converter novamente para base decimal o número binário obtido, lendo esse na ordem inversa dos bits.

Na FFT da Figura (11) a subdivisão das DFTs é feita a partir da saída, sinal em frequência, com DFT de 8 pontos, até a entrada, sinal no tempo, com as DFTs de 2 pontos com custo mínimo. Essa subdivisão é conhecida como decimação, e no caso da Figura (11), ela é feita a partir do sinal no tempo, logo essa estrutura é conhecida como Decimação no Tempo. Porém, existe uma outra forma, a decimação na frequência. A partir de uma alteração em (34), é possível alterar a ordem da decimação, partindo do sinal de entrada no tempo, até a saída em frequência, como pode ser visto na Figura(12) (CHU; GEORGE, 1999, p. 37).



**Figura 12: FFT DIF de 8 Pontos**  
Fonte: Adaptado de Chu e George (1999, p. 44)

Na decimação em frequência, a ordem de entrada dos sinais na FFT ficar ordenados de forma crescente, porém a saída passa a estar ordenado no esquema *Bit-Reverse*. Este esquema de decimação é preferencialmente escolhido por dispensar a reordenação dos dados de entrada.

Ao final de todo o processo de simplificação do cálculo da DFT, o algoritmo da FFT necessita apenas realizar  $(N_0/2) \log_2 N_0$  multiplicações e  $N_0 \log_2 N_0$  somas complexas (LATHI, 2007, p. 720). Desta forma, reduz-se assim a complexibilidade do algoritmo, tornando a DFT praticável até para valores elevados de  $N_0$ .

## 2.4 ALGORITMO CORDIC

Na equação (38) nota-se que o cálculo a FFT depende essencialmente de um multiplicação complexa entre  $W_{N_0}^r$  e  $H_r$ , onde  $H_r$  representa um vetor complexo qualquer, e  $W_{N_0}^r$  é igual a  $e^{\frac{2\pi r}{N_0}}$ . Esta tarefa pode ser realizada através da representação de  $W_{N_0}^r$  e  $H_r$  na forma retangular, considerando para efeitos de exemplo  $W_{N_0}^r = x + iy$  e  $H_r = a + ib$ , logo:

$$W_{N_0}^r = x + iy, \quad (40)$$

$$H_r = a + ib, \quad (41)$$

$$H_r \cdot W_{N_0}^r = (x + iy)(a + ib), \quad (42)$$

$$H_r \cdot W_{N_0}^r = (xa - yb) + i(xb + ya). \quad (43)$$

Desta forma, serão necessários utilizar 4 multiplicadores e 2 somadores, para completar esta operação (DESPAIN, 1974, p. 1). Apesar disso, em termos de complexidade hardware, multiplicadores são mais elaborados e, em muitos dispositivos FPGA, possuem apenas algumas dezenas, o que limita a utilização de multiplicadores em paralelo, reduzindo a velocidade de cálculo da FFT.

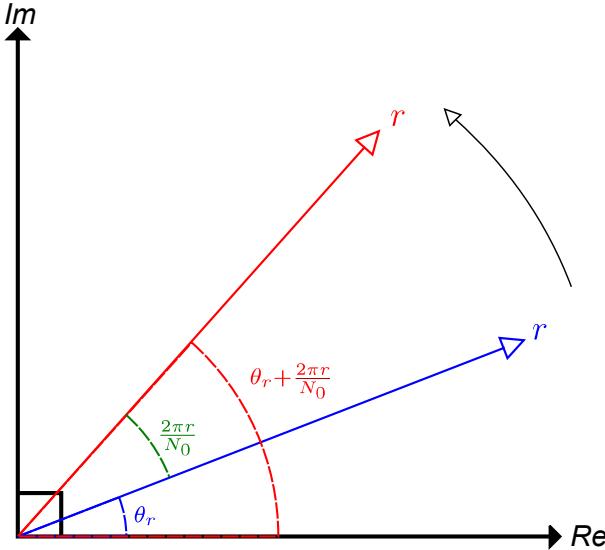
Para contornar o problema, considere expressar o vetor  $H_r$  na forma polar  $re^{\theta_r}$  e manter  $W_{N_0}^r$  igual a  $e^{\frac{2\pi r}{N_0}}$ , logo:

$$H_r = Re^{(\theta_r)}, \quad (44)$$

$$H_r \cdot W_{N_0}^r = re^{(\theta_r)} \cdot e^{\left(\frac{2\pi r}{N_0}\right)}, \quad (45)$$

$$H_r \cdot W_{N_0}^r = re^{\left(\theta_r + \frac{2\pi r}{N_0}\right)}. \quad (46)$$

Assim  $H_r \cdot W_{N_0}^r$  é equivalente a operação trigonométrica de rotacionar o vetor complexo  $H_r$  pelo ângulo de  $2\pi r/N_0$ , como pode ser visto na Figura (13).



**Figura 13: Rotação de  $H_r$  pelo ângulo de  $2\pi r/N_0$**

**Fonte:** Autoria Própria

Implementar operações trigonométricas sem utilizar multiplicadores, para evitar o gargalo que eles possam provocar, parece difícil, porém existe uma técnica bastante usada no desenvolvimento de circuitos lógicos em FPGA que o possibilita. Com o objetivo de oferecer uma solução para o cálculo de funções trigonométricas mais simples, utilizando o mínimo de recursos de tempo e *hardware*, Jack E. Volder(VOLDER, 1959) desenvolveu a técnica de computação trigonométrica CORDIC (*COordinate Rotation Digital Computer*).

#### 2.4.1 CORDIC TRADICIONAL

O algoritmo CORDIC tem como base as micro-rotações do vetor alvo, de tal modo que cada micro-rotação possa ser feita por somas e deslocamentos de bits. Para tal, considere o vetor  $H_r = x + iy$ , e a matriz de rotação  $A$  em função do angulo  $\theta$ . A rotação do vetor pode ser dada por (GARRIDO *et al.*, 2016):

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \quad (47)$$

Isolando o termo  $\cos(\theta)$  de (47) é obtido o seguinte sistema:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \cos(\theta) \begin{bmatrix} 1 & -\tan(\theta) \\ \tan(\theta) & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}, \quad (48)$$

o qual é a base da técnica CORDIC convencional (EL-MOTAZ *et al.*, 2014).

Segundo (VORDER, 1959), ao invés de rotacionar completamente o vetor  $H_r$  pelo ângulo  $\theta$ , ou pelo ângulo  $2\pi r/N_0$  como mostrado na Figura (13), o algoritmo CORDIC rotaciona  $H_r$  por ângulos  $\theta_n$  muito menores, sendo estes frações de  $\theta$ , de tal forma que:

$$\theta = \sum_{n=0}^{N-1} \mu_n \theta_n + \zeta, \quad (49)$$

onde  $\mu_n$  representa o sentido da micro rotação  $\theta_n$ , sendo que se for no sentido horário é igual a 1, caso contrário é igual a -1, e  $\zeta$  é o erro acumulado da aproximação pelo somatório, que será considerado suficientemente pequeno para ser ignorado.

O primeiro ponto dessa abordagem é o fato de que se um angulo  $\theta_n$  for suficientemente pequeno é possível afirmar que:

$$\theta_n \simeq \tan^{-1} \theta_n. \quad (50)$$

Assim é possível substituir  $\theta$  no sistema de (48), e obter a seguinte expressão:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \cos(\tan^{-1} \theta_n) \begin{bmatrix} 1 & -\tan(\tan^{-1} \theta_n) \\ \tan(\tan^{-1} \theta_n) & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \quad (51)$$

O segundo ponto é que para satisfazer (49) só é necessário que a soma do conjunto de micro rotações  $\theta_n$  resulte no ângulo  $\theta$ , o que abre a possibilidade de se escolher um conjunto de micro rotações baseadas em deslocamento de bits, como:

$$\theta_n = \tan^{-1} 2^{-n} \quad (52)$$

Logo, aplicando a Equação(52) em (51), é obtido a expressão final para CORDIC:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \prod_{n=0}^{N-1} K_c \begin{bmatrix} 1 & -\mu 2^{-n} \\ \mu 2^{-n} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}, \quad (53)$$

$$k_c = \cos(\tan^{-1}(2^{-n})) = \frac{1}{\sqrt{1+2^{-2n}}}, \quad (54)$$

onde:

$$\mu \in \{-1, 1\} \quad (55)$$

$K_c$  é calculado a cada micro-rotação  $n$ , assim seu valor total é dado pelo produto de todos os  $N$  ganhos. O número total de micro-rotações ( $N$ ) é escolhido de acordo com o SQNR (*Signal-to-Quantization-Noise Ratio*) admissível a cada rotação. Segundo (VOLDER, 1959), considerando  $N \rightarrow \infty$ ,  $K_c$  é passa a ser constante e aproximadamente igual a 0,6073. O inverso de  $K_c$  é igual a 1,647, sendo conhecido como Ganhador CORDIC. Tal ganho é independente do ângulo a ser rotacionado, e em muitos sistemas este ganho é só compensado fora do bloco lógico de cálculo do CORDIC.

Como pode ser observado, o algoritmo CORDIC se resume a operações de deslocamento de *bit* e somas. A direção das micro-rotações é determinada por  $\mu$ , que depende diretamente do sinal do ângulo  $\theta_n$  a se rotacionar. Em aplicações onde o ângulo a se rotacionar é previamente conhecido, que é o caso da FFT, as sequências de rotações  $\mu$  podem ser armazenadas em uma memória (ROM) (EL-MOTAZ *et al.*, 2014) *Read-Only Memory*. Isto torna as micro-rotações como  $n$  interações, armazenando em  $z$  os ângulos rotacionados a cada interação de  $\theta_n$ , sendo o algoritmo CORDIC expressado por:

$$x(n+1) = x(n) - [\mu 2^{-n}]y(n), \quad (56)$$

$$y(n+1) = y(n) + [\mu 2^{-n}]x(n), \quad (57)$$

$$z(n+1) = z(n) - \tan^{-1}[\mu 2^{-n}], \quad (58)$$

$$(59)$$

onde:

$$H_r = x(0) + iy(0), \quad (60)$$

$$\theta = z(0), \quad (61)$$

$$n = \{0, 1, \dots, N-1\}, \quad (62)$$

$$\mu = \begin{cases} 1 & z(n) \geq 0, \\ -1 & z(n) < 0 \end{cases}. \quad (63)$$

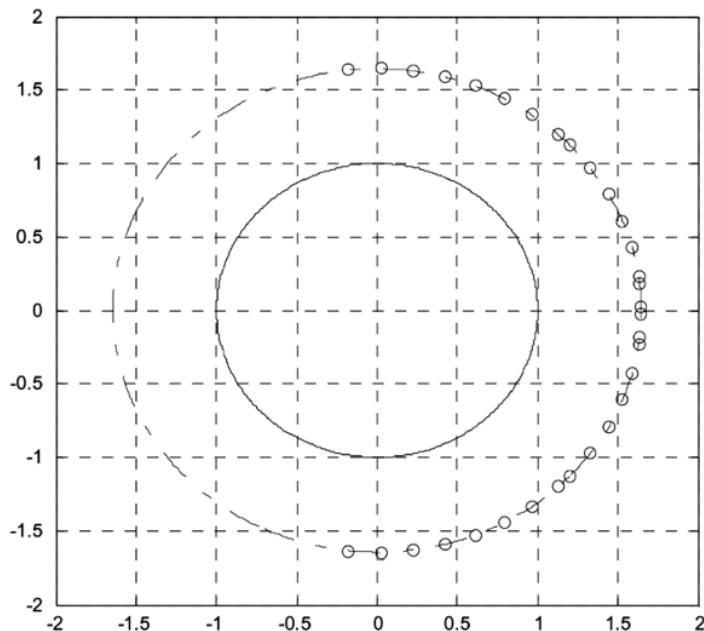
#### 2.4.2 EEAS-CORDIC

No algoritmo CORDIC, cada ângulo de rotação  $\theta_n$  é necessariamente determinado de maneira sequencial após cada interação  $n$ , utilizando o conjunto de ângulos elementares definidos como (WU *et al.*, 2003):

$$S_1 = \{\tan^{-1}(\mu 2^{-n})\}, \quad (64)$$

$$: \mu \in \{-1, 1\}, n \in \{0, 1, \dots, N-1\}. \quad (65)$$

A partir do ângulo inicial de  $H_r$ , o algoritmo Cordic realiza as  $N$  interações, deslocando o vetor por meio do conjunto de ângulos elementares, de forma a reduzir a diferença entre o ângulo atual e o ângulo desejado. Para fins de comparação a Figura (14) apresenta a densidade combinacional dentro do círculo unitário do conjunto de ângulos elementares do algoritmo CORDIC tradicional.



**Figura 14: Ângulos Elementares - Cordic Tradicional com N=4**  
Fonte: (LIN; WU, 2005)

Segundo Wu e Wu (2001) o maior problema do CORDIC é a baixa velocidade computacional de cálculo deste algoritmo, devido principalmente a necessidade de um grande número de interações  $N$  para atingir um erro de aproximação  $\zeta$  aceitável. O erro de aproximação  $\zeta$  é dado por:

$$\zeta = \left| \theta - \sum_{n=0}^{N-1} \mu \theta_n \right| = \left| \theta - \sum_{n=0}^{N-1} \mu \tan^{-1}(2^{-n}) \right|, \quad (66)$$

onde:

$$\mu \in \{-1, 0, 1\} \quad (67)$$

Em aplicações onde os ângulos de rotação são conhecidos, é possível relaxar as restrições da Equação (14), utilizando o método *Angle Recoding* (AR) (WU; WU, 2001). Os AR tem como objetivo reduzir o número de interações CORDIC e o erro  $\zeta$ . Para tal, o AR que expande o conjunto de combinações lineares em (65), adicionando zero ao conjunto de  $\mu$ . Com isso, obtém-se uma melhor aproximação para certos valores de  $\theta$  e uma redução de até 50% no número de interações (MEHER *et al.*, 2009).

Por outro lado, o método *Extended Elementary-Angle-Set Recoding* (EE-ASR) apresenta um método baseado no AR que, além de expandir o conjunto de  $\mu$ , estende também o conjunto de ângulos elementares (*Elementary-Angle-Set*, EAS), visando aumentar a possibilidades de decomposição do ângulo de rotação(WU *et al.*, 2003). Para perceber a modificação que o EEASR propõem, nota-se primeiramente que o conjunto de ângulos elementares no CORDIC utilizando AR é definido como:

$$S_1 = \{\tan^{-1}(\mu 2^{-s})\}, \quad (68)$$

$$: \mu \in \{-1, 0, 1\}, s \in \{0, 1, \dots, N-1\}. \quad (69)$$

Como é possível notar em (69), os ângulos elementares dependem de apenas um termo potência de dois, ou *Signed Power of Two* (SPT). Segundo (WU *et al.*, 2003), para aumentar a precisão dos ângulos elementares e, consequentemente, reduzir o número de interações pode-se adicionar mais um termo SPT em (69), resultando em:

$$S_2 = \{\tan^{-1}(\mu_0 2^{-s_0} + \mu_1 2^{-s_1})\} \quad (70)$$

$$: \mu_0, \mu_1 \in \{-1, 0, 1\}, s_0, s_1 \in \{0, 1, \dots, S\} \quad (71)$$

onde  $S$  é denominado como o número máximo de deslocamentos de *bits* a direita que podem ser realizados. Este valor está diretamente relacionado com o quantidade de *bits* utilizados para representar um número dentro da arquitetura onde o CORDIC é implementado. Por exemplo, em uma aplicação em FPGA, onde a palavra binária utilizada para representar um número inteiro tenha apenas 16 *bits*, não faz sentido o valor de  $S$  ser maior do 15. Portanto, alterando a equação (51) com base em (71) é obtido a expressão para cálculo interativo CORDIC utilizando o EEASR:

$$\begin{bmatrix} x(n+1) \\ y(n+1) \end{bmatrix} = \begin{bmatrix} 1 & \mu_0 2^{-s_0(n)} + \mu_1 2^{-s_1(n)} \\ \mu_0 2^{-s_0(n)} + \mu_1 2^{-s_1(n)} & 1 \end{bmatrix} \begin{bmatrix} x(n) \\ y(n) \end{bmatrix}, \quad (72)$$

$$: \mu_i, \mu_j \in \{-1, 0, 1\}, s_0, s_1 \in \{0, 1, \dots, S\}, \quad (73)$$

$$\theta_n = \tan^{-1} (\mu_0 2^{-s_0(n)} + \mu_1 2^{-s_1(n)}), \quad (74)$$

$$z(n+1) = z(n-1) + \theta_n, \quad (75)$$

É importante notar que ao adicionar mais termos a  $S_1$ , o ganho  $K_c$  também é modificado, passando a ser definido por:

$$K_n = \frac{1}{\sqrt{1 + [\mu_0 2^{-s_0(n)} + \mu_1 2^{-s_1(n)}]^2}}. \quad (76)$$

Com a alteração de  $K_c$ , o ganho passa a não ser mais constante, e varia de acordo com cada interação  $N$ . Sendo assim necessário calcular o ganho  $K_c$  para cada interação afim de realizar a compensação. O valor de  $K_c$  ao final de cada operação de cálculo CORDIC passa a ser definido por:

$$K_c = \prod_{n=0}^{N-1} K_n. \quad (77)$$

Para casos em que o ângulo  $\theta$  a ser rotacionado é conhecido, como é o caso da FFT, é possível escolher previamente o conjunto de valores  $\mu_0$ ,  $\mu_1$ ,  $s_0$  e  $s_1$ , e consequentemente através da Equação (76) determinar o valor de  $K_c$  a ser compensado a cada interação, e por meio de (77) determinar a compensação de  $K_c$  a ser realizada após cada operação de rotação. Para tal é realizado, após as interações de rotação do algoritmo, uma correção no módulo do vetor resultante por meio da seguinte operação de rotação modificada:

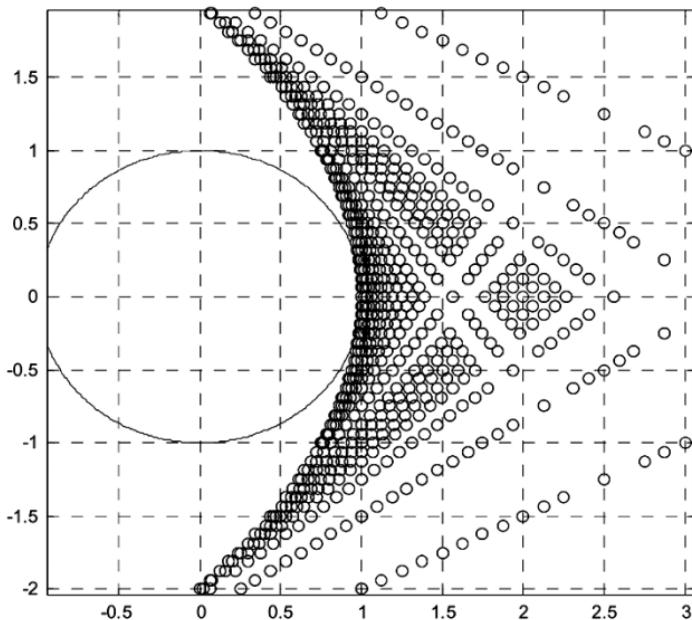
$$x(n+1) = x(n) - [\mu_0 2^{-s_0(n)} + \mu_1 2^{-s_1(n)}]x(n), \quad (78)$$

$$y(n+1) = y(n) - [\mu_0 2^{-s_0(n)} + \mu_1 2^{-s_1(n)}]y(n), \quad (79)$$

$$z(n+1) = z(n). \quad (80)$$

onde  $\mu_0$ ,  $\mu_1$ ,  $s_0$  e  $s_1$  para esta operação são escolhidos de modo a minimizar o erro de compensação de  $K_c$ .

Com a relaxação das restrições introduzidas pelo método AR e pelo EE-ASR, a densidade combinacional dentro do círculo unitário do conjunto de ângulos elementares do Algoritmo CORDIC aumenta, se comparado ao CORDIC tradicional, como pode ser visto na Figura (15).



**Figura 15: EEAS com N=2 e S=4**  
Fonte: Lin e Wu (2005)

#### 2.4.2.1 ALGORITMO TBS

A relaxação das restrições feita pelos métodos AR e EEASR tornaram o algoritmo iterativo CORDIC mais eficiente, porém ele abre uma questão crucial: a determinação dos conjuntos  $\mu$  e  $s$ . No algoritmo CORDIC tradicional, o valor de  $\mu$  estava contido no conjunto  $\{-1, 1\}$  e era determinado com base no sinal de  $z(n)$  a cada interação. No entanto, no método AR  $\mu$  passa a estar contido no conjunto  $\{-1, 0, 1\}$ . Logo, determinar o valor de  $\mu$  a cada interação torna-se uma tarefa de otimização, em relação a minimizar o erro  $\zeta$ , na forma (WU *et al.*, 2003):

$$\min \zeta = \left| \theta - \sum_{n=0}^{N-1} \mu \tan^{-1}(2^{-n}) \right|, \quad (81)$$

onde:

$$\mu \in \{-1, 0, 1\}. \quad (82)$$

No algoritmo CORDIC convencional o conjunto dos ângulos elementares é definido por  $S_1$  em (65), e a cada interação o deslocamento do vetor é feito com base no elemento  $n$  deste conjunto. O método EEASR adiciona mais um termo SPT a expressão do conjunto dos ângulos elementares  $S_2$ , definido em (71), o que possibilita a escolha de qualquer termo  $s_0$  e  $s_1$ . Desta forma, além de agregar a mesma necessidade de determinar o conjunto otimizado  $\mu_0$  e  $\mu_1$  de AR, o EEASR também requer a determinação do conjunto otimizado  $s_0$  e  $s_1$ , visando minimizar também o erro  $\zeta$ . Portanto, a função de minimização do erro  $\epsilon$  é dada pela seguinte expressão:

$$\min \zeta = \left| \theta - \sum_{n=0}^{N-1} \tan^{-1}(\mu_0 2^{-s_0} + \mu_1 2^{-s_1}) \right|, \quad (83)$$

$$: \mu_0, \mu_1 \in \{-1, 0, 1\}, s_0, s_1 \in \{0, 1, \dots, S\}. \quad (84)$$

Para otimizar a Função (84) é possível utilizar um variedade de algoritmos heurísticos ou não heurístico a fim de determinar um conjunto de parâmetros que minimize o erro  $\zeta$ , como por exemplo *Dijkstra*, Caminhos Mínimos ou o mais comum Algoritmo *Greedy*. Wu *et al.* (2003) propõe um método de otimização para parâmetros EEAS chamado *Trellis-Based Search*(TBS).

A partir do número máximo de interações  $N$ , do ângulo de rotação  $\theta$  e de  $W$ , que representa numero de bits de  $\theta$ , o TBS emprega um método de otimização para encontrar os melhores parâmetros  $s^0(n)$ ,  $s^1(n)$ ,  $\mu_0$  e  $\mu_1$  (WU *et al.*, 2003). TBS é baseado no efeito que os diferentes arranjos dos ângulos elementares  $S_2$  possíveis tem sobre  $\zeta$ . O algoritmo segue o seguintes passos:

1. *Inicialização*: Inicialmente é definido o vetor  $r(k)$ , o qual representa o conjunto de ângulos elementares presentes na Equação (84), obtidos através das possíveis combinação não redundantes de  $\mu_0$ ,  $\mu_1$ ,  $s_0$  e  $s_1$ . Em seguida, é definido a matriz  $\phi(n, k)$  para  $(1 \leq n \leq N)$  e  $(1 \leq k \leq z(S_2))$ , representando a estrutura de acumulação do algoritmo utilizada para alcançar o melhor resultado.  $N$  é o número de interações desejadas, enquanto  $z(S_2)$  representa o número de elementos do vetor  $r(k)$ . Para então iniciar o algoritmo,  $\phi(1, k)$  é preenchido com o vetor  $r(k)$ .
2. *Acumulação*: Em cada iteração  $n$  o algoritmo percorre os elementos  $k$  do vetor  $\phi(n+1, k^*)$ , atribuindo a cada valor  $k$  o menor resultado encontrado da expressão  $\|\phi(n, k^*) + r(k) - \theta\|$ . O índice  $i$  varia  $1 \leq i \leq N - 1$ , e  $k$  varia de  $1 \leq k \leq Z(S_2)$ .

3. *Determinação do Ótimo Global:* Ao final do processo de acumulação, os elemento da coluna  $\phi(N,)$  apresentam os resultados globais. Logo, dentro desta coluna é determinado o valor que mais se aproxima de  $\theta$ , defini-se este como  $\theta_{TBS}$ .
4. *Determinação do Caminho Solução:* A partir do elemento  $\theta_{TBS}$  da coluna  $\phi(N,)$  é traçado o caminho reverso até a coluna  $\phi(1,)$ . A começar por ( $k' = \theta_{TBS}$ ) e ( $i = N$ ) é escolhido o valor da coluna ( $i - 1$ ) que minimiza  $\|\phi(1 : z(S_2), i - 1) + r(k') - \theta\|$ . Em seguida se atribui a  $k'$  a posição  $k$  do valor minimizante. Os valores  $k'$  são armazenados em um vetor e representam as combinações de ângulos elementares que minimizam o erro  $\epsilon$ .

Segue abaixo o pseudo-código que implementa o algoritmo TBS.

```

1 % Inicialização
2  $\phi(1, k) = r(k)$  para todo  $k$ ,
3
4 %Acumulação
5 FOR  $i = 1$  to  $N - 1$ 
6   FOR  $k = 1$  to  $Z(S_2)$ 
7      $\phi(i + 1, k) = \min \{ | \phi(i, k^*) + r(k) - \theta | : 1 \leq k^* \leq Z(S_2) \}$ 
8   END
9 END
10
11 %Determinação do Ótimo Global
12  $\theta_{TBS} = \min \{ | \phi(R_m, k^*) - \theta | : 1 \leq k^* \leq Z(S_2) \}$ 
13 Result( $N$ ) =  $k^*$ 
14
15 %Determinação do Caminho Solução
16 FOR  $i = N$  to  $2$ 
17   Encontra  $k'$  tal que  $\phi(i, k^*) = \min \{ | \phi(i - 1, k') + r(k^*) - \theta | : 1 \leq k' \leq Z(S_2) \}$ 
18    $k^* = k'$ 
19   Result( $i - 1$ ) =  $k'$ 
20 END
21

```

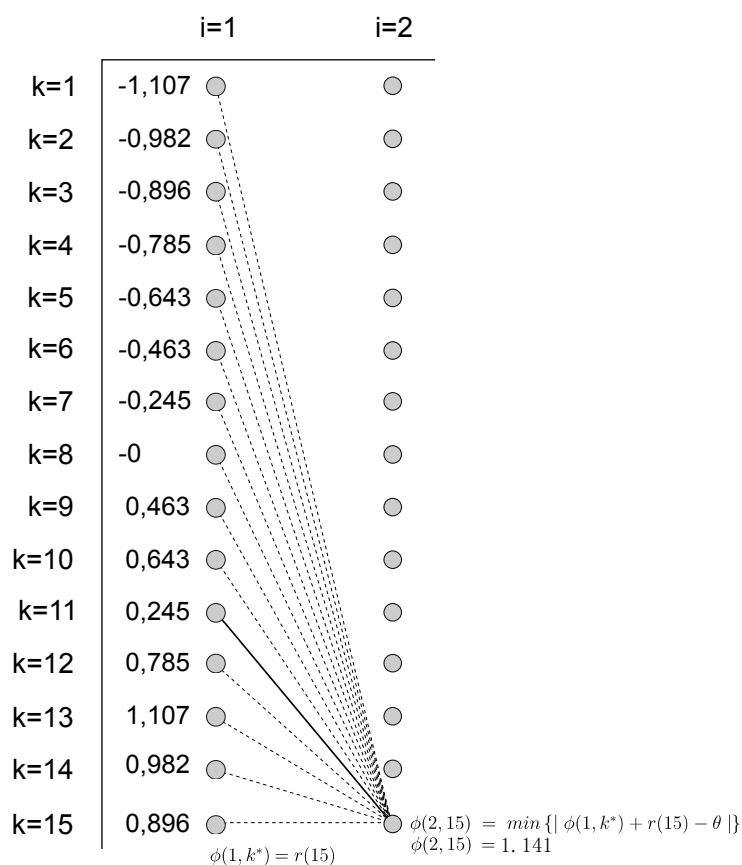
#### Código 2.1: Algoritmo TBS

Fonte : Wu et al. (2003)

Nos trabalhos de Wu et al. (2003), os autores utilizam um exemplo base para explicar o algoritmo TBS, o qual também será usado aqui. Para este exemplo,

o ângulo de rotação  $\theta$  será  $\pi/3$ , o número máximo de interações será  $N = 4$  e a resolução de  $\theta$  será de  $W = 4$  bits.

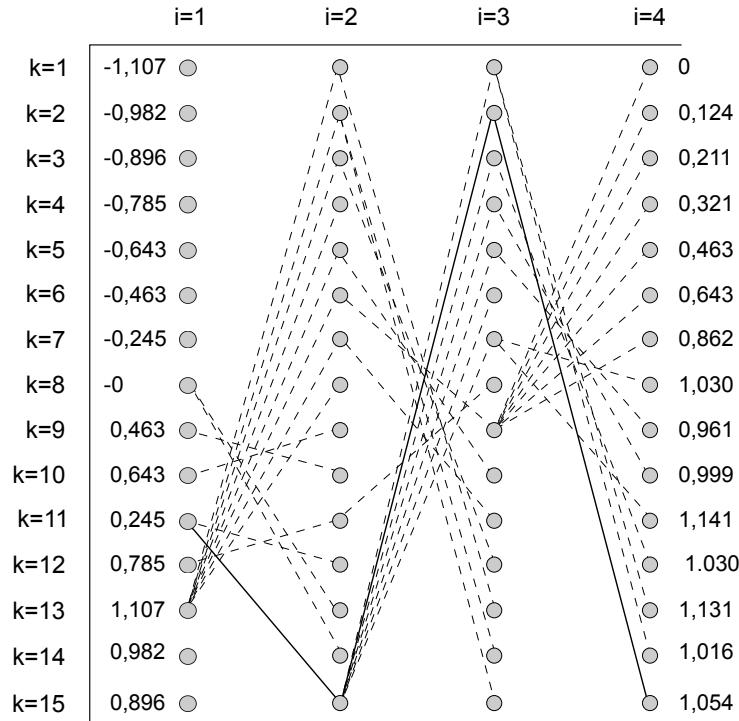
A primeira etapa do algoritmo então inicia com a definição do vetor  $r(k)$ , e o preenchimento da primeira coluna da matriz  $\phi(1, )$ , como pode ser visto na Figura (16). Na segunda etapa do algoritmo é realizada a acumulação, onde cada elemento das colunas 2, 3 e 4 são preenchidos com base na coluna anterior a partir da expressão  $\|\phi(n, k^*) + r(k) - \theta\|$ . Na Figura (16) o elemento  $\phi(2, 15)$  representa esta etapa, onde o valor do elemento  $\phi(2, 15)$  é definido a partir da soma do ângulo representante da linha 15 ( $r(15)$ ), com o elemento da coluna 1 que minimiza o erro de aproximação do resultado desta soma com  $\theta$ , que neste caso é  $\pi/3$ . Este processo se estende até completar a coluna 4 da matriz  $\phi$ .



**Figura 16: Exemplo de execução do Algoritmo TBS**  
**Fonte: Autoria Própria**

Após a etapa de acumulação, a última coluna de  $\phi$  apresenta os resultados globais da acumulação. Logo, para determinar o valor ótimo global é escolhido na coluna  $\phi(4, )$  o valor  $\theta_{TBS}$ , que mais se aproxima de  $\theta$  ou  $\pi/3$ . Após esta etapa é iniciada a determinação do vetor *Result*, o qual deve conter o caminho solução necessário

para atingir o valor de ótimo global  $\theta_{TBS}$ . Na Figura (17) é apresentado os caminhos mínimos para cada coluna da matriz  $\phi$ , consequentemente, o melhor caminho encontrado pelo algoritmo.



**Figura 17: Exemplo de execução do Algoritmo TBS**  
Fonte: Autoria Própria

A tabela (2) apresenta os valores encontrados para  $\phi$  na execução do exemplo apresentado. Em destaque estão os valores do caminho solução, que são armazenados em *Result*.

	i=1	i=2	i=3	i=4
k=1	-1.1071	0	0.0339	0
k=2	-0.9828	0.1244	0.1582	0.1244
k=3	-0.8961	0.2111	0.2450	0.2111
k=4	-0.7854	0.3218	0.3556	0.3218
k=5	-0.6435	0.4636	0.4975	0.4636
k=6	-0.4636	0.6435	0.6774	0.6435
k=7	-0.2450	0.8622	0.8961	0.8622
k=8	0	1.1071	1.0304	1.0304
k=9	0.4636	1.1071	1.1071	0.9612
k=10	0.6435	1.1071	1.1071	0.9991
k=11	0.2450	1.0304	1.1071	1.1410
k=12	0.7854	1.0304	0.9965	1.0304
k=13	1.1071	1.1071	1.1071	1.1410
k=14	0.9828	0.9828	1.1071	1.0167
k=15	0.8961	1.1410	1.0204	1.0543

**Tabela 2: Matriz  $\phi$  para o dado Exemplo**  
**Fonte: Autoria Própria**

### 2.4.3 MSR-CORDIC

Como pode ser visto na Seção (2.4.1), o Algoritmo Cordic Tradicional possui a necessidade de realizar a correção do ganho  $k_c$  após o processo de rotação do vetor, além de requisitar um número elevado de interações a fim de reduzir o erro de aproximação do vetor. Para reduzir o número de interações CORDIC, o EEAS aumenta o número de termos SPT, porém acaba por impactar o ganho  $K_c$ , que passa a não ser mais constante e necessita ser compensado a cada iteração. Tanto no algoritmo tradicional quanto o EEAS a correção final de  $K_c$ , que se estabelece sempre abaixo da unidade, provoca uma degradação do SQNR.

Segundo Lin e Wu (2005), para evitar esta degradação é necessário manter o módulo do vetor de entrada o mais perto da unidade a cada interação. Assim, Lin e Wu (2005) reformulam o algoritmo Cordic para um novo esquema que passa a distribuir os termos SPT de forma diferente do EEAS, com o intuito de dar mais liberdade ao ganho  $K_c$  e reduzir o número de interações necessárias para atingir um erro de aproximação admissível. Tal algoritmo pode ser expressado por:

$$\begin{bmatrix} x(n+1) \\ y(n+1) \end{bmatrix} \begin{bmatrix} \sum_{i=1}^I \mu_i 2^{-s_i(n)} & -\sum_{j=1}^J \mu_j 2^{-s_j(n)} \\ \sum_{j=1}^J \mu_j 2^{-s_j(n)} & \sum_{i=1}^I \mu_i 2^{-s_i(n)} \end{bmatrix} \begin{bmatrix} x(n) \\ y(n) \end{bmatrix}, \quad (85)$$

$$z(n+1) = z(n-1) + \theta_n, \quad (86)$$

$$: \mu_i, \mu_j \in \{-1, 0, 1\}, s_i, s_j \in \{0, 1, \dots, S\}, \quad (87)$$

$$: I + J = N_{spt}. \quad (88)$$

A reformulação apresentada por Lin e Wu (2005) incrementa termos SPT nas parcelas  $x(n)$  de  $x(n+1)$  e  $y(n)$  de  $y(n+1)$ , a fim de possibilitar o ganho  $K_c$  se tornar maior ou menor do que a unidade, dependendo da escolha dos parâmetros Cordic. Assim é possível realizar a operação de microrotação e a compensação do ganho  $K_c$  simultaneamente. Por tais motivos, este algoritmo CORDIC é denominado de *Mixed Scaling Rotation* (MSR). Consequentemente, ao inserir mais termos na função CORDIC, tanto  $\theta_n$  quanto  $K_c$  precisam ser corrigidos (KUO *et al.*, 2003). Portanto  $\theta_n$  e  $K_c$  passam a definidos por:

$$\theta_n = \tan^{-1} \left( \frac{\sum_{j=1}^J \mu_j 2^{-s_j(n)}}{\sum_{i=1}^I \mu_i 2^{-s_i(n)}} \right), \quad (89)$$

$$K_n = \frac{1}{\sqrt{\left(\sum_{i=1}^I 2^{-s_i(n)}\right)^2 + \left(\sum_{j=1}^J 2^{-s_j(n)}\right)^2}}, \quad (90)$$

$$K_c = \prod_{n=0}^{N-1} K(n). \quad (91)$$

Como pode ser percebido em (90) e (89), com a escolha adequada dos conjuntos de termos  $\mu_i$ ,  $\mu_j$ ,  $S_i$  e  $S_j$ , e dos valores de  $I$  e  $J$  é possível aproximar  $K_c$  da unidade a cada interação ao mesmo tempo em que o erro  $\zeta$  é reduzido. No MSR o conjunto de ângulos elementares é definidor por:

$$S_3 = \left\{ \tan^{-1} \left( \frac{\sum_{j=1}^J \mu_j 2^{-s_j(n)}}{\sum_{i=1}^I \mu_i 2^{-s_i(n)}} \right) \right\}, \quad (92)$$

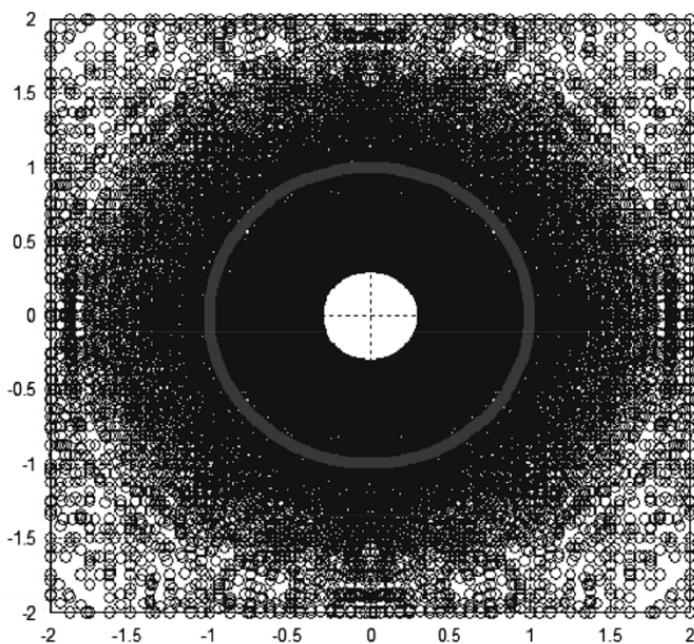
$$: \mu_i, \mu_j \in \{-1, 0, 1\}, s_i, s_j \in \{0, 1, \dots, S\}, \quad (93)$$

sendo o erro  $\zeta$  é definido por:

$$\min \zeta = \left| \theta - \sum_{n=0}^{N-1} \tan^{-1} \left( \frac{\sum_{j=1}^J \mu_j 2^{-s_j(n)}}{\sum_{i=1}^I \mu_i 2^{-s_i(n)}} \right) \right|, \quad (94)$$

$$: \mu_i, \mu_j \in \{-1, 0, 1\}, s_0, s_1 \in \{0, 1, \dots, S\}. \quad (95)$$

Com mais graus de liberdade o MSR possui uma densidade combinacional dentro do círculo unitário maior do que o CORDIC tradicional e o EEAS-CORDIC (KUO *et al.*, 2003), como pode ser visto na Figura (18).



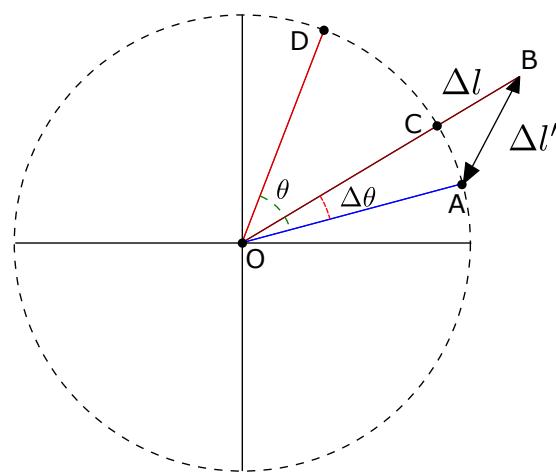
**Figura 18: MSR com I=2, J=1 e N=2**

Fonte: (LIN; WU, 2005)

O incremento de mais termos SPT na definição do ângulo de rotação  $\theta_n$  acaba por dar ao conjunto dos ângulos elementares mais liberdade combinacional dentro do círculo unitário. No algoritmo CORDIC tradicional e no EEAS havia uma dificuldade de realizar rotações maiores que  $\pi/4$ , devido a distribuição dos ângulos elementares, como pode ser visto nas Figura (14) e (15). Devido a esta dificuldade era necessário realizar uma pré-rotação no vetor de entrada, quando o ângulo de rotação era maior que  $\pi/4$ , a fim de reduzir o ângulo para uma faixa mais aceitável de rotação, o que aumenta o custo computacional do algoritmo, e demandava mais recursos de hardware. No algoritmo MSR o conjunto dos ângulos elementares no círculo unitário é maior e melhor distribuídos como visto na Figura (18). Isto possibilita ao MSR alcançar ângulos de rotação de  $0$  á  $2\pi$ , varrendo todo o círculo unitário.

O desempenho do algoritmo EEAS, como mostrado na seção anterior, é diretamente relacionado com a escolha dos parâmetros CORDIC  $\mu_0$ ,  $\mu_1$ ,  $s_0$  e  $s_1$ . De maneira análoga é a determinação dos parâmetros CORDIC  $J$ ,  $I$ ,  $\mu_i$ ,  $\mu_j$ ,  $s_i$  e  $s_j$  que garante a convergência e eficiência do algoritmo.

Para determinar os parâmetros Cordic do algoritmo MSR, Lin e Wu (2005) utiliza a análise do erro entre o vetor idealmente rotacionado, e o vetor que na prática do Algoritmo MSR consegue alcançar no rotacionamento. Para exemplificar, considere que se deseja rotacionar um vetor  $\overline{OD}$  pelo ângulo  $\theta$  até a posição do ponto  $A$ , formando assim um novo vetor rotacionado  $\overline{OA}$ . Considerando que o MSR não consegue rotacionar o vetor  $\overline{OD}$  por exatamente  $\theta$ , e que devido ao ganho Cordic o módulo do vetor rotacionado seja diferente do vetor  $\overline{OD}$ . Logo, a nova posição do vetor rotacionado é  $\overline{OB}$ , como mostrado na Figura (19)(LIN; WU, 2005).



**Figura 19: MSR com I=2, J=1 e N=2**  
Fonte: (LIN; WU, 2005)

Analizando a Figura (19), o erro de aproximação  $\epsilon$  do algoritmo MSR é representado pelo trecho  $\overline{OB}$ , denominado  $\Delta l'$ . Já o erro angular é representado por  $\Delta\theta$ , e o erro do módulo é representado pelo trecho  $\overline{OC}$ , denominado  $\Delta l'$ . Segundo Lin e Wu (2005), quando  $\Delta\theta$  é suficientemente pequeno é possível obter a seguinte expressão:

$$\Delta l'^2 = \overline{OA}^2 + \overline{OB}^2 - 2\overline{OA} \times \overline{OB} \cos(\Delta\theta), \quad (96)$$

$$\Delta l'^2 = 1 + (1 + \Delta l)^2 - 2 \times 1 \times (1 + \Delta l) \times \sqrt{1 - \sin^2(\Delta\theta)}, \quad (97)$$

$$\Delta l'^2 \approx \Delta l^2 + \Delta\theta^2. \quad (98)$$

Por meio de (98) é possível concluir que o erro angular  $\Delta\theta$ , e o erro modular  $\Delta l$  possuem o mesmo impacto no erro de aproximação do algoritmo MSR. Portanto, afim de otimizar o desempenho do algoritmo MSR é necessário escolher um conjunto de parâmetros CORDIC que privilegiem igualmente a redução de ambos os erros de aproximação (LIN; WU, 2005).

Assim determinar o conjunto dos melhores valores para os parâmetros CORDIC  $J$ ,  $I$ ,  $\mu_i$ ,  $\mu_j$ ,  $s_i$  e  $s_j$  nada mais é do que uma tarefa de otimização, onde a equação de minimização é a expressão do erro  $\epsilon$ , o qual pode ser definido como:

$$\min \epsilon = \sqrt{\Delta l^2 + \Delta \theta^2}. \quad (99)$$

Para solucionar esta tarefa de otimização de  $\epsilon$  é possível utilizar diferentes tipos de algoritmos, e inclusive adaptar o algoritmo TBS já apresentado. Segundo Lin e Wu (2005), existe uma certa restrição quanto a utilização de algoritmos como o *greedy*, pois este tipo de algoritmo Gulosso não garante a melhor solução dentro do conjunto de soluções possíveis deste problema.

#### 2.4.3.1 ANÁLISE DO ERRO

Um importante fator para a determinação do melhor conjunto de parâmetros Cordic para o MSR, além da minimização do erro  $\epsilon$ , é o efeito do ruído de arredondamento (*Roundoff Noise Analysis*)  $e_n$ , e o erro de *overflow*, ou estouro da palavra binária. Ambos os efeitos ocorrem ao longo das operações lógicas promovidas pelo algoritmo CORDIC, em especial devido a operação de compensação do ganho  $K_c$ . Esta compensação envolve uma operação de multiplicação, o que acentua os problemas de arredondamento da parte fracionária e, dependendo do valor da compensação, pode causar um *overflow* da palavras binária.

Nos dispositivos digitais o número de *bits* utilizados para representar sinais quantizados é limitado de acordo com a arquitetura. Este limite de representação acaba por impactar na resolução da palavra binária para números fracionários, o que força a realização de arredondamentos na representação binária.

Para representar o conjunto dos números racionais no mundo digital, a arquitetura dos sistemas digitais reserva parte dos *bits* da palavra binária para representar o fracionário do número racional. Assim, segundo Lin e Wu (2005), os níveis de

quantização para um sinal digital são definidos como:

$$\{-2^{W-i-1}, \dots, -2^{-i+1}, -2^{-i}, -2^{-i+1}, \dots, 2^{W-i-1}\}, \quad (100)$$

Onde  $W$  representa o numero de *bits* da palavra binária, e  $i$  representa o número de dígitos fracionários.

Uma solução simples para reduzir o efeito do  $e_n$  é aumentar o número de *bits*  $W$  e  $i$ , melhorando a resolução da representação. Porém, esta solução, como afirma (LIN; WU, 2005), provocaria uma redução na velocidade computacional do sistema, e ainda consumiria mais recursos de *hardware*. Outra opção seria reduzir o número de *bits* utilizados para representar a parte inteira do número, e transferir estes *bits* para a parte fracionária. Porém, este método pode provocar um *overflow* da palavra binária, o que é ainda pior do que um ruído de arredondamento.

Considerando a amplitude  $\rho$  de um sinal de entrada, definido entre os intervalos:

$$-2^{W_a} \leq \rho \leq 2^{W_b} - 1, \quad \text{para } W_a, W_b \geq 0, \quad (101)$$

onde  $W_a$  e  $W_b$  representam respectivamente o limite superior e inferior da amplitude do módulo deste sinal. Ainda:

$$W_{max} = \max\{W_a, W_b\}. \quad (102)$$

Como o limite absoluto deste sinal. O número mínimo de *bits* necessários para representar este sinal seria de  $(W_{max} + 1)$ . Assim, segundo Lin e Wu (2005), para evitar um *overflow* seria necessário manter o sinal de entrada dentro da seguinte restrição:

$$2^{W_{max}} \leq 2^{W-1-i}. \quad (103)$$

Para solver o problema do ruído de arredondamento, Lin e Wu (2005) assumem que o  $e_n$  é uniformemente distribuído e não possui correlação com outros sinais. Além de considerar que, baseado nos níveis de quantização, o  $e_n$  está situado em uma faixa simétrica de  $(-2^{-i-1}, 2^{-i-1})$ . Portanto, a variância de  $e_n$  é dada por:

$$\sigma_{e_n}^2 = \frac{V_{LSB}^2}{12}, \quad (104)$$

onde  $V_{LSB}$  é igual a  $2^i$ . Logo, a variância do erro de arredondamento é proporcional ao quadrado valor do ultimo *bit* da palavra binária.

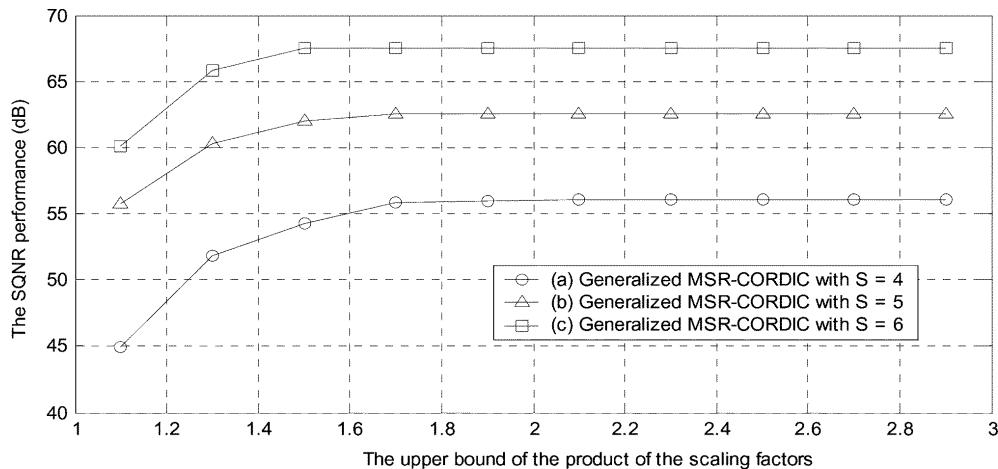
Como base em (104), Lin e Wu (2005) analisam o efeito a operação de rotação tem sobre a variância de  $e_n$ , e mais especificamente o efeito que o ganho  $K_c$  tem sobre o SQNR. Quando o ganho  $K_c$  é  $> 1$  ou  $< 1$  SQNR é reduzido para  $1/(K_n)$  a cada iteração CORDIC.

O ganho  $K_C$  é fator importante no projeto dos parâmetros Cordic para o MSR, afetando tanto o erro  $e_n$  quanto  $\epsilon$ . Para refinar a função de minimização apresentada em (99), Lin e Wu (2005) adicionam a seguinte restrição em relação ao valor do ganho  $K_c$ :

$$P_{lower} = 1/P_{upper}, \quad (105)$$

onde  $P_{upper}$  e  $P_{lower}$  são, respectivamente, o limite superior e o limite inferior de  $K_n$  a cada interação.

O gráfico da Figura (20) mostra uma análise prática entre o nível do SQNR e o  $P_{upper}$ , para diferentes valores de deslocamento de bits ( $S$ ). Por meio desta análise Lin e Wu (2005) sugerem que para um bom projeto de parâmetros MSR Cordic, onde  $N_{spt} = 3$  e  $N = 3$ , o valor de  $P_{upper}$  deve estar próximo de 1,5. Já que, como pode ser visto na Figura (20), para valores maiores o desempenho de SQNR é saturado. Logo, para este caso, é possível adicionar mais esta restrição a equação de minimização do erro  $\epsilon$ .



**Figura 20: Relação entre o  $P_{upper}$  e o SQNR para MSR-CORDIC, como  $N_{spt} = 3$  e  $N = 3$**

**Fonte:** (LIN; WU, 2005)

O último aspecto da melhoria do erro  $e_n$  é a escolha dos parâmetros  $I$  e  $J$ . Estes parâmetros estão diretamente relacionados a  $N_{spt}$ , o número de termos SPT

presentes na equação  $x(n+1)$  e  $y(n+1)$ . Esta relação é descrita por (88). O valor de  $N_{spt}$  é uma escolha que impacta no número de deslocadores de *bits* (*shifters*) necessários para realizar a operação Cordic, sendo mais comum na bibliografia observar implementação MSR Cordic com  $N_{spt} = 3$  ou  $N_{spt} = 4$  (PARK; YU, 2012) (LIN; WU, 2005) (MEHER *et al.*, 2009).

Os valores  $I$  e  $J$  podem ser escolhidos como fixos (Modo Normal) ou podem ser dinâmicos a cada operação de rotacionamento (Modo Generalizado). Tanto no modo Normal quanto o modo Generalizado, o valor de  $I$  e  $J$  devem obedecer a restrição  $I + J = N_{SPT}$ , porém ao utilizar o modo Normal é possível reduzir a complexibilidade do *hardware* e alcançar um desempenho aceitável. Para tal, quando em modo de operação Normal, os parâmetros deve  $I$  e  $J$  devem atender a duas restrição dadas a seguir (LIN; WU, 2005):

- Se  $N_{spt}$  é par:  $I = J = N_{spt}/2$ ,
- Se  $N_{spt}$  é ímpar:  $J = (N_{spt} + 1)/2$  e  $I = J - 1$ .

A operação em modo Generalizado permite que o desempenho do MSR Cordic seja maior, já que nesta configuração as possibilidades combinatórias dos parâmetros Cordic são maiores. Neste modo passa a existir, além da operação normal de rotacionamento, a possibilidade de ocorrer a operação de multiplicação de escala, quando  $J = 0$ , e a operação de inversão de escala, quando  $I = 0$ . Estas operações adicionais maximizam o desempenho nas rotações em ângulos como  $\pi/4$ ,  $\pi/2$  e  $\pi$ . Na prática, este modo requer o uso de mais 3 *switches* de controle, o que em muitas aplicações torna viável aumentar o consumo de recursos para alcançar um melhor desempenho.

Alinhando todas as restrições apresentadas com a equação de minimização apresentada em (99), é obtido um conjunto de parâmetros Cordic que otimizam a operação de rotacionamento, elevando o valor de SQNR do sinal.

## 2.5 FPGA

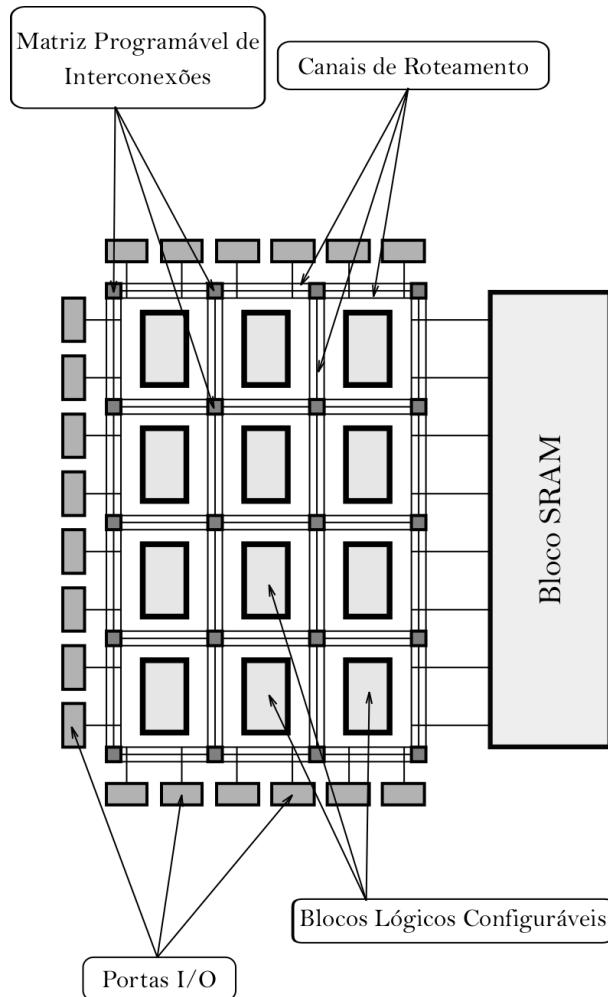
Moore (2007, p. 4) define a FPGA como um dispositivo semicondutor capaz de ser totalmente redefinido após sua fabricação, permitindo ao programador reconfigurar produtos e funções já implementadas, adaptando o *hardware* a novas funções. De forma prática, a FPGA permite uma flexibilidade em um projeto, podendo mudar

a forma como ele é implementado, sem a necessidade de se construir um *hardware* novo.

Para Moore (2007, p. 4), comparado com as outras formas de construir um hardware, a FPGA oferece duas grandes vantagens em uma aplicação. Primeiro, para uma aplicação, ao invés de se utilizar um circuito integrado padrão comercial, que geralmente é super ou subdimensionado, ou ainda desenvolver um novo projeto de circuito integrado específico, consumindo tempo e recursos, a FPGA possibilita desenvolver um *hardware* exatamente dentro das especificações, personalizado e otimizado para a função destinada. Em segundo, porém tão importante quanto, é que essa capacidade de personalização de *hardware* possibilita a realização de operações de modo mais simplificado, rápido e energeticamente eficiente se comparado a um microprocessador.

### 2.5.1 ASPECTOS CONSTRUTIVOS DA FPGA

As FPGAs são baseadas em unidades lógicas elementares básicas, ou (BLE) *Basic Logic Elements*, dentro de uma hierarquia de interconexões reconfiguráveis que permitem que os BLEs sejam fisicamente conectados uns aos outros de diferentes formas, criando uma enorme variedade de componentes digitais. A arquitetura das FPGAs modernas são constituídas, basicamente, por conjunto de memórias de armazenamento em massa SRAM (*Static Random Access Memory*), Portas de Entrada/Saída, blocos lógicos configuráveis (CLB) e sistema de roteamento, como pode ser visto na Figura (21) (MOORE, 2007, p. 5).

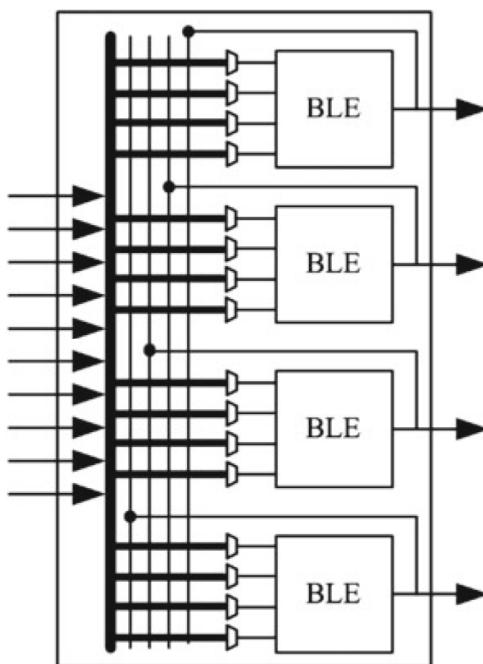


**Figura 21: Arquitetura Tipica de uma FPGA**  
**Fonte:** Adaptado Meyer-Baese (2007, p. 6)

Os CLBs são blocos que realizam operações lógicas básicas e armazenam pequenos volumes de dados. Comumente, as operações complexas, necessárias para o processamento de uma aplicação, são divididas em processos mais simples para cada uma das CLBs selecionadas, de modo que a soma das tarefas de cada CLB seja equivalente a uma operação complexa, em uma estratégia de divisão e conquista. Para realizar operações lógicas básicas e ainda armazenar pequenos volumes de dados, os CLBs tecnicamente poderiam ser apenas um pequeno circuito de transistores (granularidade fina), ou até mesmo um processador completo (granularidade grosseira). Se os CLBs fossem de granularidade fina, para realizar tarefas complexas seria necessário um grande número de CLBs e um sistema de roteamento complexo para interconectá-los, o que resultaria em uma FPGA de baixo desempenho e um elevado consumo energético. Por outro lado, se as CLBs forem de uma granularidade mais grosseira, seria um desperdício de recurso utilizá-los em operações mais simples (FAROOQ *et al.*,

2012, p. 11). Assim a escolha do nível de complexabilidade, ou granulação, das CLBs de uma FPGA é um compromisso de otimização de recursos.

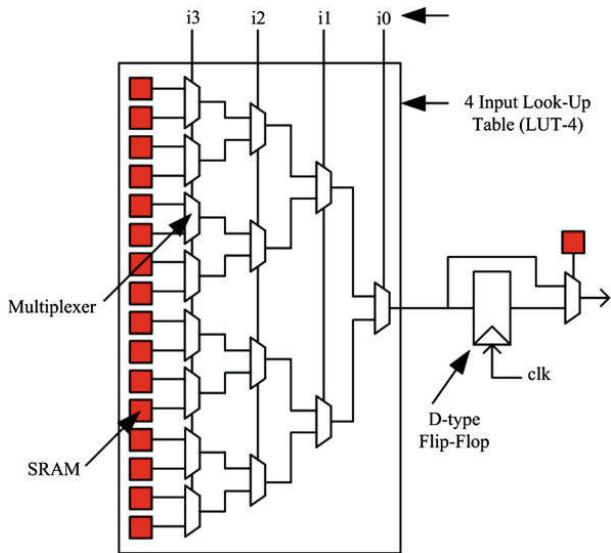
Segundo Farooq *et al.* (2012, p. 11), dentro da gama de granulação das CLBs, algumas arquiteturas incluem o uso de portas NAND, interconexão de multiplexadores e tabelas de busca LUT (*Lookup Table*). Em especial, fabricantes como a Xilinx utilizam CLBs baseadas em LUTs, já que CLBs baseadas em LUT oferecem uma boa relação de granulação, otimizando os recursos da FPGA para aplicações simples até as mais complexas. Este tipo de CLB pode incluir uma único BLE, ou mesmo um *cluster* de BLEs interconectados, como mostrado na Figura (22).



**Figura 22: Arquitetura de uma CLB com 4 BLEs**

**Fonte:** Adaptado Farooq *et al.* (2012, p. 13)

Segundo Farooq *et al.* (2012, p. 11), um BLE mais simples consiste basicamente de um LUT e um *Flip-Flop* tipo D, como pode ser visto na Figura (23). Um LUT com  $k$  entradas pode implementar  $k$  funções booleanas utilizando os espaços de memória SRAM dentro da LUT. O exemplo apresentado na Figura (23) utiliza 16 bits de memória SRAM, os quais são conectadas a entrada do multiplexador que possui 4 bits de seleção, e cuja saída é ligada ao *flip-flop*. Esta configuração permite que a LUT tenha  $2^k$  combinações das  $k$  operações booleanas.



**Figura 23: Arquitetura de uma BLE (Basic Logic Element)**

Fonte: Adaptado Farooq et al. (2012, p. 13)

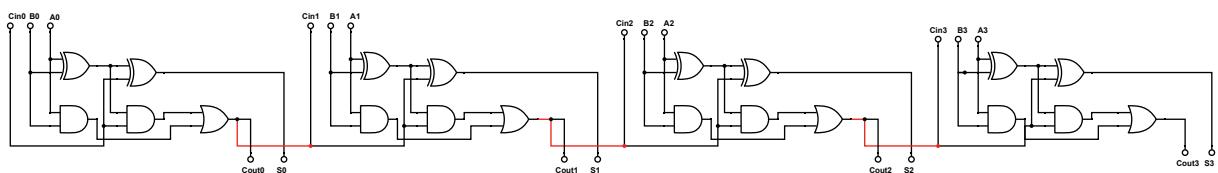
Um único BLE é capaz de realizar algumas operações booleanas básicas, porém em clusters as combinações de operações aumentam. FPGAs modernas tipicamente contém de 4 a 10 BLEs em um único cluster. Porém, estas FPGAs não possuem apenas BLEs idênticas, na verdade há uma grande heterogeneia de blocos, sendo muitos deles desenvolvidos para propósitos específicos. Entre estes blocos de propósito específico estão multiplicadores, somadores, memórias e DSPs (*Digital Signal Processor*), entre outros. Estes blocos são desenvolvidos para otimizar o espaço, processamento, roteamento e demais recursos de *hardware* que seriam necessários para implementar as mesmas funções em BLEs comuns, sendo essenciais em certas aplicações Farooq et al. (2012, p. 10).

A implementação de qualquer circuito lógico é feita pela associação de diferentes blocos lógicos e pelas portas de entrada e saída da FPGA, os quais são conectados uns aos outros por meio da rede de roteamento programável, ou PLN (*Programmable Logic Network*). Na Figura (21) a PLN é representada pela Matriz Programável de Interconexões e pelos Canais de Roteamento. Para que a FPGA possa implementar qualquer circuito digital, as interconexões de roteamento devem ser flexíveis para suportar a grande variedade de conexões demandada, otimizando sempre as distâncias das conexões e reduzindo a latência dos sinais. Portanto, ao projetar um circuito a ser implementado na FPGA deve ser ter especial atenção a forma como o roteamento dos blocos lógicos é feito, buscando flexibilidade e eficiência Farooq et al. (2012, p. 13).

Nas FPGAs modernas, além da unidades de armazenamento de Dados SRAM contido dentro das BLEs, mais especificamente nas LUTs, existe ainda grandes blocos SRAM isolados das BLEs, destinados a funcionar como o armazenamento de dados em massa. Estes blocos são importantes em aplicações digitais onde é necessário armazenar, como por exemplo, dados de amostragem ou mesmo dados que devem aguardar para serem passados para uma próxima etapa de processamento, ou mesmo transmitidos para fora da FPGA pelas portas de entrada e saída de dados. Estes blocos de memória é apresentada na Figura (21) como parte integrante da arquitetura tipica de uma FPGA.

### 2.5.2 PROGRAMAÇÃO NA FPGA

O desenvolvimento de uma aplicação em FPGA começa pela elaboração do Design de Referência, que nada mais é do que uma descrição lógica equivalente que deve ser programada na FPGA para implementação das operações lógicas desejadas. O Design de Referência pode ser feito utilizando diagrama de portas lógicas ou ainda usando qualquer linguagem de descrição de hardware como VHDL (*VHSIC Hardware Description Language*) ou Verilog. A maioria dos ambientes de desenvolvimento integrados (IDE - *Integrated Development Environment*), disponibilizados pelos fabricantes de FPGAs possuem a opção de programação visual utilizando portas lógicas. A Figura (24) apresenta o diagrama de um somador de 4 bits, feito no IDE ISE Design Suite 14, da fabricante Xilinx.



**Figura 24: Diagrama Lógico Full Adder 4 Bits - ISE Design Suite 14**  
Fonte: Autoria Própria

Programar um somador de 4 bits, como apresentado na Figura (24), utilizando portas lógicas parece ser realmente simples. Porém, para circuitos mais elaborados como um contador, ou até mesmo uma máquina de estados, pode ser tornar impraticável utilizar este método de desenvolvimento. Para circuitos mais elaborados é possível utilizar uma linguagem de descrição de *hardware* (HDL), para tornam o desenvolvimento mais fácil e intuitivo. Tendo em vista, que mesmo esta linguagem não ser destinada a execução por um processador, e na verdade descrever o comporta-

mento de circuitos lógicos, ela é visivelmente mais familiar ao desenvolvedor que está acostumado a linguagens de programação, como por exemplo Linguagem C.

O Código VHDL (2.5.2), descrito abaixo, apresenta o mesmo somador de 4 bits apresentado anteriormente em (24).

```
1 library IEEE;
2 use IEEE.STD.LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity FullAdder4Bits is
6     port(InputA : in unsigned(3 downto 0);
7          InputB : in unsigned(3 downto 0);
8          Result : out unsigned(3 downto 0);
9          CarryOut : out std_logic);
10 end entity;
11
12 architecture Behavioral of FullAdder4Bits is
13
14     signal Aux : unsigned(4 downto 0);
15
16 begin
17
18     Aux <= ("0" & InputA) + InputB;
19     Result <= temp(3 downto 0);
20     CarryOut <= Aux(4);
21
22 end architecture Behavioral;
```

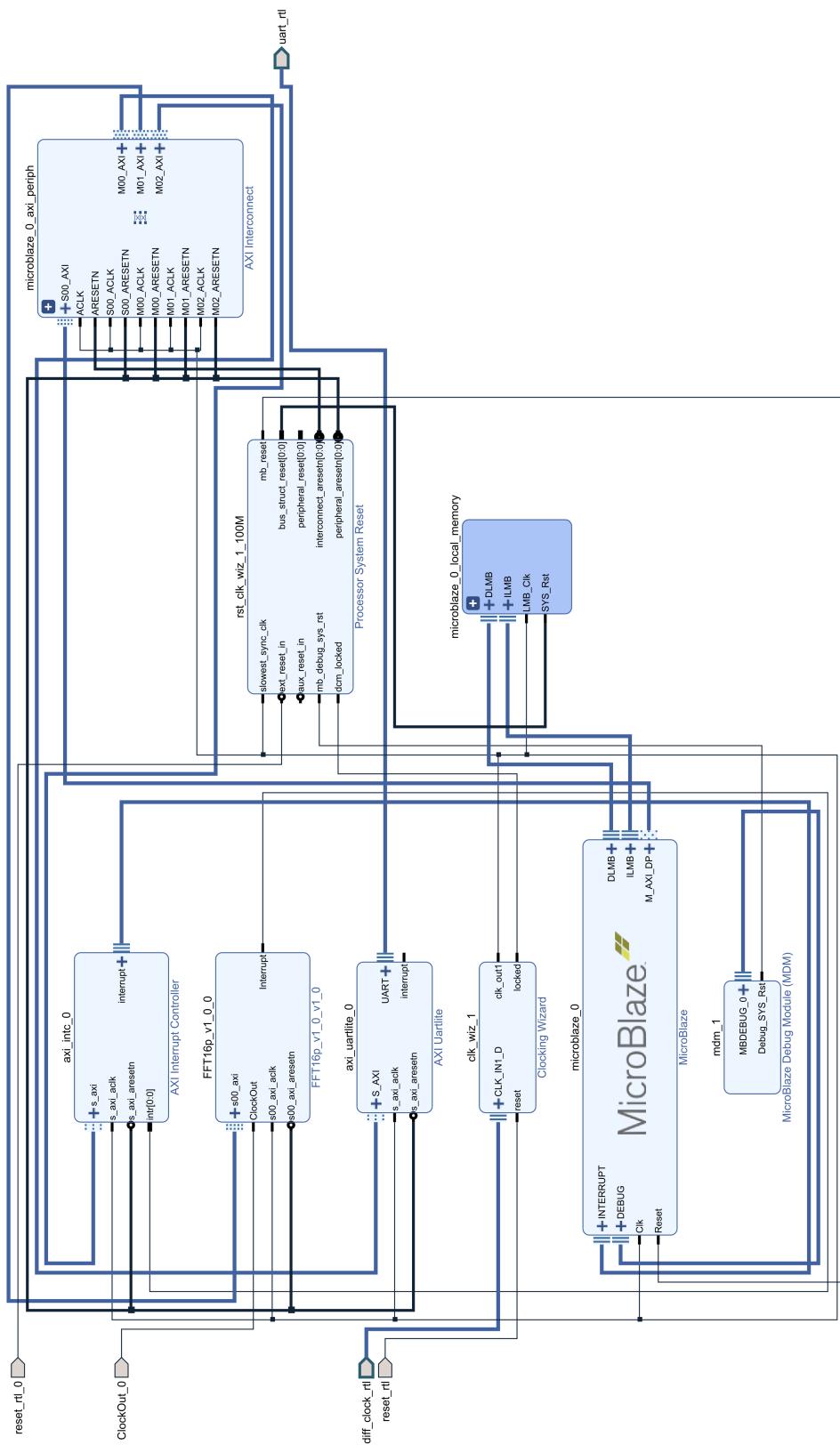
### Código 2.2: Full Adder 4 bits

Fonte : Autoria Própria

Após definir o Design de Referência, via diagrama de portas lógicas ou mesmo em linguagem VHDL, o próximo passo é utilizar uma ferramenta de síntese da própria IDE para converter o design de referência em um conjunto de configurações de registradores, conexões e portas que serão usadas na FPGA para implementar as funcionalidades descritas no design de referência. Durante o processo de síntese a ferramenta verifica a sintaxe do código HDL, e a coerência entre as portas de externas, como sinais de *clock* e portas de entrada e saída, selecionadas no design de referência.

Ao desenvolver um design para implementação em FPGA é comum dividir as funcionalidades do sistema em pequenos blocos de modo a modularizá-lo, permitindo reaproveitar trechos de circuitos lógicos em diferentes aplicações. Segundo Moore (2007, p. 20) ao longo dos anos os fabricantes FPGA perceberam também que vários dos sistemas implementados pelos desenvolvedores tinham funcionalidades muito comuns, como por exemplo processamento gráfico, interfaces de comunicação serial e até mesmo implementação de microprocessadores. Logo, não fazia sentido o desenvolvedor desperdiçar tempo implementando um circuito extremamente comum. Assim, os fabricantes passaram a oferecer bibliotecas circuitos lógicos modularizados para funcionalidades comuns, que passaram a ser chamados de IP (*Intellectual Property*).

A maioria dos IDEs mais recentes possuem uma interface gráfica de diagrama de blocos, onde cada bloco representa uma IP. Nesta interface é possível construir um Design de Referência utilizando a associação de IPs das bibliotecas do fabricante, ou de um repositório externo ou ainda utilizar uma IP própria, já que estes IDEs possibilitam a criação de IPs personalizadas. A Figura (25) apresenta o diagrama de blocos de um circuito lógico desenvolvido na IDE Vivado 2017.4, composto pelas seguintes IPs da Biblioteca padrão Xilinx: microprocessador MicroBlaze 10.0, bloco seu memória local, o controlador de periféricos MicroBlaze, a interface de comunicação UART, controle global de interrupções e controle de global de reset. A IP *FFT16p\_V1\_0\_0* desenvolvida individualmente utilizando código VHDL.



**Figura 25: CPU MicroBlaze e Coprocessador para FFT com Interface UART - Vivado**  
2017.4

**Fonte: Autoria Própria**

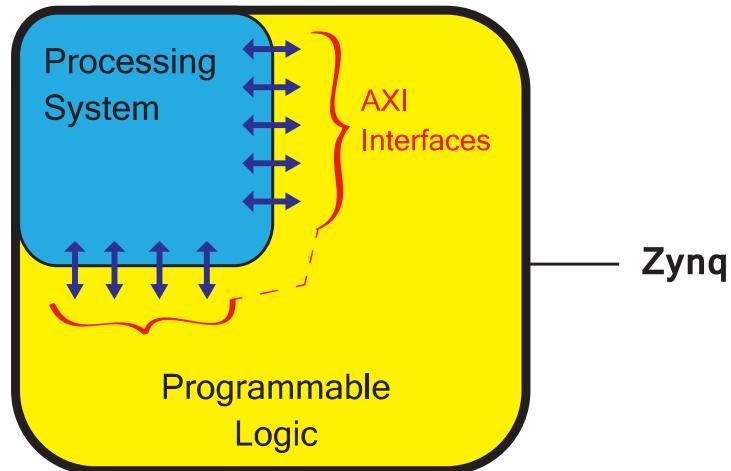
A FPGA é uma boa escolha para a implementação do algoritmo da FFT devido a grande variedade de recursos de *hardware* sintetizáveis, além de possuir recursos de programação paralela que permite o processamento paralelo de sinais, conferindo assim uma maior rapidez na execução do algoritmo (IBRAHIM *et al.*, 2016). Como afirma Meyer-Baese (2007, Prefácio), muitos algoritmos de processamento de sinais, como FFT (*Fast Fourier Transform*) e os filtros FIR ou IIR, implementados anteriormente em Circuitos Integrados de Aplicação Específica ou ASIC (*Application Specific Integrated Circuits*), agora estão sendo implementados em FPGAs.

### 2.5.2.1 ZYNQ-7000

Segundo a Crockett *et al.* (2014), Zynq-7000 é uma família de SoCs que integram a programabilidade em *software* de um processador ARM Cortex-A9, com a programabilidade em *hardware* de uma FPGA, possibilitando a integração entre CPU, DSPs e FPGA, agregando diversas funcionalidades em um único dispositivo. Zynq-7000 representa uma solução completa em processamento de sinais em um único dispositivo, com um ótima relação performance/consumo energético.

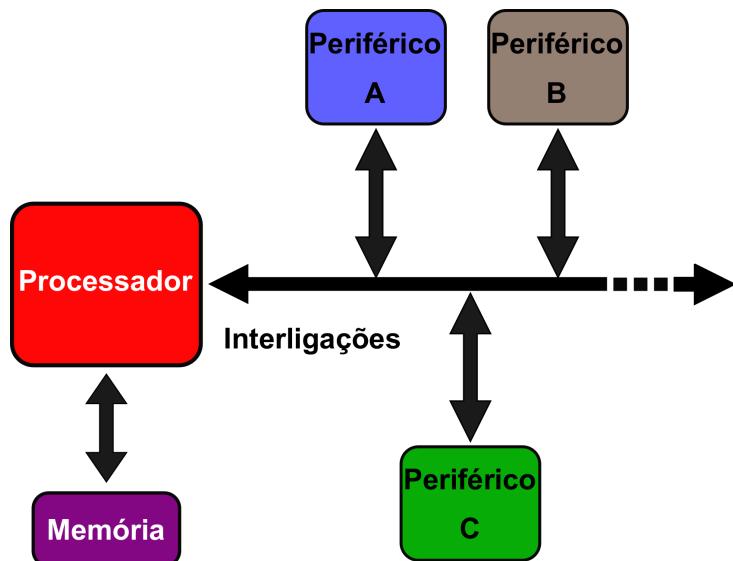
A principal característica do Zynq-7000 é a forma com que ele combina um sistema de processamento (PS - *Processing System*), formado pelo entorno do processador ARM Cortex-A9, e um sistema de lógico programável (PL - *Programmable Logic*), caracterizado como um FPGA. Processadores dedicados já tem sido utilizados em conjunto com FPGAs em diferentes aplicações, porém não da mesma forma como é feita na família Zynq-7000 Crockett *et al.* (2014, Introdução).

Para Crockett *et al.* (2014, p. 26), o PL é ideal para a implementação de operações lógicas de alta performance e sistemas de fluxo de dados contínuos. Por outro lado o PS é capaz de suportar rotinas de *software* e sistemas operacionais. Qualquer aplicação pode ser particionada em duas partes a serem implementadas uma em PL e outra em PS, a fim de se tirar proveito do melhor dos dois mundos. Porém, estas duas partes, mesmo que estando contidas dentro do encapsulamento do Zynq, como pode ser visto na Figura 26, são fisicamente distintas, e comumente estão operando em frequências diferentes. Para realizar a ponte de comunicação entre o PL e o PS, a família Zynq-7000 utiliza o padrão industrial conhecido como AXI (*Advanced eXtensible Interface*), ou interface extensível avançada. Esta interface permite estabelecer um fluxo de dados sincronizados entre PS e PL, de ambos os sentidos, suportando inclusive o disparo de interrupções através de ambos os sistemas.



**Figura 26: Arquitetura Simplificada - Zynq-7000**  
Fonte: Crockett *et al.* (2014, p. 26)

Para entender como os componentes de um sistema digital são mapeados dentro de um dispositivo Zynq, e como estes são divididos entre o PS e o PL, é necessário compreender como é a arquitetura de um sistema digital comum. Para Crockett *et al.* (2014, p. 27), o modelo básico do *hardware* de um sistemas digitais incorpora processadores, memórias, barramentos de interligação e os mais distintos periféricos, como pode ser visto na Figura (27).



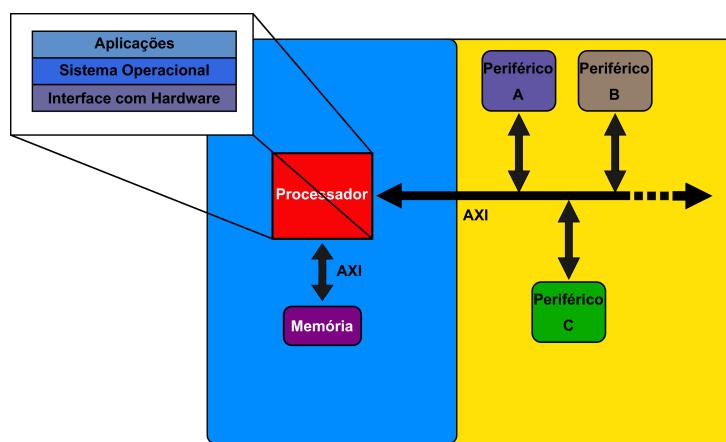
**Figura 27: Arquitetura Simplificada de um Sistema Digital**  
Fonte: Crockett *et al.* (2014, p. 27)

O processador é o elemento central deste modelo, pois é ele que executa o sistema de *software*, que compreende as camadas de mais alto nível como aplicações

baseadas em sistemas operacionais, o próprio sistema operacional, e até o nível mais baixo como o *firmware* de interface com os periféricos do *hardware*(CROCKETT *et al.*, 2014, p. 27). Já os periféricos são componentes funcionais externos ao processador, e que em geral são divididos em três tipos:

- **Coprocessadores** : Elementos que auxiliam o processador principal na realização de tarefas específicas, geralmente projetados para otimizar tal tarefa.
- **Interfaces de Comunicação**: Elementos responsáveis pela interação com interfaces externas, acionando gatilhos ou controlando portas digitais. Utilizando muitas vezes protocolos específicos de comunicação como UART ou SPI.
- **Elementos Adicionais de Memória** : Elementos exclusivamente destinados ao armazenamento de dados.

A Figura (28) apresentam a mesma arquitetura simplificada da Figura (27), porém mapeado para um dispositivo Zynq. A estrutura do sistema digital é dividida entre processador e memória para o lado PS, e os demais possíveis periféricos para o lado PL. Do lado PS, a arquitetura é fixa, obedecendo a estrutura definida pelo fabricante, em total contraponto com o lado PL. No lado PS, a estrutura é totalmente flexível, limitada apenas pelo número de CLBs disponíveis na FPGA, o que oferece ao desenvolvedor um ambiente de caixa de areia para construir qualquer tipo de periférico.

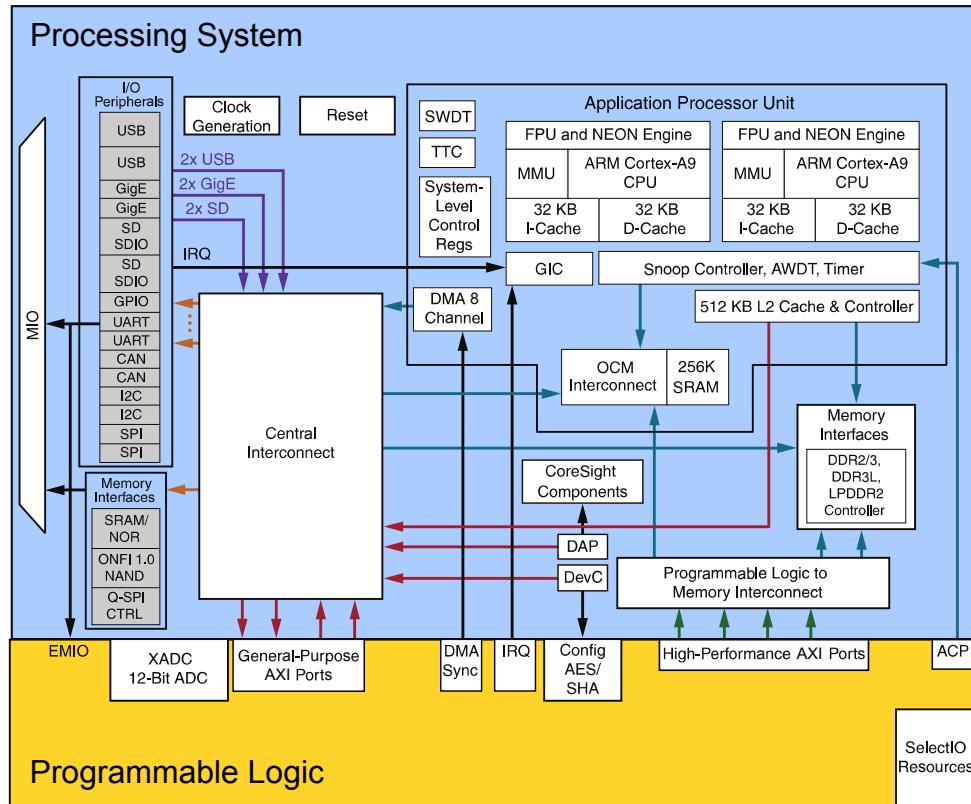


**Figura 28: Arquitetura Simplificada do um Sistema Digital Mapeado para o Zynq**  
Fonte: Adaptado de Crockett *et al.* (2014, p. 27)

A interface AXI possui uma grande importância no desenvolvimento no Zynq-7000, já que por meio desta interface é que os periféricos em PL se comunicam

com o processador em PS. A interface AXI faz parte da família de barramentos para microcontroladores ARM AMBA (*Advanced Microcontroller Bus Architecture*). O ARM AMBA é um protocolo *Open Standard* para conexões e gerenciamento de blocos funcionais dentro de dispositivos *Systens-on-Chip* (Soc), facilitando o desenvolvimento de designs com múltiplos processadores, e com grande número de controladores e periféricos (ARM, 2018). A família de SoCs Zynq-7000 utilizam a interface AXI versão 4, o qual obedece ao mais recentes padrões ARM AMBA 4.0 (XILINX, 2017).

Dentro do ambiente do PS no Zynq, além do processador ARM Cortex-A9, existe ainda um conjunto de periféricos de memória, interconexão, comunicação e gerenciamento. A maioria destes periféricos utiliza a interface AXI, com 32 ou 64 Bits, inclusive a próprias conexões de fronteira entre PS e PL. Diferente de PS, onde a arquitetura já está pronta e é estática, em PL o desenvolvedor pode criar todo uma gama de periféricos, e consequentemente conectá-los com uma quantidade enorme de interfaces AXI de qualquer tipo dos já citados. Porém, a fronteira entre PS e PL é limitada e representa um gargalo na interação entre os dois lados. Os dispositivos da família Zynq-7000 possuem 2 interfaces AXI Mestre de 32-Bits, 2 interfaces AXI Escravo de 32-Bits e 4 interfaces 64-Bit/32-Bit configuráveis de alta velocidade. A Figura (29) apresenta uma visão geral da arquitetura do SoC Zynq-700, mostrando as conexões AXI dentro de PS e também as conexões na fronteira de PL.



**Figura 29: Visão Geral da Arquitetura - Zynq 7000**

**Legenda:** AXI 32-Bits/64-Bits, AXI 64-Bits, AXI 32-Bits, AHB 32-Bits, AXI 32-Bits

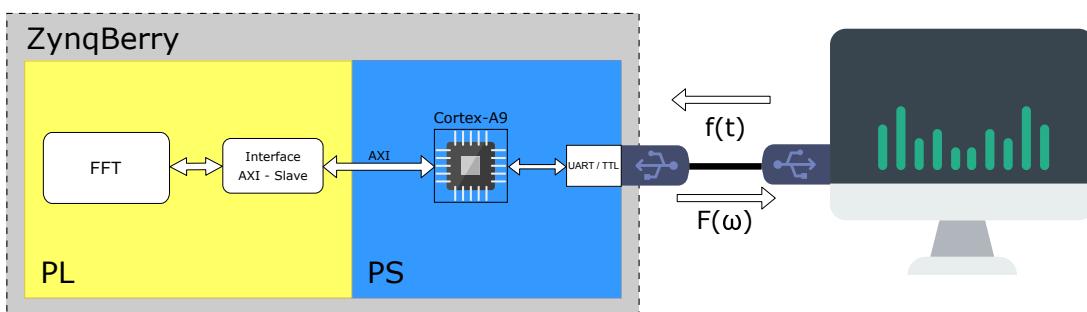
**Fonte:** Adaptado de (XILINX, 2018)

### 3 MATERIAIS E MÉTODOS

Este capítulo destina-se a apresentação dos dispositivos e programas, bem como os algoritmos, funções e metodologias utilizadas para projetar e implementar os circuitos de cálculo da FFT. Todos os passos apresentados aqui foram embasados na teoria apresentada no Capítulo (2).

O desenvolvimento de um *hardware* para cálculo de uma FFT em FPGA, abrindo mão de IPs prontas e blocos de DSPs, utilizando apenas as bibliotecas padrão de componentes, como a *IEEE 1164* e a *UNISIM*, disponibilizadas pelo fabricante, é uma tarefa que exige um projeto e implementação eficiente. Para o projeto da arquitetura da FFT, é necessário ter conhecimento de toda a base matemática, tanto da Transformada de Fourier, quanto do algoritmo CORDIC e de suas variantes, para que se possa tirar o máximo proveito das simplificações e otimizações matemáticas possíveis. Na implementação do algoritmo, um design eficiente dos diferentes componentes que formam o *hardware*, além de reduzir a latência dos sinais, também reduz o consumo de Flip-Flops, LUTs, *Muxes* e blocos de memória. Tornando, assim, possível a implementação de um *hardware* mais eficiente dentro das restrições de recursos da FPGA.

Para que fosse possível testar as funcionalidades e o desempenho após a implementação, desde o inicio do projeto da FFT, fora elaborado o diagrama da Figura (30).



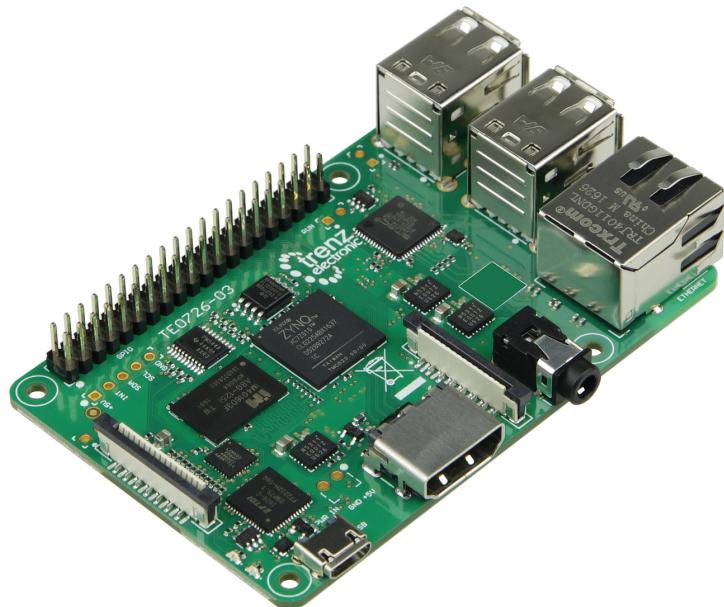
**Figura 30: Diagrama Geral do Sistema Implementado**  
Fonte: Autoria Própria

Como pode ser percebido na Figura (30), no sistema desenvolvido neste

trabalho os dados de entrada da FFT, que pode ser dados amostrados ou simplesmente gerados por *software*, são enviados por um computador via USB, pelo protocolo de comunicação UART - TTL, para dentro da parte PS da placa ZynqBerry. Os dados são encaminhados pelo processador Cortex-A9 para o endereço do *hardware* da Interface AXI, a qual finalmente disponibiliza os dados de entrada ao módulo da FFT implementada. Após computado, os dados de saída da FFT fazem o caminho reverso, para então serem enviados ao computador via UART. Assim, este é o diagrama geral utilizado como guia, ao longo de todas as etapas do desenvolvimento, que estão apresentados nas próximas seções.

### 3.1 ZYNQBERRY TE0726-03M

O dispositivo escolhido para a implementação do algoritmo Radix-2 é o *ZynqBerry - TE0726* da fabricante *Trenz Eletronic®*, apresentado na figura (31). Este dispositivo é baseado no SoC (*System On Chip*) Raspberry Pi modelo 2, e vem equipado com uma FPGA SoM (*System on Module*) XC7Z010-1CLG225C-REV3, da família Zynq-700, fabricada pela *Xilinx®* (ELETRONIC, 2018).



**Figura 31: SoC ZynqBerry - TE0726**  
Fonte: Eletronic (2018)

A FPGA XC7Z010-1CLG225C tem a mesma arquitetura dos modelos da família Artix-7, também da Xilinx, e contém recursos como: 28.000 células lógicas, 17.600 LUTs, 2.1 Mb de memória RAM divididos em blocos de 26Kb e um total de

36.200 *flip-flops*. Cada CLB, nesta FPGA, para implementar diferentes operações lógicas, utiliza 16 *flip-flops*, 2 somadores de 4 *bits* cascateáveis e ainda 8 LUTs. Sendo possível configurar a memoria RAM das LUTs para 64x1 ou 32x2 bits, ou ainda como um *shift register (SRL)*. Além disso, esta FPGA possuir 80 blocos DSP, cada um equipado com um multiplicador simples de 18x25, e um somador/acumulador de 48 bits (XILINX, 2018). Todas estes recursos fazem do XC7Z010 um *hardware* competente para as mais diversas aplicações de processamento de sinais, como o calculo da FFT.

O processador utilizado no ZynqBerry TE0726-03M é um dual-core ARM Cortex-A9 de 866MHz, que proporciona um desempenho significativo em Sistemas Operacionais (S.O.), como o sistemas baseados em kernel Linux, que podem incluem interface gráfica sofistica. Cada núcleo deste processador conta ainda uma unidade NEON™Media Processing Engine (MPE), para alto desempenho em codificação e decodificação de áudio e vídeo, e uma unidade de ponto flutuante para incremento da precisão em operações matemáticas (XILINX, 2018). Aliando a versatilidade da XC7Z010 e o poder de processamento do ARM Cortex-A9, é possível construir um sistema que execute uma versão do S.O. Linux, tirando proveito de toda a funcionalidade de tal sistema operacional pode prover, e que ainda disponha de um *hardware* acelerador personalizado para aplicações específicas, trabalhando com paralelismo em alta frequênci. Provendo assim uma solução engenhosa de alto desempenho para processamento de sinais.

Para a implementação das FFTs realizadas neste trabalho, utilizando o XC7Z010-1CLG225C, foi necessário realizar a programação lógica da FPGA (PL), implementando os circuitos lógicos da arquitetura, e ainda programar o Sistema de Processamento (PS), com as rotinas de envio e recebimento de dados Via UART. O processo de programação de ambas partes PL e PS, podem ser vistas no Apêndice (A)

## 3.2 IMPLEMENTANDO O PROCESSADOR CORDIC

Como apresentado na Seção (2.4), devido as limitações em disponibilidade de blocos multiplicadores nas FPGAs, se faz necessário utilizar o algoritmo Cordic implementado em *hardware* para realizar as operações de rotacionamento do vetor  $H_r$  pelo ângulo descrito por  $W_{N_o}^r$ .

A FPGA XC7Z010-1CLG225, presente no ZynqBerry, possui no total 80 blocos DSP (*Digital signal processing*), dentro de cada um deles há um *hardware*

multiplicador. Tais multiplicadores seriam suficientes para implementar a função de rotação de vetores, necessária ao cálculo da FFT. Comumente em aplicações de processamento de sinais, o módulo de calcula da FFT é apenas umas das etapas de processamento de sinais. Logo, utilizar todos estes blocos de DSPs para computar a FFT tornaria a implementação de outras tarefas mais difíceis. Outro ponto importante é que implementar a FFT, utilizando apenas blocos lógicos comuns, torna a implementação mais generalista.

Nesta etapa é realizado o projeto e implementação em FPGA do processador Cordic, utilizado para montar o bloco da FFT. Tal processador segue a metodologia do algoritmo MSR Cordic apresentado na Seção (2.4.3).

### 3.2.1 PROJETO DOS PARÂMETROS CORDIC

Como apresentado na Seção (2.4.3), a determinação dos parâmetros Cordic  $\mu_i$ ,  $\mu_j$ ,  $s_i$ ,  $s_j$ ,  $I$  e  $J$  em (85), é feita com base na minimização do erro de aproximação  $\epsilon$ , descrito em (99). Para encontrar o conjunto ótimo de parâmetros CORDIC é possível utilizar uma variedade de algoritmos determinísticos, já que este sistema se assemelha a problemas clássicos de otimização.

Na Seção (2.4.2.1) fora apresentado o algoritmo de otimização TBS, o qual se destinava a determinação do conjunto de parâmetros CORDIC para o EEASR. O EEASR possuía um conjunto de ângulos elementares menores do que o MSR, e a função de otimização não inclui a redução do erro de  $K_c$  em relação a unidade, já que a compensação deste ganho era realizada em uma etapa independente através de uma pseudo-rotação. Logo, fora decidido adaptar o algoritmo TBS, corrigindo a equação de minimização para o caso do MSR e, consequentemente, incluir no conjunto de solução os ângulos elementares do MSR.

Segue abaixo o pseudo-código do TBS adaptado, utilizado para obter o conjunto ótimo de parâmetros CORDIC para o MSR. O vetor  $r_\theta$  representa o conjunto de todos ângulos elementares, e  $r_p$  presenta o conjuntos dos ganhos  $K_n$ , ambos oriundos das diferentes combinações entre os parâmetros CORDIC.

```

1 % Inicialização
2  $\phi_\theta(1, k) = r_\theta(k)$  para todo  $k$ ,
3  $\phi_p(1, k) = r_p(k)$  para todo  $k$ ,
4
5 %Acumulação

```

```

6   FOR i = 1 to N - 1
7     FOR k = 1 to Z(S2)
8       Encontra k' tal que :
9        $\epsilon = \min \sqrt{(\phi_\theta(i, k') + r_\theta(k) - \theta)^2 + (\phi_p(i, k^*) * r_p(k) - 1)^2} : 1 \leq k^* \leq Z(S_3)}$ 
10       $\phi_\theta(i + 1, k) = \phi_\theta(i, k') + r_\theta(k)$ 
11       $\phi_p(i + 1, k^*) = \phi_p(i, k^*) * r_p(k)$ 
12    END
13  END
14
15 %Determinação do Ótimo Global
16 Encontra k* tal que
17  $\epsilon = \min \sqrt{(\phi_\theta(N, k^*) - \theta)^2 + (\phi_p(N, k^*) - 1)^2} : 1 \leq k^* \leq Z(S_3)$ 
18 Result(N) = K*
19
20 %Determinação do Caminho Solução
21 FOR i = N to 2
22   Encontra k' tal que
23    $\epsilon = \min \sqrt{(\phi_\theta(i - 1, k') + r_\theta(k) - \theta)^2 + (\phi_p(i - 1, k^*) * r_p(k) - 1)^2} : 1 \leq k^* \leq Z(S_3)}$ 
24   k* = k'
25   Result(i - 1) = k'
26 END
27
28

```

**Código 3.1: Algoritmo TBS Adaptado para MSR****Fonte : Adaptado de Wu *et al.* (2003)**

Uma atenção especial é dada para o vetor de ângulos elementares  $r_\theta$ . Este vetor é preenchido com os ângulos obtidos pelo conjunto de combinações possíveis dos parâmetros CORDIC do MSR, através de (93), o qual é reescrito como:

$$\alpha = \sum_{i=1}^I \mu_i 2^{-s_i(n)} \quad (106)$$

$$\beta = \sum_{j=1}^J \mu_j 2^{-s_j(n)} \quad (107)$$

$$r_\theta = \tan^{-1} \left( \frac{\beta}{\alpha} \right) \quad (108)$$

$$: \mu_i, \mu_j \in \{-1, 0, 1\} \quad (109)$$

$$s_i, s_j \in \{0, 1, \dots, S\} \quad (110)$$

A partir das diferentes combinações de parâmetros, uma variedade de possibilidades de ângulos preenchem o vetor  $r_\theta$ . Porém há um problema com o cálculo do arco tangente para combinações onde  $(\alpha < 0, \beta < 0)$  e  $(\alpha > 0, \beta < 0)$ . Pois, nestes casos as operações de rotação deslocariam o vetor para o 4º e 3º quadrantes respectivamente, o que deveria resultar em valores de ângulos maiores que  $\pi/2$ . Porém, o arco tangente considera apenas valores dentro do intervalo  $(-\pi/2, \pi/2)$ . Para corrigir tal efeito basta aplicar a seguinte condição em  $r_\theta$ :

$$\text{Se } \alpha < 0 \text{ e } \beta < 0 : \quad r_\theta = \tan^{-1} \left( \frac{\alpha}{\beta} \right) - \frac{\pi}{2} \quad (111)$$

$$\text{Se } \alpha < 0 \text{ e } \beta > 0 : \quad r_\theta = \tan^{-1} \left( \frac{\alpha}{\beta} \right) + \frac{\pi}{2} \quad (112)$$

$$(113)$$

Alguns parâmetros do MSR são determinados de acordo com as limitações de disponibilidade de *hardware*, como é o caso do número de iterações Cordic  $N$  e o número de termos SPT  $N_{SPT}$ . Para estes valores fora observado a recomendação usual na literatura, em (LIN; WU, 2005), (KUO *et al.*, 2003) e (PARK; YU, 2012).

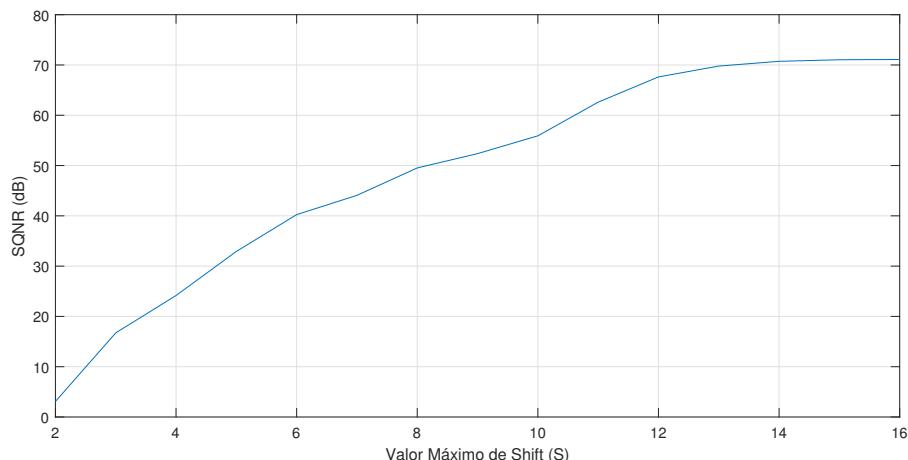
O número de iteração CORDIC ( $N$ ), impacta diretamente no número de ciclos de *clock* necessários para realizar a operação de rotacionamento, impactando no erro final  $\epsilon$ . Como observado em Lin e Wu (2005), é admissível tomar  $N$  como 3, e ainda obter um bom nível de SQNR.  $N_{SPT}$  também impacta no desempenho do erro  $\epsilon$ , mas afeta ainda no consumo de *hardware*, já que mais termos SPT significa mais operadores de deslocamento de *bit*. Como a intenção é montar uma FFT de bom desempenho, com o maior número de pontos possível,  $N_{SPT} = 3$  é admissível baseado no que é visto em Lin e Wu (2005).

A escolha do número máximo de deslocamento de *bits* ( $S$ ), implica na utilização de *barrel shifters* maiores, e também impacta no aumento do volume de memória necessária para guardar os conjuntos de parâmetros CORDIC, utilizados durante as operações de rotação. Quanto maior for  $S$ , mais liberdade é dado ao conjunto de ângulos elementares, e mais fácil é encontrar conjuntos que reduzam  $\epsilon$  dentro das restrições.

A arquitetura da FFT implementada neste trabalho é pensada de modo que o fluxo de dados de entrada possam vir do Sistema de Processamento (PS), mas também possilite a entrada de dados advinda de um ADC Flash de 12 *bits*. Portanto as palavras binárias utilizadas para representar os sinais precisam ter no mínimo 12

*bits*. Para evitar a ocorrência de *overflow*, em uma estrutura de 1024 pontos, onde há 10 níveis, apenas um ponto de soma a cada nível, e os ganhos são próximos da unidade, é adequado utilizar 16 *bits* para a representação de sinais.

Para determinar um valor para  $S$ , foi implementado o Algoritmo (3.2.1) com auxílio do *software Matlab®*. Para cada valor de  $S \in \{1\dots16\}$  foram criados conjuntos de parâmetros CORDIC, fixando  $I = 1$  e  $J = 2$  (Modo Normal). Com base nos parâmetros gerados, foram realizadas operações de rotacionamento de vetores nos moldes de (85), e obtidos os valores de SQNR para cada valor de  $S$ , a fim de medir o impacto que este parâmetro tem no desempenho do algoritmo. O resultado deste teste é expresso na Figura (32).



**Figura 32: Relação entre  $S$  e o SQNR Médio para MSR Cordic Modo Normal**  
Fonte: Autoria Própria

Os conjuntos parâmetros  $s_i$  e  $s_j$  para cada iteração do algoritmo serão armazenados em uma ROM de controle de cada Unidade CORDIC. O número de *bits* necessários para armazenar estes parâmetros é dado por  $\log(S)$ . Como visto na Figura (32), escolher  $S = 8$  promove um desempenho SQNR de 50dB, e se faz necessário armazenar apenas 3 *bits* para cada elemento  $s_i$  e  $s_j$ . Porém, ao utilizar 4 *bits* para o mesmo fim, é possível fazer  $S = 15$ , e alcançar um desempenho médio de 70dB. Logo, toma-se  $S = 15$ , para alcançar o máximo desempenho possível.

Com citado na Seção (2.4.3.1), os parâmetros  $I$  e  $J$  podem ser constantes independentes da operação de rotação (Modo Normal), ou podem variar a cada iteração (Modo Generalizado). Com base no mesmo algoritmo (3.2.1), foi incluído no vetor de ângulos elementares  $r$  as combinações de parâmetros possíveis quanto  $N_{spt} = 4$ , e  $(I = 1, J = 2)$  e  $(I = 2, J = 1)$ . Então, foram gerados os conjuntos ótimos

de parâmetros CORDIC, para estas combinações. Em seguida, foram simulados as operações de rotação para um conjunto de ângulos de rotação  $[0 : 0, 703125^\circ : 360^\circ]$ . A Tabela (3), apresenta nível do SQNR encontrado nestas simulações.

Modo de Operação	$I$	$J$	$S_{max}$	SQNR
Normal	1	2	15	83,9786dB
Normal	2	1	15	78,52dB
Generalizado			15	92.9786 dB

**Tabela 3: Nível SQNR entre o Modo Generalizado e Normal,**

$N_{SPT} = 3$

**Fonte:** Autoria Própria

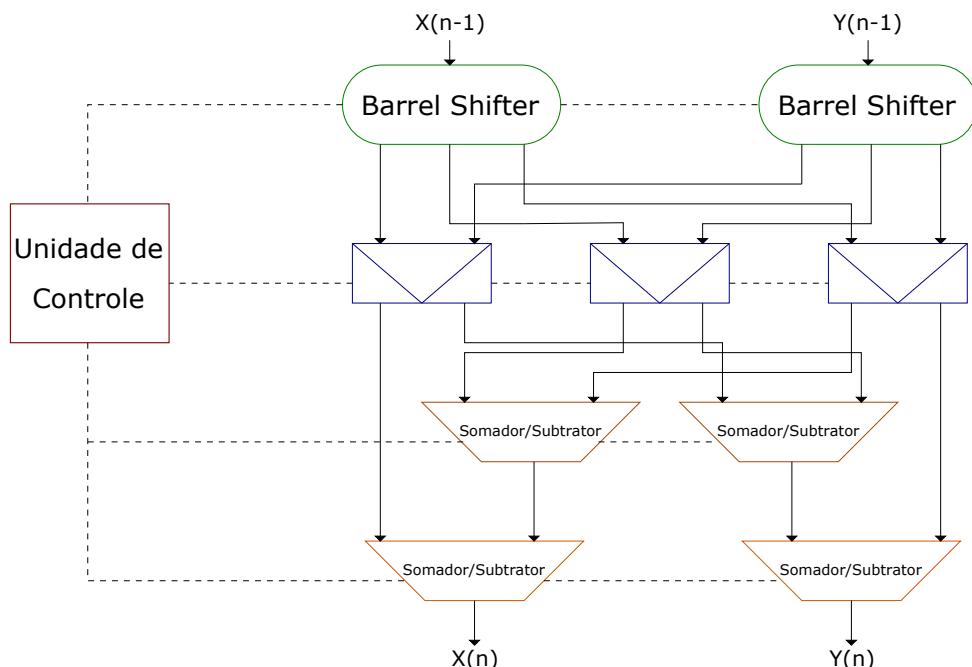
O Modo Generalizado, para a simulação proposta, apresentou um valor médio de SQNR de 92.97 dB. Já o Modo Normal, para o mesmo método de simulação, apresentou um valor de 83.97 dB. Segundo (LIN; WU, 2005), para armazenar os parâmetros  $\{\mu_i, \mu_j\} \in \{-1, 0, 1\}$ ,  $\{s_i, s_j\} \in \{0, 1, \dots, 6\}$  de uma única interação do Algoritmo MSR CORDIC, são necessários  $(\log(S) + 2)N_{SPT}$  bits para o modo Normalizado e  $(\log(S) + 3)N_{SPT}$  para o Modo Generalizado. Ou seja o impacto do inserção dos *switches*, necessários no caso do Modo Generalizado, em termos de implementação é apenas a inclusão de mais um bit no conjunto de dados de cada iteração. Portanto, isso Justifica a escolha da implementação do MSR Cordic no Modo Generalizado. Assim os dados referentes aos parâmetros escolhidos com base no Algoritmo (3.2.1), para o MSR Cordic Modo Generalizado, com  $N_{SPT} = 3$  e  $S = 16$ , são armazenados na forma binária em uma componente ROM, individual a cada unidade CORDIC.

### 3.2.2 ARQUITETURA CORDIC IMPLEMENTADA

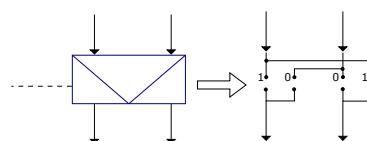
Após a definição dos parâmetros CORDIC, é então realizada a implementação da arquitetura em FPGA, através da linguagem VHDL. Segundo Lin e Wu (2005), as operações de multiplicação dos termos SPT em (85) podem ser realizadas utilizando operadores lógicos de deslocamento de *bit*, ou *shifters*. Como observado em (85), as duas partes  $x$  e  $y$  do sinal de entrada precisam ser deslocados para direita de forma diferenciada, com base nos parâmetros  $I$  e  $J$ , gerando assim cada parte  $N_{SPT}$  sinais deslocados.

As operações de deslocamento de  $x$  e  $y$  podem ser realizadas utilizando apenas dois *Barrel shifters*, ambos com uma entrada e três saídas. O *Barrel shifter* é um componentes lógico capaz de deslocar um palavras binária por um número específico de *bits* utilizando apenas lógica combinacional, o que possibilita efetuar a operação de deslocamento em apenas um ciclo de *clock*. A única desvantagem deste componente é o número elevado de multiplexadores necessários para sua implementação:  $n \log_2 n$ , onde  $n$  é o tamanho da palavra binária. Porém, como neste projeto  $n = 16$  *bits*, admitiu-se o custo de inclusão de 64 multiplexadores em prol de se obter o melhor desempenho na operação base do algoritmo CORDIC.

Para a implementação em VHDL fora utilizado o diagrama de interação sugerido por Lin e Wu (2005), o qual segue na Figura (33).



**Figura 33:** Arquitetura da Iteração MSR Cordic Modo Normal  $N_{spt} = 3$   
**Fonte:** Adaptado (LIN; WU, 2005)



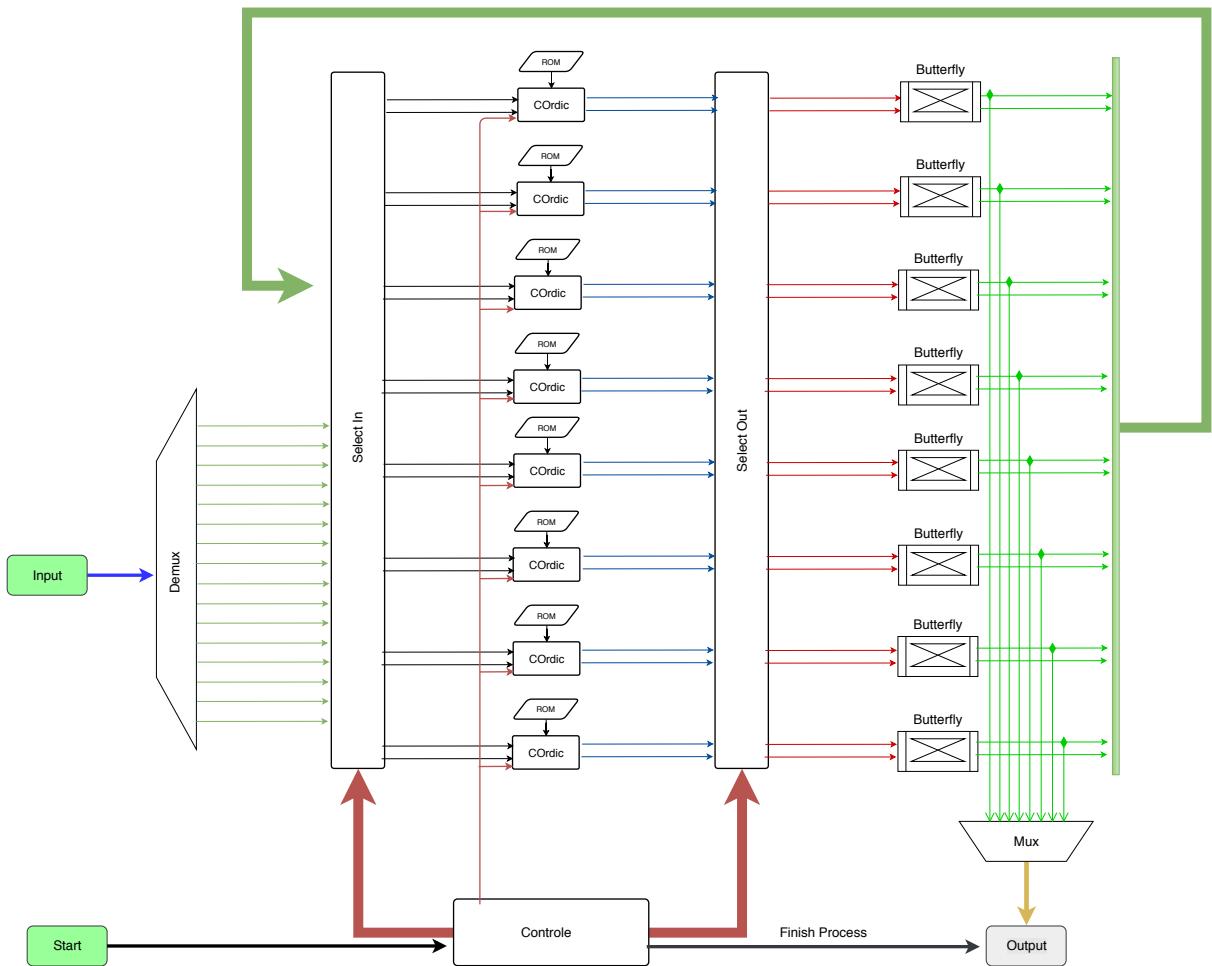
**Figura 34:** Arquitetura  
**Switch 2x2**  
**Fonte:** Adaptado (LIN; WU,  
2005)

Como pode ser visto na Figura (33), a implementação da interação CORDIC é feita basicamente de um conjunto de 4 somadores e 2 *Barrel Shifters*. A implementação do Algoritmo CORDIC pode ser feita na forma sequencial ou na forma *pipeline*. Na forma sequencial, a Unidade de Controle da Figura (33), por meio de um conjunto de *flip-flops*, armazenam o valor final da interação e retro alimentam a entrada do circuito, durante os  $N$  ciclos de interação CORDIC. A cada interação a Unidade de Controle muda os parâmetros de acordo com os dados presentes na ROM. Na forma de *pipeline* várias unidades CORDIC, como as vista Figura (33), são encadeadas sequencialmente de modo que cada unidade seja responsável por uma interação separadamente, o que possibilita a operação ser realizada em apenas um ciclo de *clock*.

Na forma sequencial a implementação do Algoritmo ocupa menos recursos da FPGA, já que são necessários apenas um hardware de interação, e *flip-flops* de controle. Apesar disso, o número de ciclos de *clock* necessários para concluir a operação é ditada pelas operações sequenciais da Unidade de Controle, que nunca será inferior a  $N$  ciclos. Na forma *pipeline*, o número de ciclos de *clock* necessários para finalizar a operação de rotação é dependente apenas da velocidade de propagação dos sinais, através do hardware encadeado das iterações. No entanto, como neste modo existem  $N$  hardwares de interação encadeados, o consumo de recursos da FPGA é bem maior. Por questão de limitação de recursos da FPGA utilizada, se optou pela implementação na forma sequencial.

### 3.3 IMPLEMENTANDO A FFT 16 PONTOS

Após implementado o processador CORDIC, é necessário montar uma primeira versão da FFT, para que seja possível testar o desempenho do processador, e o consumo de recursos da arquitetura implementada. Então, foi escolhido montar uma FFT de apenas 16 pontos, o que demanda o uso de 8 unidades CORDIC, que serão suficientes para executar as 8 operações de rotacionamento vetorial por nível. A Figura (35) apresenta o diagrama da arquitetura implementada.



**Figura 35: Arquitetura Implementada FFT de 16 Pontos**

Fonte: Autoria Própria

Como visto na Figura (35), para receber os dados de entrada serializados, o *Input* da FFT possui um *Demux*, assim como a porta (*Output*) possui um *Mux*, para devolver os dados computados. Para transferir tais dados seriais via AXI, o qual realiza a ponte entre o PS e o PL, é necessário implementar tal interface para testar a FFT. A implementação do módulo AXI, utilizado como interface para as implementações de hardware deste trabalho, podem ser vistas no Apêndice (B).

Ainda na Figura (35), cada unidade CORDIC possui uma ROM, que armazena os conjuntos de parâmetros otimizantes, os quais foram previamente calculados por (3.2.1). A unidade *Select Out* é responsável por ordenar, a cada iteração, as saídas das unidades CORDIC para a ordem adequada de entrada das unidades *Butterfly*, de acordo com o diagrama da FFT por decimação na frequência, apresentado em (12). A unidade *Select In*, na Figura (35), realiza a seleção da entrada de sinais nas unidades CORDIC, escolhendo entre o canal do *Demux* dos dados de entrada, e o *Feedback* da saída das unidades *Butterfly*.

A arquitetura da Figura (35) foi concebida de forma que a unidade *Controle*, após o recebimento do sinal de *Start*, acione as unidades CORDIC, iniciando a operação de rotação de vetores. Após 3 ciclos de *Clock*, os sinais de saída são direcionados pelo *Select Out*, controlado pela Unidade *Controle*, para as unidades *Butterfly* que retroalimentam o bloco *Select In*. Este processo, que dura 3 ciclos de *clock*, caracteriza o cálculo de um nível da FFT. Como uma FFT de 16 pontos possui 4 níveis, a unidade de Controle realiza 4 iterações, retroalimentando as unidades CORDIC por meio do *Select In*.

Ao final do processo de iteração dos 4 níveis da FFT, a unidade de Controle aciona o sinal de *Finish Process*, sinalizando a Interface AXI que os dados já foram computados, e já podem ser transferidos ao PS de forma serial. Um ponto importante aqui, é que se faz necessário transferir apenas metade dos sinais computados, pois o espectro de Fourier calculado é simétrico. Logo, são transferidos os 8 primeiros pontos da FFT, obedecendo a ordem *Bit-Reverse*, e compensando o módulo dos sinais, transferindo os 15 *bits* mais significativos de cada ponto.

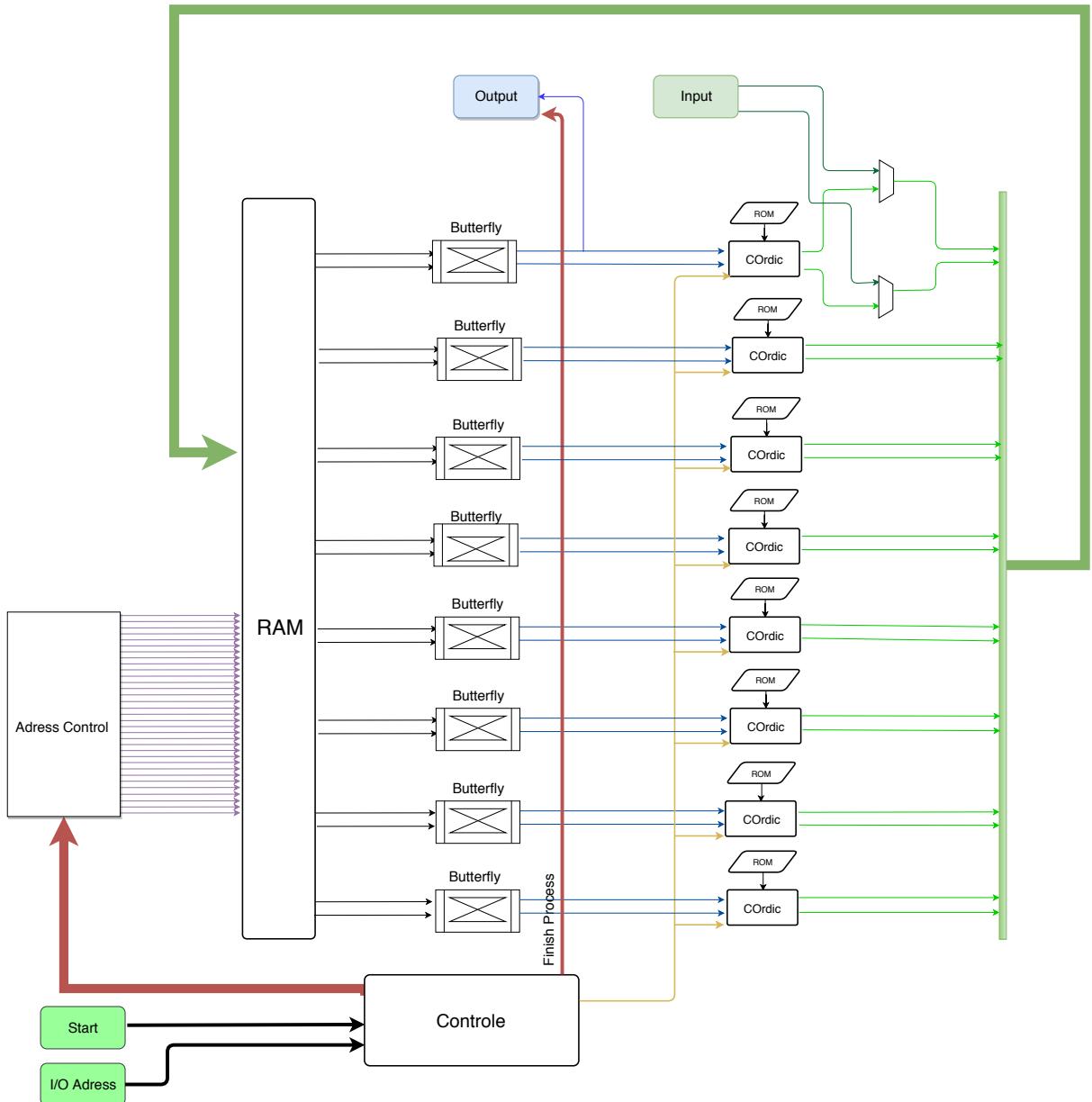
### 3.4 IMPLEMENTANDO A FFT 1024 PONTOS

Na bibliografia geralmente é mais comum encontrar implementações de algoritmos para cálculo da FFT utilizando implementações com 1024 pontos. Portanto, com o intuito de realizar uma comparação de desempenho, em um nível próximo dos artigos mais recentes da área, é desenvolvido uma nova arquitetura para a FFT de 1024 pontos.

A arquitetura implementada para a FFT de 16 pontos possui um processamento paralelo em relação aos módulos de cálculo CORDIC. Desse modo, que todas as operações de rotação de vetores, para um determinado nível, são realizadas paralelamente. Para a implementação de uma FFT com 1024, ou mesmo 512 pontos, não seria possível utilizar a mesma arquitetura paralela em relação as unidades CORDIC, uma vez que o número de conexões, *mux* e *Flip-flops* presentes na FPGA não seriam suficientes para tal tarefa.

A solução adotada para montar a FFT de 1024 pontos foi realizar as operações CORDIC de cada nível da FFT de modo fracionário. Uma FFT de 1024 pontos possui ao total 10 níveis de cálculo ( $\log_2 N$ ), e 512 operações CORDIC por nível. Se for utilizado 8 unidades CORDIC em paralelo, para calcular conjuntos de 8 operações por iteração, é possível realizar o cálculo de um nível reutilizando as unidades CORDIC 64

vezes. Porém, para esta tarefa é necessário utilizar um bloco RAM para armazenar os 1024 pontos e permitir o acesso a 16 pontos a cada iteração. A Figura (36) apresenta um diagrama da arquitetura implementada para a FFT de 1024 pontos.



**Figura 36: Arquitetura FFT 1024 Pontos Implementada**  
Fonte: Autoria Própria

A FFT de 16 pontos não possui nenhum bloco de memória RAM (*BRAM*), utilizando apenas registradores para operar as iterações, o que a torna menos exigente em termos de recursos da FPGA. Já uma FFT de 1024 requer não apenas uma bloco comum de RAM, mas uma memória de acesso múltiplo, já que se faz necessário ler e escrever em 16 pontos diferentes a cada iteração. Pois, caso contrário, seria ne-

cessário transferir cada ponto da FFT individualmente para cada unidade CORDIC em paralelo, reduzindo assim o desempenho da FFT. Para criar algo como um bloco de RAM com 16 canais, foram criados 16 blocos de memória RAM *dual port*, cada um com 128 endereços de memória. A entrada e saída de dados destes blocos é ligado a uma rede de multiplexadores, os quais ordenam os dados a serem armazenados em blocos específicos. Como a cada nível da FFT o conjunto de dados a serem calculados devem ser divididos em 64 blocos, para que as 8 unidades CORDIC processem 1 bloco por vez, se faz necessário ordenar a entrada de dados nos blocos de RAM de modo que os dados estejam na posição certa para o próximo nível.

No diagrama da Figura (36), a FFT de 1024 pontos, além de dispor de uma unidade *RAM* de múltiplos acessos, ainda contém um bloco *Adress Control*, o qual é responsável por endereçar os resultados das operações CORDIC para o espaço de memória adequado no bloco *RAM*. A entrada de dados nesta arquitetura é feita por meio do bloco *Input*. Este bloco separa os dados seriais de reais de 32 *bits* oriundos de PS, em dois sinais reais de 16 *bits*, e por meio de dois multiplexadores, insere esses dados nas duas primeiras portas de dados do bloco *RAM*. Enquanto a interface AXI estiver repassando ao módulo da FFT os dados de entrada, e endereçando estes por meio da porta *I/O Adress*, o bloco de *Controle* vai preenchendo a memória *RAM* com os dados de entrada.

Após preencher todos os dados da *RAM*, a interface AXI dispara a porta *start*, e a operação da FFT é iniciada. Ao final, o bloco de *Controle* aciona a porta *Finish Process*, provocando uma interrupção em PS, notificando sobre o fim da operação. Então, os dados são transferidos ao PS, de acordo com o endereço requisitado pela interface AXI na porta *I/O Adress*, repassando os dados pela primeira porta de leitura do bloco *RAM*.

## 4 RESULTADOS E DISCUSSÃO

Neste capítulo serão apresentados os resultados oriundos dos teste realizados com os circuitos implementados no capítulo anterior. Para executar tais testes, o PS do Zynqberry não foi programado utilizando o suporte de um sistema operacional, ou seja a programação foi feita a nível *bare metal*, sendo utilizado apenas as bibliotecas do *Board Suport Package - PS7* (BSP), destinado a processadores Cortex-A9. Para que fosse possível gerar conjuntos de sinais de entrada e analisar por meio gráfico os dados de saída da FFT, fora utilizado a funcionalidade de comunicação serial do *Software Matlab*. Assim todos os dados gráficos apresentados neste capítulo foram obtidos com auxilio desta ferramenta.

### 4.1 FFT DE 16 PONTOS

A Tabela (4) apresenta os dados referentes ao volume de recursos utilizados pela implementação da FFT de 16 pontos na FPGA XC7Z010-1CLG225C do kit ZynqBerry, após o processo de síntese do Vivado HLx 2017.4.

Recurso	Utilização	Utilização %
LUT	5760	32,73
FF	4895	13,91
BRAM	0	0
DSP	0	0

**Tabela 4: Resultado de Síntese FFT 16 Pontos**

**Fonte: Autoria Própria**

Em termos de consumo de recursos da FPGA, é possível fazer um paralelo dos dados entrados na bibliografia, com os obtidos pela implementação da FFT de 16 pontos. A Tabela 5 traz essa comparação.

Referência	LUT	FF	BRAM	DSP	Wordlength( <i>bits</i> )
Autoria Própria	5760	4895	0	0	16
Santhosh e Thomas (2013)	2127	1572	0	14	9

**Tabela 5: Comparativo de Síntese - FFT 16 Pontos**

**Fonte: Autoria Própria**

Como é possível observar na Tabela (5), o consumo de recursos da FFT implementada é o dobro da apresentada por Santhosh e Thomas (2013). Isso deve principalmente ao número de *bits* utilizados para representação de sinais e, também, ao fato de que na FFT implementada existe 16 blocos CORDIC, os quais não são necessários em Santhosh e Thomas (2013), pois este utiliza blocos DSPs.

Para efetuar uma operação de rotacionamento vetorial, na forma de (46), em apenas um ciclo de clock se faz necessário utilizar 4 multiplicadores. Assim em FPGAs como a XC7Z010-1CLG225C, onde há um multiplicador por bloco DSP, seriam necessários 64 destes blocos. Blocos DSP são recursos importantes para operações de processamento de sinais, e normalmente estão presentes nas FPGAs em um número bastante reduzido, se comparado ao número de células lógicas. Como por exemplo, a FPGA utilizada neste trabalho possui 28 mil células lógicas, e apenas 80 blocos DSP. Logo, utilizar uma arquitetura de FFT que não necessite de blocos DSPs torna a implementação mais acessível a FPGAs com limitações de recursos, e ainda permite dedicar o uso dos blocos DSPs a tarefas de processamentos de sinais, como circuitos para Filtros FIR e IIR, que comumente são aplicados em projetos juntamente com o cálculo da FFT.

Após carregar a FPGA com o arquivo *Bitstream* e programar o PS, fora enviado conjunto de sinais de entrada gerados pelo *Matlab* para a FFT implementada via interface UART do ZynqBerry. A resposta a estes sinais foi comparada com o resultado teórico, obtido pelo uso do comando *fft()* do *Matlab*, aplicada aos sinais gerados. A fim de mensurar o erro entre a FFT teórica, obtida no *Matlab*, e a implementada, proveniente da FFT do ZynqBerry, foi utilizada a função de cálculo do SQNR (*snr()*). O nível SQNR médio dos teste realizados com a FFT de 16 pontos foi de 52dB. A Tabela (6) apresenta um comparativo entre o nível de SQNR encontrado para esta FFT e a vista na bibliografia.

Arquitetura	CORDIC	SQNR(dB)	Wordlength(bits)	Referência
Radix-2	MSR	52	16	Autoria Própria
Radix-4	-	40,92	8	Hassan <i>et al.</i> (2018)
Radix-8	-	41,97	8	Hassan <i>et al.</i> (2018)
Radix- $2^2$	-	46	12	Saeed <i>et al.</i> (2009)

**Tabela 6: Comparativo Nível de SNQR para FFT de 16 Pontos**

**Fonte:** Autoria Própria

Em comparação entre as demais FFTs apresentadas na Tabela (6), a principal diferença, entra a FFT implementada neste trabalho e a vista na bibliografia, é a

arquitetura utilizada. Como afirma Siqueira (2004), a arquitetura Radix-4 consegue reduzir em 25% as operações de rotações vetoriais, reduzindo os recursos consumidos da FPGA, e impactando também em um aumento de desempenho. O mesmo acontece para o Radix-8 e o Radix-2<sup>2</sup>, os quais possuem arquiteturas que beneficiam a redução de recursos, e o aumento de desempenho evitando as operações de rotação.

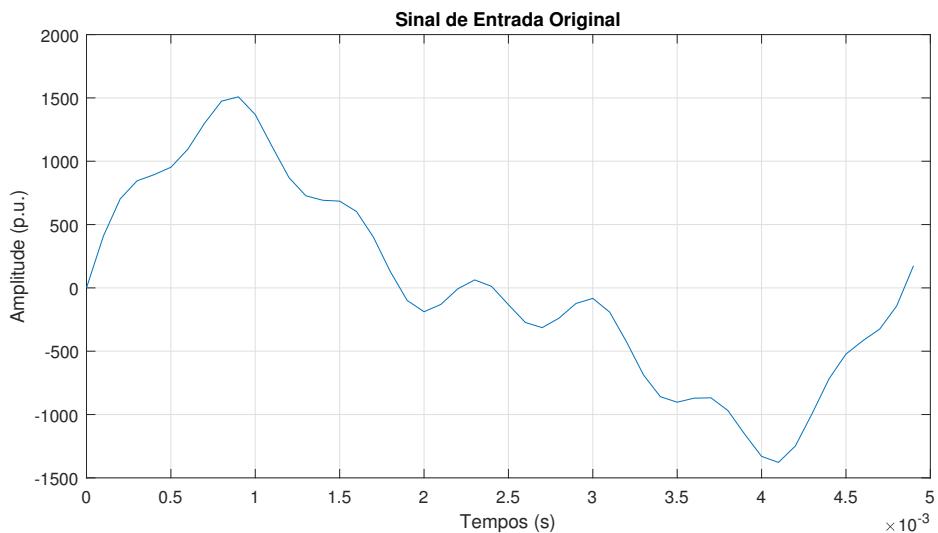
O desempenho da FFT aqui implementada, mesmo que possua uma melhor resolução, o que poderia justificar o bom desempenho, ainda parece se beneficiar bem do operador MSR CORDIC, atingindo assim um bom valor de SQNR. Outro ponto importante é o desempenho da FFT em termos de ciclos de *clock*. Pois, nesta arquitetura a FFT de 16 pontos computa todos os dados de entrada em apenas 12 ciclos de *clock*.

Para demonstrar o desempenho da FFT de 16 pontos, a Figura (37) apresenta o um sinal de entrada enviado a FFT, o qual pode ser expresso por:

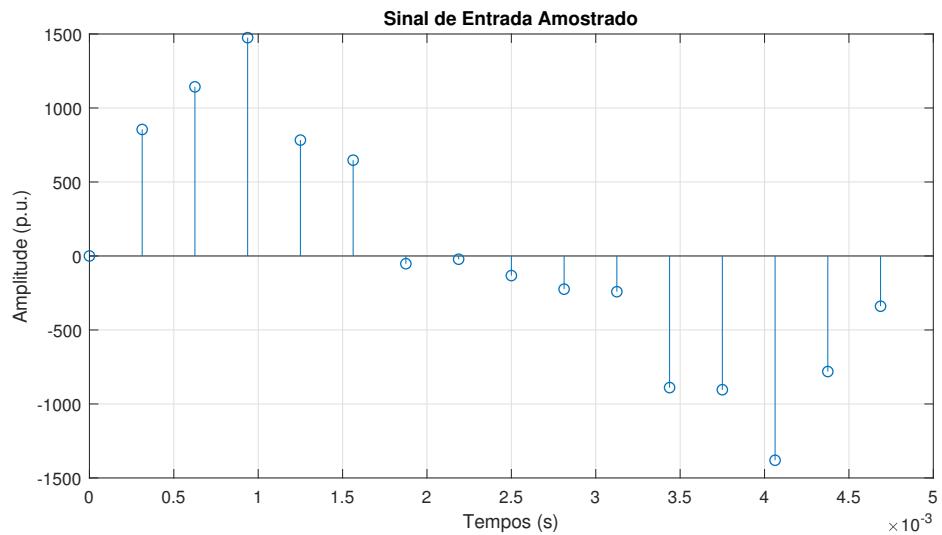
$$X_{in} = 1012 * \sin(2\pi(213)t) + 520 * \sin(2\pi(410)t) + \dots \quad (114)$$

$$+ 183 * \sin(2\pi(1403)t). \quad (115)$$

A Figura (38) apresenta o sinal de entrada da FFT implementada, amostrado a uma frequência de 3200Hz. O resultado obtido pela FFT implementada é apresentada na Figura (39).

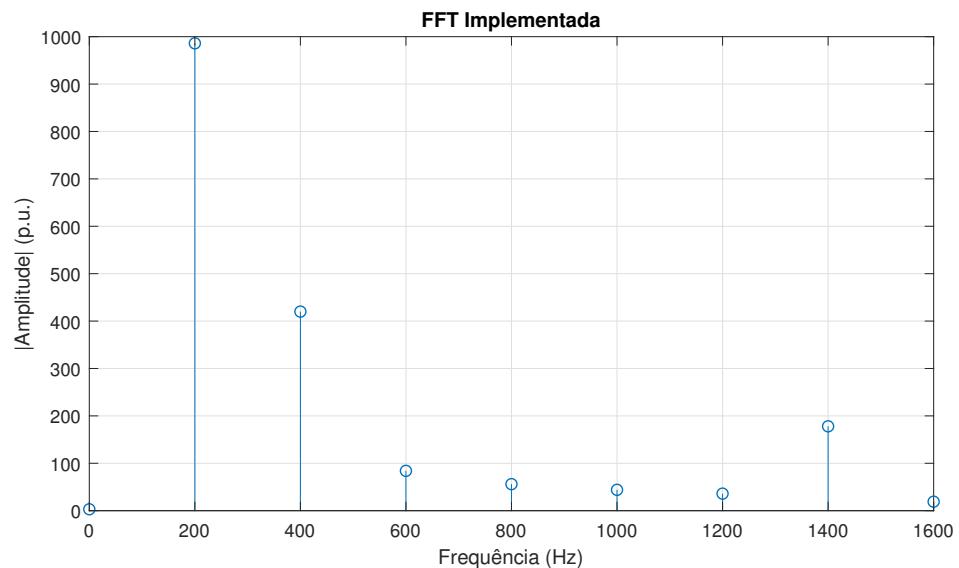


**Figura 37: Sinal com componentes em 213Hz, 410Hz, e 1403Hz - FFT de 16 Pontos**  
Fonte: Simulação Matlab

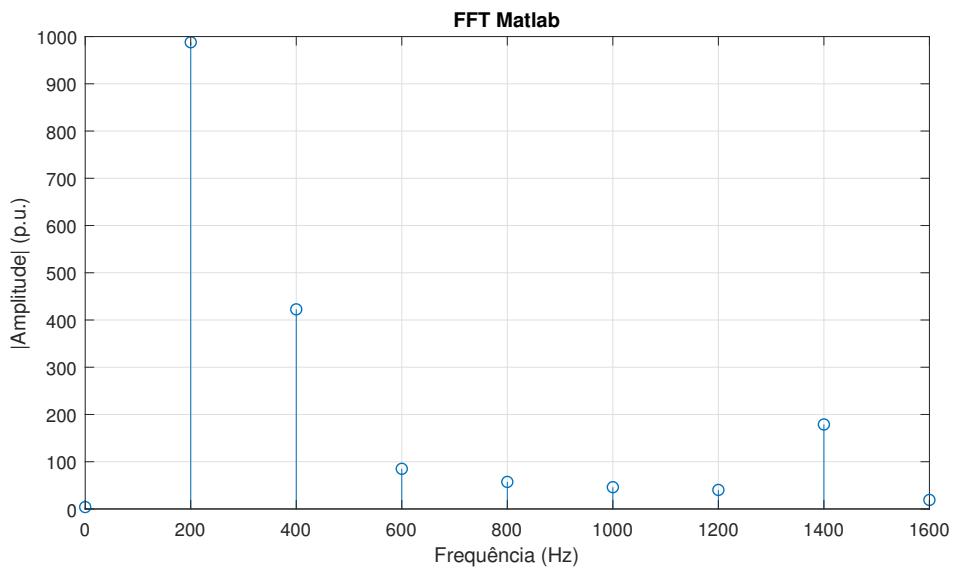


**Figura 38: Sinal de Entrada da FFT de 16 Pontos Amostrado a 3200 Hz**  
**Fonte:** Simulação Matlab

A partir do mesmo sinal de entrada da Figura (38), fora utilizado a função `fft()` do *Matlab*, a fim de realizar um comparativo dos resultados obtidos na Figura (39). Assim a Figura (40) apresenta o espectro de Fourier obtido no *Matlab*. A Figura (41) apresenta o erro entre o espectro de Fourier obtido pela FFT implementada e a obtida através do *Matlab*.

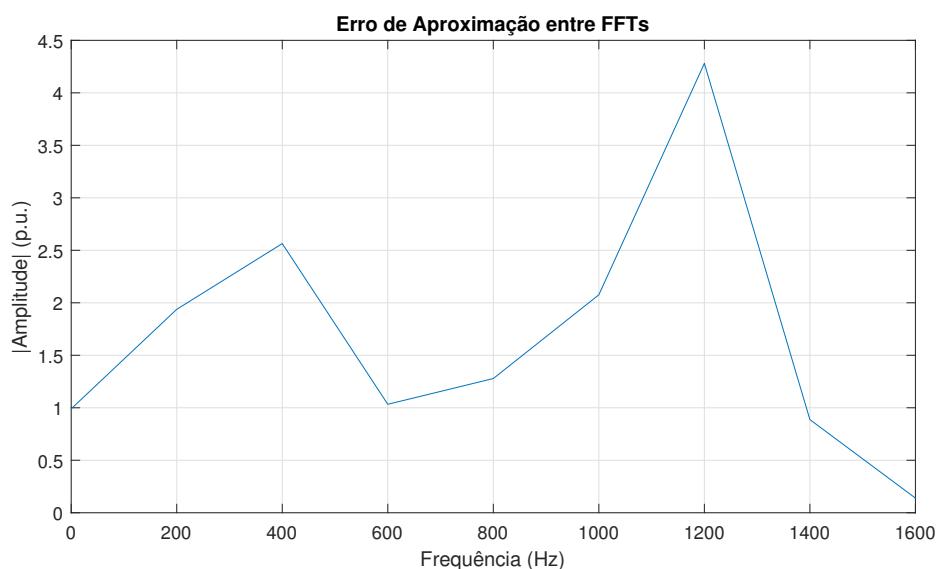


**Figura 39: Espectro de Fourier proveniente da FFT de 16 Pontos**  
**Fonte:** Autoria Própria



**Figura 40: Espectro de Fourier proveniente do *Matlab* para 16 Pontos**

Fonte: Autoria Própria



**Figura 41: Erro entre o resultado obtido pela FFT de 16 Pontos Implementada e o *Matlab***

Fonte: Autoria Própria

## 4.2 FFT DE 1024 PONTOS

Após a implementação do código VHDL, é realizada a síntese da FFT de 1024 pontos. A tabela (7) apresenta os dados referentes ao volume de recursos utilizados para esta implementação.

Recurso	Utilização	Utilização %
LUT	17021	96%
FF	22178	63%
BRAM	32	100%
DSP	0	0

**Tabela 7: Resultado de Síntese FFT 1024 Pontos**

**Fonte:** Autoria Própria

Assim como foi feito na seção anterior, a comparação entre os recursos consumidos pela FFT de 1024 pontos em relação as FFTs encontradas na bibliografia, deve ser realizada. Para tal, segue a Tabela (8).

Referência	LUT	FF	BRAM	DSP	Wordlength( <i>bits</i> )
Autoria Própria	17021	22178	16	0	16
Zhou <i>et al.</i> (2009)	18384	12256	8	16	16

**Tabela 8: Comparativo de Síntese - FFT 1024 Pontos**

**Fonte:** Autoria Própria

Além das diferenças explícitas na Tabela (8), a FFT apresentada por Zhou *et al.* (2009) implementa a arquitetura Radix-4SDC e não utiliza operadores CORDIC, mas sim blocos de DSP para realizar as operações de rotação vetorial. Em comparação, a FFT implementada neste trabalho consome mais recursos em termos de *flip-flops* e blocos de *RAM*, devido principalmente a presença dos operadores CORDIC.

Em relação ao desempenho de números de *clock* necessários para efetuar o cálculo da FFT, o *hardware* aqui implementado cumpre sua função em 1728 ciclos de *clock*. Levando em consideração que a cada nível da FFT as unidades CORDIC são reutilizadas 64 vezes, e que em cada operação é gasto 3 ciclos de *clock*, logo, cada nível leva 192 ciclos de *clock* para ser computado. O último nível da FFT não é considerado neste cálculo de desempenho, pois no nível 9 nenhuma operação CORDIC é realizada, sendo somente executado o envio de dados ao PS.

A FFT Radix-4SDC, apresentada por Zhou *et al.* (2009), consegue desempenhar o cálculo da FFT em 1024 ciclos. Segundo (HE; GUO, 2008), o DSP (*Digital Signal Processor*) comercial de ponto-fixo *TMS320C6416*, da fabricante *Texas Instruments*, computa uma FFT de 1024 pontos, com *wordlength* de 16 *bits*, em cerca de

6.526 ciclos de *clock*. Comparando o desempenho entre este DSP comercial e a FFT Radix-4SDC, pode-se perceber que a FFT aqui implementada possui um desempenho em velocidade de processamento superior a um DSP comercial, e mais próximo de uma FFT em FPGA da bibliografia.

Após programar a FPGA e o PS do ZynqBerry, foi realizado os testes de desempenho da FFT de 1024 pontos. Foram enviados dados de diferentes formas de onda, todas geradas pelo *software Matlab*. Após recolher os dados, fora calculado o valor médio do nível SQNR da FFT. Ao longo dos testes o valor do nível de SQNR da FFT ficou próximo a 41dB. A Tabela (9), relaciona este resultado aos demais encontrados na bibliografia.

Arquitetura	CORDIC	SQNR(dB)	Wordlength(bits)	Referência
Radix-2	MSR	41	16	Autoria Própria
Radix-4SDC	-	61,25	16	Zhou <i>et al.</i> (2009)
Radix-2	MSR	62,39	16	El-Shafiey <i>et al.</i> (2015)
Radix-2	EEAS	55,05	16	El-Shafiey <i>et al.</i> (2015)
Radix-2	MVR	51,53	16	El-Shafiey <i>et al.</i> (2015)

**Tabela 9: Comparativo Nível de SNQR para FFT de 1024 Pontos**

**Fonte:** Autoria Própria

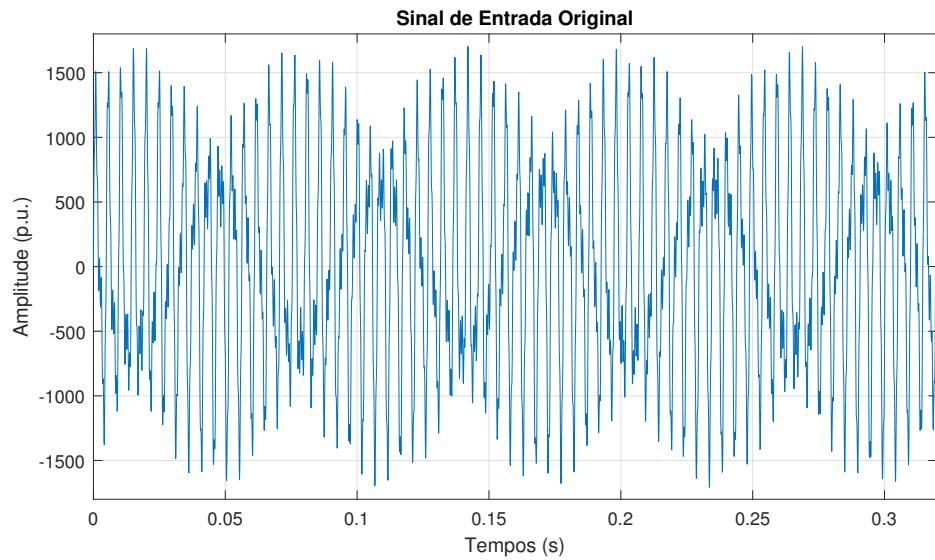
Como pode ser observado na Tabela (9), o desempenho da FFT implementada em relação ao nível de SQNR ficou abaixo do observado na bibliografia, mesmo para arquiteturas Radix-2 utilizando processadores EEAS-CORDIC. Este baixo desempenho se deve principalmente ao fato de que na arquitetura implementada não foram utilizados métodos de redução de latência de sinais, ou mesmos métodos de redução de erros como os aplicados por El-Shafiey *et al.* (2015).

Então, para demonstrar o desempenho da FFT de 1024 pontos, a Figura (??) apresenta o sinal descrito por:

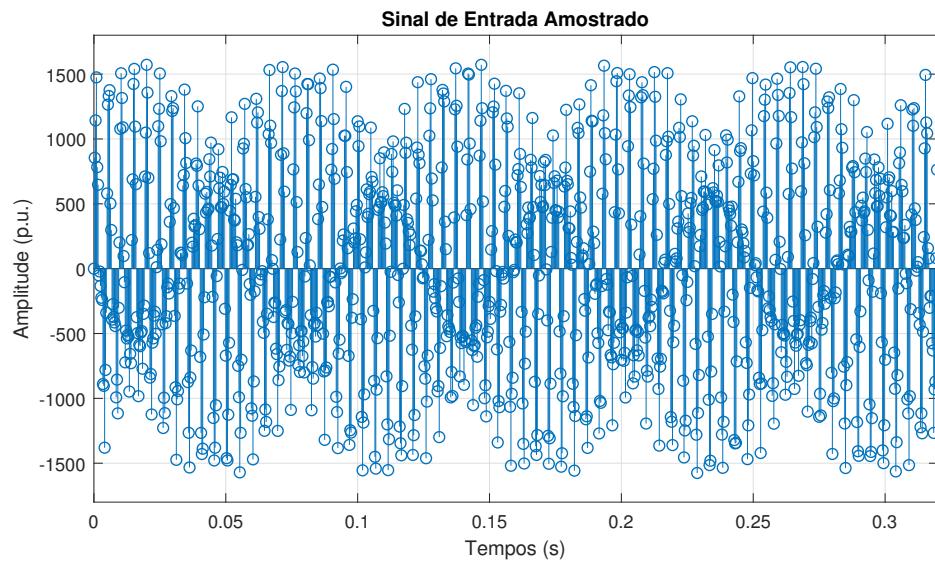
$$X_{in} = 1012 * \sin(2\pi(213)t) + 520 * \sin(2\pi(410)t) + \dots \quad (116)$$

$$+ 183 * \sin(2\pi(1403)t). \quad (117)$$

A Figura (43) apresenta o sinal de entrada da FFT de 1024 pontos amostrado a uma frequência de 3200Hz. Os dados da forma de onda da Figura 43 foram enviados a FFT, e o resultado obtido é apresentada na Figura (44).

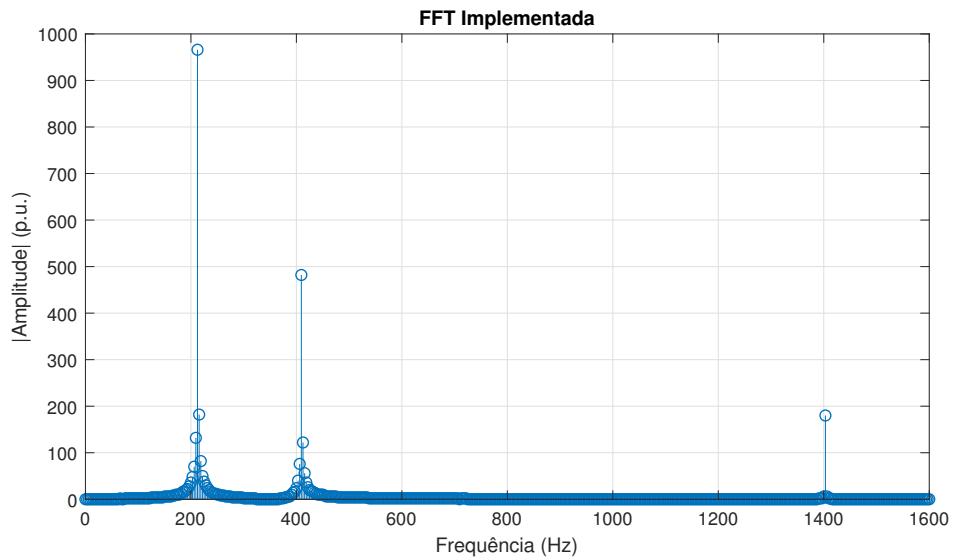


**Figura 42: Sinal com Componentes em 213Hz, 410Hz, e 1403Hz - FFT de 1024 Pontos**  
Fonte: Simulação Matlab

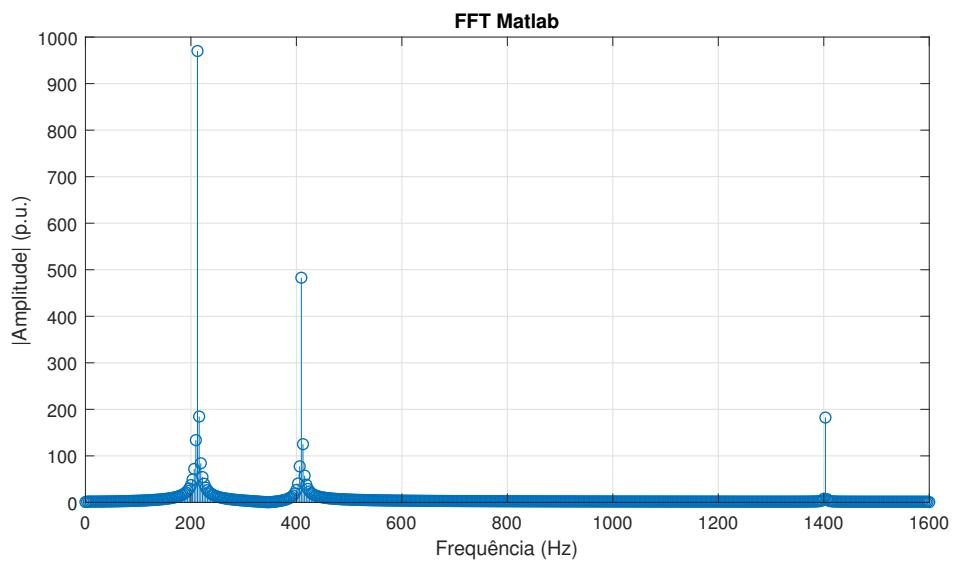


**Figura 43: Sinal de Entrada da FFT de 1024 Pontos Amostrado a 3200 Hz**  
Fonte: Simulação Matlab

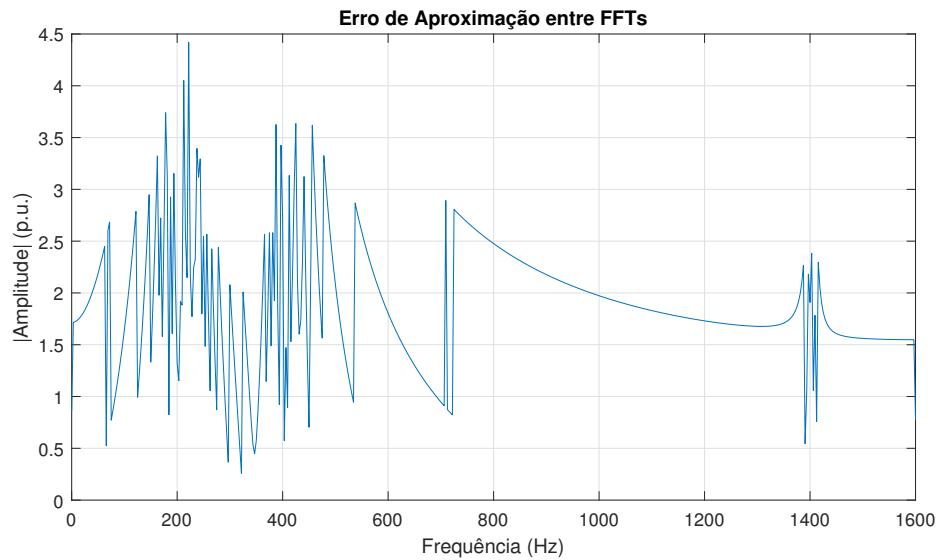
Então fora utilizado o *software Matlab* para obter o espectro da Fourier de referência do sinal amostrado, apresentado na Figura (43), de modo a realizar uma comparação com espectro obtido pela FFT Implementada. A Figura (45) apresenta o espectro de Fourier obtido no *Matlab*, e a Figura (46) apresenta o erro entre os espectros.



**Figura 44: Espectro de Fourier proveniente da FFT de 1024 Pontos**  
Fonte: Autoria Própria



**Figura 45: Espectro de Fourier proveniente do *Matlab* para 1024 Pontos**  
Fonte: Autoria Própria



**Figura 46: Erro entre o resultado obtido pela FFT de 1024 Pontos Implementada e o Matlab**

**Fonte:** Autoria Própria

## 5 CONCLUSÃO

O cálculo da FFT é uma ferramenta extremamente útil na análise de sinais, sendo empregada em diversas áreas da engenharia, desde telecomunicações, até na área de engenharia biomédica. Implementar esta poderosa ferramenta, em um ambiente como a FPGA, tem o intuito de desenvolver uma acelerador por *hardware*, que tome proveito do paralelismo natural do ambiente da FPGA, para tornar este processo de cálculo mais rápido e eficiente.

Neste trabalho fora apresentado as bases da transformada de Fourier, demonstrando a importância que o espectro de Fourier tem para a análise de sinais e aplicação desta série no ambiente digital, por meio da Transformada Rápida de Fourier. Em seguida, a fundamentação tórica deste trabalho ainda incluiu o algoritmo Radix-2, e o algoritmo CORDIC, juntamente com a sua variante, o MSR, utilizado na implementação deste trabalho. Fora apresentado ainda uma introdução aos dispositivos FPGA e também a família Zynq-7000.

No desenvolvimento deste trabalho fora introduzido a metodologia com que os parâmetros de operação MSR-CORDIC foram determinados, apresentando, inclusive, uma variação do algoritmo TBS, próprio para o MSR. As arquiteturas do módulo CORDIC, da FFT de 16 pontos e da FFT 1024 pontos também foram apresentados e devidamente justificados. Para que fosse possível integrar a implementação da FFT dentro do ambiente PL, com o processador em PS, fora desenvolvido uma interface *AXI-Lite* no modo *Slave*.

Após implementada, a FFT de 16 pontos foi capaz de computar a FFT em apenas 12 ciclos de *clock*, consumindo no total 5760 LUTs e 4895 Flip-flops. Os resultados dos testes realizados com auxílio do software *Matlab* comprovou a eficácia desta FFT, tendo esta alcançado um nível médio de SQNR de 52dB. Já a FFT de 1024 pontos implementada, foi capaz de computar a FFT em 1728 ciclos de *clock*, consumindo um total de 17021 LUTs e 22178 Flip-flops, além de 16 blocos de *RAM*. Já em relação ao desempenho SQNR, esta FFT atingiu uma média de 41dB durante os testes.

Um dos maiores obstáculo para a realização da implementação deste trabalho foi desenvolver uma arquitetura que possibilita-se obter um bom nível de parale-

lismo no processamento dos dados, de modo a acelerar o cálculo da FFT, e que ainda atingi-se um bom desempenho na redução do erro de cálculo, e que por fim estive-se dentro das restrições de recursos da FPGA utilizada. Para encontrar uma solução foi necessário implementar inicialmente a FFT de 16 pontos, de modo a medir o volume de recursos que uma arquitetura de menos iterações e maior paralelismo demandava. Em seguida foi desenvolvido uma arquitetura com um paralelismo parecido com a FFT de 16 pontos, porém capaz de calcular 1024 pontos utilizando blocos de RAM e mais iterações. Outro ponto que dificultou a realização deste trabalho foi a implementação efetiva da FFT na plataforma da Zynq-7000, já que esta família de SoC possuem muitas particularidades com relação a interface PS e PL. Pois além de desenvolver uma interface AXI adequada em PL, foi necessário desenvolver uma aplicação em PS, que fosse capaz de acessar os endereços de memória adequados de forma sincronizada com PL.

Os resultados aqui obtidos demonstram a eficiência do algoritmo Radix-2 para o cálculo da FFT aplicado a FPGA. Porém ao longo do desenvolvimento deste trabalho notou-se a possibilidade de se utilizar outro algoritmo de cálculo da FFT, que seja capaz de reduzir o número de operações CORDIC e que também reduza o erro de cálculo. Logo, para futuros trabalhos é interessante realizar implementações de algoritmos de cálculo como o algoritmo Radix-4 ou Radix-4SDC, os quais são capazes de reduzir o número de operações de rotação vetorial necessários. Ou ainda é possível utilizar o algoritmo Radix-2 modificado, proposto por El-Motaz *et al.* (2014), o qual se apresenta mais amigável a utilização de operadores CORDIC, reduzindo em até 38% da utilização de recursos da FPGA, além de elevar o nível SQNR.

Ao fim, partindo dos conceitos básicos sobre séries de Fourier, passando pelas aplicações destas séries no ambiente discreto e não periódico, até chegar na aplicação da arquitetura Radix-2 e do algoritmo CORDIC, este trabalho apresentou todo o embasamento necessário para realizar a implementação do cálculo da FFT em FPGA, usado o algoritmo Radix-2.

## REFERÊNCIAS

- AMARAL, Carlos Eduardo Ferrante do; LOPES, Heitor S; ARRUDA, Lígia V; HARA, Marcos S; GONÇALVES, Antonio J; DIAS, Adilson A. Design of a complex bioimpedance spectrometer using dft and undersampling for neural networks diagnostics. **Medical engineering & physics**, Elsevier, v. 33, n. 3, p. 356–361, 2011.
- ARM. System IP AMBA Specifications.** 2018. <https://www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications>. Acessado em 13 de Agosto de 2017.
- BINGHAM, J. A. C. Multicarrier modulation for data transmission: an idea whose time has come. **IEEE Communications Magazine**, v. 28, n. 5, p. 5–14, May 1990. ISSN 0163-6804.
- CHU, Eleanor; GEORGE, Alan. **Inside the FFT black box: serial and parallel fast Fourier transform algorithms**. 1. ed. [S.I.]: CRC Press, 1999.
- COOLEY, James W; TUKEY, John W. An algorithm for the machine calculation of complex fourier series. **Mathematics of computation**, JSTOR, v. 19, n. 90, p. 297–301, 1965.
- CORMEN, Thomas H; LEISERSON, Charles E; RIVEST, Ronald L; STEIN, Clifford. **Algoritmos: teoria e prática**. 2. ed. [S.I.]: Editora Campus, 2002.
- CROCKETT, Louise H; ELLIOT, Ross A; ENDERWITZ, Martin A; STEWART, Robert W. The zynq book. **Strathclyde Academic Media**, 2014.
- DESPAIN, Alvin M. Fourier transform computers using cordic iterations. **IEEE Transactions on Computers**, IEEE, v. 100, n. 10, p. 993–1001, 1974.
- EL-MOTAZ, M. A.; NASR, O. A.; OSAMA, K. A cordic-friendly fft architecture. In: **2014 International Wireless Communications and Mobile Computing Conference (IWCMC)**. [S.I.: s.n.], 2014. p. 1087–1092. ISSN 2376-6492.
- EL-SHAFIEY, Ahmed M; FARAG, Mohamed E; EL-MOTAZ, Mohammed A; NASR, Omar A; FAHMY, Hossam AH. Two-stage optimization of cordic-friendly fft. In: **IEEE. Electronics, Circuits, and Systems (ICECS), 2015 IEEE International Conference on**. [S.I.], 2015. p. 408–411.
- ELETTRONIC, Inc Trenz. **TE0726 - ZynqBerry**. San Jose, EUA, Julho 2018. V2.1.
- FAROOQ, Umer; MARRAKCHI, Zied; MEHREZ, Habib. **Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization**. [S.I.]: Springer Science & Business Media, 2012.

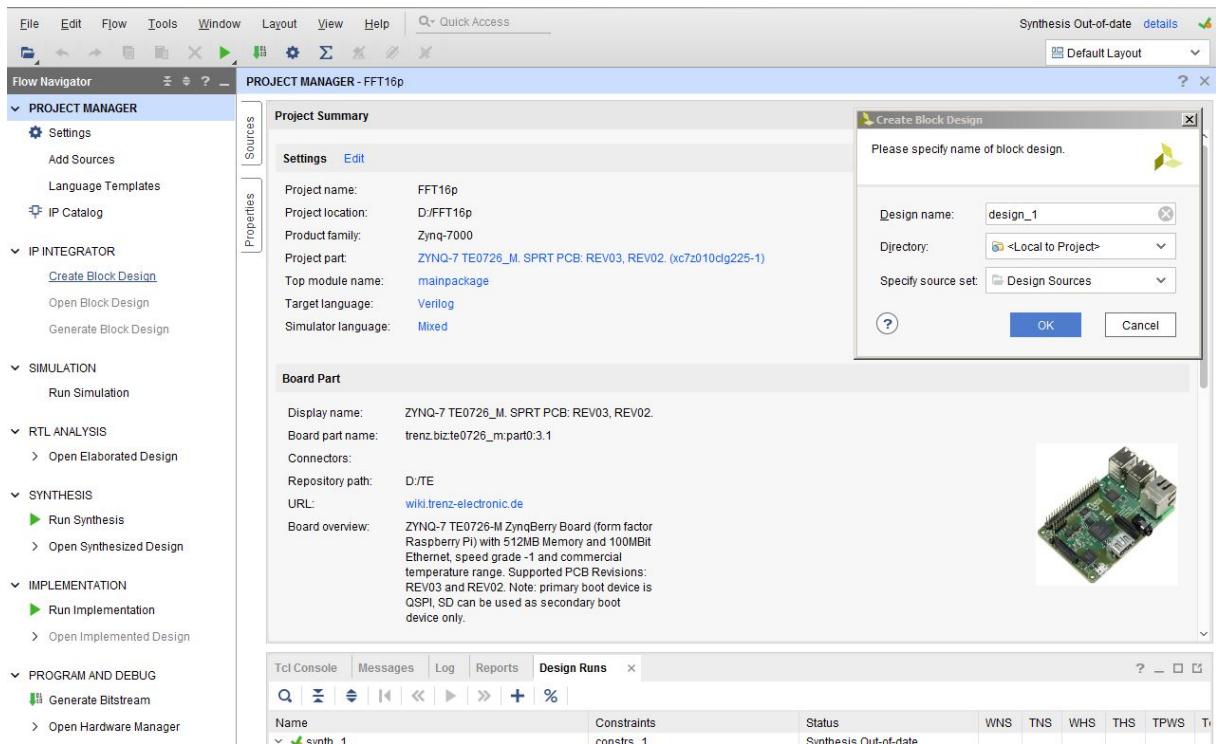
- GARRIDO, M.; KİLLİSTRİM, P.; KUMM, M.; GUSTAFSSON, O. Cordic ii: A new improved cordic algorithm. **IEEE Transactions on Circuits and Systems II: Express Briefs**, v. 63, n. 2, p. 186–190, Feb 2016. ISSN 1549-7747.
- HASSAN, SLM; SULAIMAN, N; SHARIFFUDIN, SS; YAAKUB, TNT. Signal-to-noise ratio study on pipelined fast fourier transform processor. **Bulletin of Electrical Engineering and Informatics**, v. 7, n. 2, p. 230–235, 2018.
- HAYKIN, SIMON S; VEEN, Barry Van. **Sinais e sistemas**. [S.I.]: Bookman, 2001.
- HE, Hongjiang; GUO, Hui. The realization of fft algorithm based on fpga co-processor. In: **IEEE. Intelligent Information Technology Application, 2008. IITA'08. Second International Symposium on**. [S.I.], 2008. v. 3, p. 239–243.
- IBRAHIM, Muhammad; KAMAL, Mohsin; KHAN, Omar; ULLAH, Khalil. Analysis of radix-2 decimation in time algorithm for fpga co-processors. Computing, Electronic and Electrical Engineering 2016 International Conference, 2016.
- KUO, Jen-Chih; WEN, Ching-Hua; LIN, Chih-Hsiu; WU, An-Yeu (Andy). Vlsi design of a variable-length fft/ifft processor for ofdm-based communication systems. **EURASIP Journal on Advances in Signal Processing**, v. 2003, n. 13, p. 439360, Dec 2003. ISSN 1687-6180. Disponível em: <<https://doi.org/10.1155/S1110865703309060>>.
- LATHI, Bhagwandas Pannalal. **Sinais e Sistemas Lineares**. 2. ed. [S.I.]: Brasil: Bookman, 2007.
- LIN, Chih-Hsiu; WU, An-Yeu. Mixed-scaling-rotation cordic (msr-cordic) algorithm and architecture for high-performance vector rotational dsp applications. **IEEE Transactions on Circuits and Systems I: Regular Papers**, v. 52, n. 11, p. 2385–2396, Nov 2005. ISSN 1549-8328.
- LIU, C.; MOREIRA, P.; ZEMITI, N.; POIGNET, P. 3d force control for robotic-assisted beating heart surgery based on viscoelastic tissue model. In: **2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society**. [S.I.]: s.n., 2011. p. 7054–7058. ISSN 1094-687X.
- MARTINSEN, Orjan G; GRIMNES, Sverre. **Bioimpedance and bioelectricity basics**. [S.I.]: Academic press, 2011.
- MEHER, P. K.; VALLS, J.; JUANG, T. B.; SRIDHARAN, K.; MAHARATNA, K. 50 years of cordic: Algorithms, architectures, and applications. **IEEE Transactions on Circuits and Systems I: Regular Papers**, v. 56, n. 9, p. 1893–1907, Sept 2009. ISSN 1549-8328.
- MEYER-BAESE, Uwe. **Digital signal processing with field programmable gate arrays**. 3. ed. [S.I.]: Springer, 2007.
- MOORE, Andrew. **FPGAs for Dummies Altera Special Edition**. [S.I.]: Wiley Brand, 2007.
- OPPENHEIM, Alan V; WILLSKY, Alan S. **Sinais e Sistemas**. 2. ed. [S.I.]: Brasil:Pearson, 2010.

- PARK, S. Y.; YU, Y. J. Fixed-point analysis and parameter selections of msr-cordic with applications to fft designs. **IEEE Transactions on Signal Processing**, v. 60, n. 12, p. 6245–6256, Dec 2012. ISSN 1053-587X.
- SAEED, Ahmed; ELBAILY, M; ABDELFADDEEL, G; ELADAWY, MI. Efficient fpga implementation of fft/ifft processor. **International Journal of circuits, systems and signal processing**, v. 3, n. 3, p. 103–110, 2009.
- SANTHOSH, Lakshmi; THOMAS, Anoop. Implementation of radix 2 and radix 2 2 fft algorithms on spartan6 fpga. In: **IEEE. Computing, Communications and Networking Technologies (ICCCNT), 2013 Fourth International Conference on**. [S.I.], 2013. p. 1–4.
- SIQUEIRA, Tonny Matos. **Implementação de um modem OFDM em FPGA**. 2004. <ftp://ftp.lab.unb.br/pub/hardware/opencores/ofdm/Docs/OFDM%20-%20portuguese.pdf>. Acessado em 16 de Agosto de 2017.
- TRADINGVIEW. **Índice BM & FBOVESPA**. 2017. <https://www.tradingview.com/symbols/BMFBVESPA-IBOV/>. Acessado em 14 de Agosto de 2017.
- TRIANTIS, Iasonas F; DEMOSTHENOUS, Andreas; RAHAL, Mohamad; HONG, Hongwei; BAYFORD, Richard. A multi-frequency bioimpedance measurementasic for electrical impedance tomography. In: **IEEE. ESSCIRC (ESSCIRC), 2011 Proceedings of the**. [S.I.], 2011. p. 331–334.
- VANMATHI, K; SEKAR, K; RAMACHANDRAN, Remya. Fpga implementation of fast fourier transform. In: **IEEE. Green Computing Communication and Electrical Engineering (ICGCCE), 2014 International Conference on**. [S.I.], 2014. p. 1–5.
- VOLDER, J. E. The cordic trigonometric computing technique. **IRE Transactions on Electronic Computers**, EC-8, n. 3, p. 330–334, Sept 1959. ISSN 0367-9950.
- WANG, B.; ZHANG, Q.; AO, T.; HUANG, M. **Design of Pipelined FFT Processor Based on FPGA**. Jan 2010. 432-435 p.
- WU, Cheng-Shing; WU, An-Yeu. A novel trellis-based searching scheme for e eas-based cordic algorithm. In: **2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221)**. [S.I.: s.n.], 2001. v. 2, p. 1229–1232 vol.2. ISSN 1520-6149.
- WU, Cheng-Shing; WU, An-Yeu; LIN, Chih-Hsiu. A high-performance/low-latency vector rotational cordic architecture based on extended elementary angle set and trellis-based searching schemes. **IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing**, v. 50, n. 9, p. 589–601, Sept 2003.
- XILINX. **AXI Reference Guide**. 2017. [https://www.xilinx.com/support/documentation/ip\\_documentation/ug761\\_axi\\_reference\\_guide.pdf](https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf). Acessado em 15 de Agosto de 2017.
- XILINX. **Zynq-7000 SoC Data Sheet: Overview**. 2018. [https://www.xilinx.com/support/documentation/data\\_sheets/ds190-Zynq-7000-Overview.pdf](https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf). Acessado em 16 de Agosto de 2017.

ZHOU, Bin; PENG, Yingning; HWANG, David. Pipeline fft architectures optimized for fpgas. **International Journal of Reconfigurable Computing**, Hindawi Publishing Corp., v. 2009, p. 1, 2009.

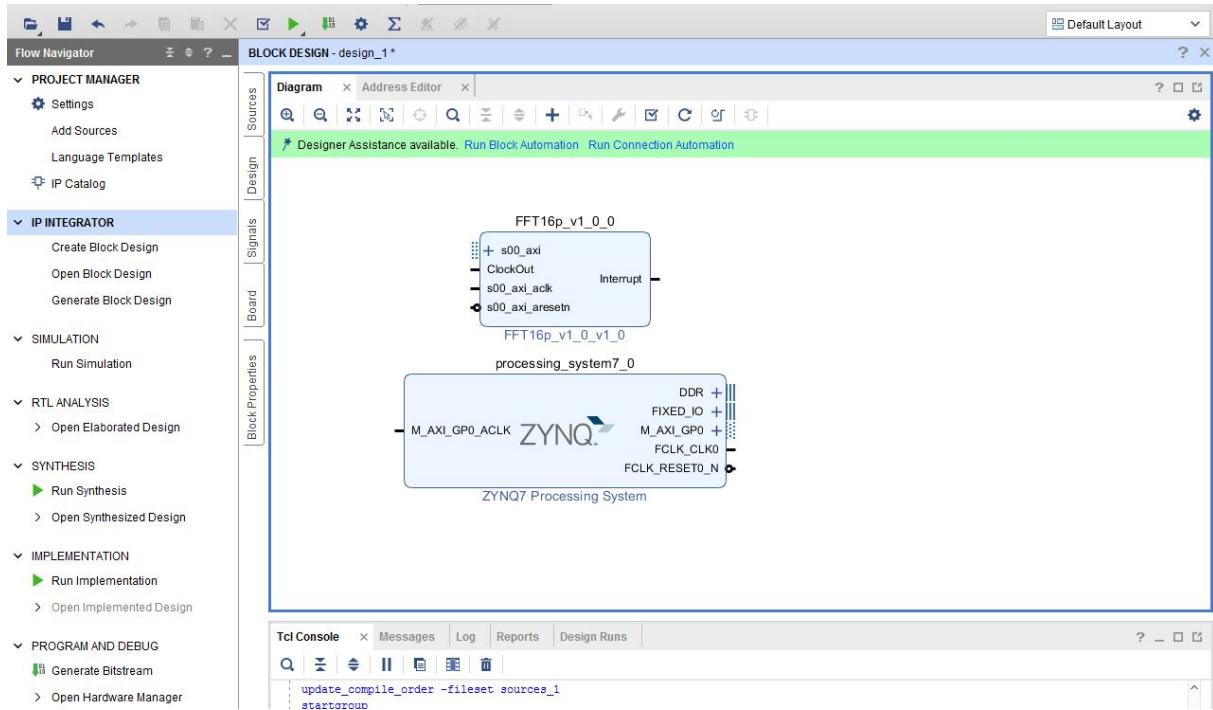
## APÊNDICE A - PROGRAMANDO O ZYNQBERRY

Para enviar os dados para a FFT em PL, é necessário que o processador Cortex-A9 consiga mapear o endereço do periférico de Interface AXI relacionada com a FFT. Após criar uma nova IP com a interface AXI, é necessário criar um novo Bloco de Design, no Vivado HLx 2017.4, como pode ser visto na Figura(47).



**Figura 47: Criação de um Novo Bloco de Design**  
Fonte: Vivado HLx 2017.4

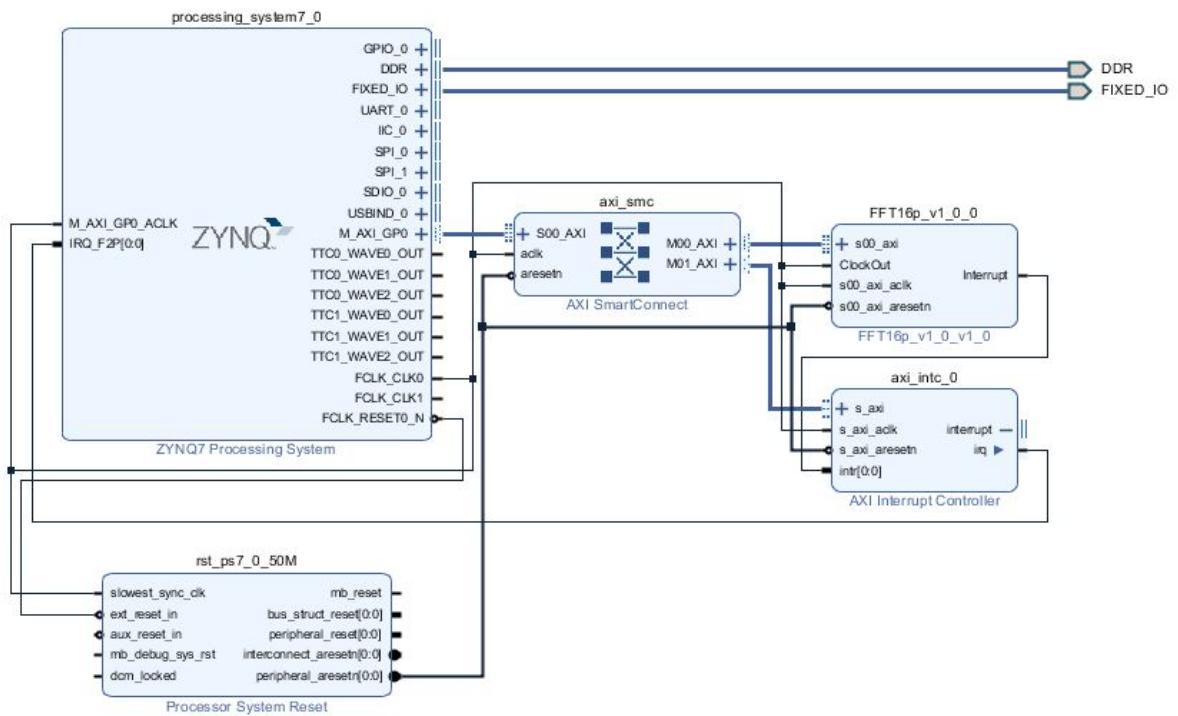
A partir deste novo bloco é preciso incluir uma IP *ZYNQ7 Process System*, o qual representa o ambiente PS, juntamente com um IP AXI da FFT, criada no Anexo (B), e também uma unidade de controle de Interrupções (*axi\_int*).



**Figura 48: Criação de um Novo Bloco de Design**

Fonte: Vivado HLx 2017.4

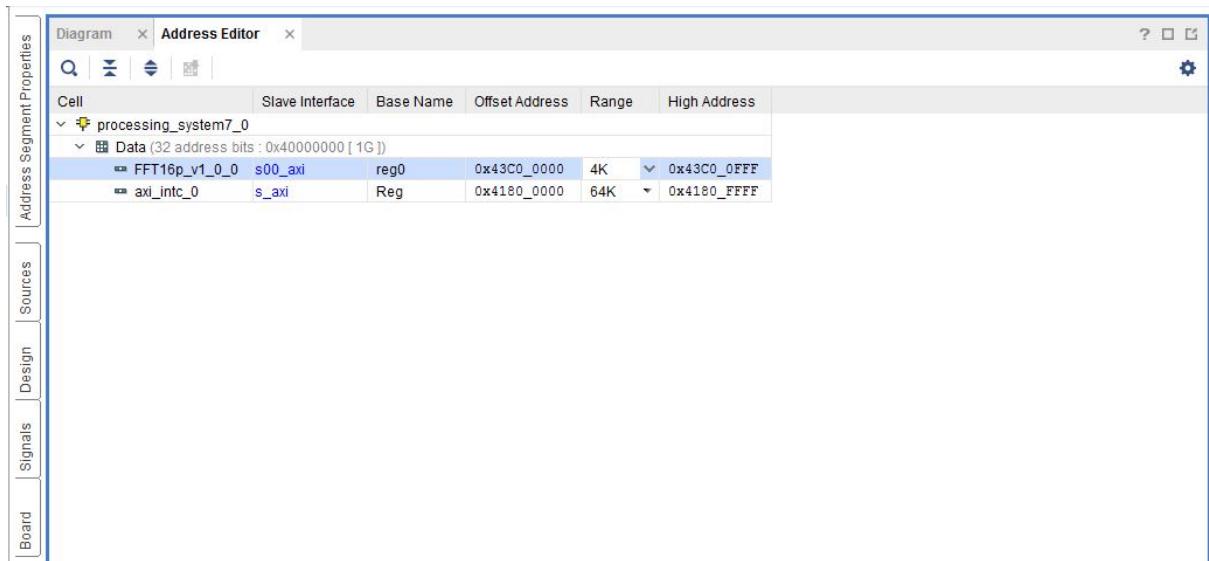
Após a inclusão deste 3 blocos é preciso realizar as devidas conexões entre os blocos e a configuração do bloco ZYNQ7. O Vivado 2017.4 disponibiliza ferramentas de automatização das conexões entre blocos e suas configurações. Após executar estas ferramentas, e conectar a saída da interrupção do Bloco da FFT ao bloco de controle de interrupções, o bloco de design é concluído, como mostra a Figura (49).



**Figura 49: Bloco de Design para FFT 16 Pontos**

Fonte: Vivado HLx 2017.4

Para identificar o endereço de memória que dá acesso aos registradores da Interface AXI do FFT, basta abrir a aba *Address Editor*. Como mostra a Figura (50), o Vivado automaticamente reservou um endereço de memória que começa em *0x43C00000*. Logo, basta somente criar uma rotina em PS, para acessar este endereço e enviar os dados.



**Figura 50: Endereço de Memória da Interface AXI FFT**

Fonte: Vivado HLx 2017.4

Por fim, após a inclusão e configurações de todos os blocos de IPs, basta rodar o comando *Create HDL Wrapper*, para que o Vivado crie a hierarquia de códigos HDL das IPs instanciadas. Em seguida é rodado a função de síntese (*Run Synthesis*), e implementação (*Run Implementation*). Se todos os passos anteriores forem executados corretamente, então o comando de geração do Bitstream (*Generate Bitstream*) pode ser executado.

Para programar a parte PS e também carregar a FPGA com o arquivo *Bitstream* gerado, o projeto de *hardware* feito no Vivado HLx precisa ser exportado para um projeto de *hardware/software* e então carregado para dentro do IDE Vivado SDK 2017.4. Para isso, basta escolher a opção File, na barra de tarefas do projeto aberto no Vivado HLx, ir em *Export*, em seguida em *Export Hardware*. Ao fim, basta abrir o Vivado SDK, pela opções *Launch SDK*.

Após exportar o projeto para o Vivado SDK, o PS foi programado para utilizar a interface UART0 do ZynqBerry para receber e enviar os dados retirados do endereço de memória da interface AXI da FFT de 16 pontos implementada. Para realizar esta operação foram necessários apenas algumas operações com ponteiros de memória. Ao fim, basta executar a compilação e *debugger* do Vivado SDK, para então realizar os testes de funcionamento da FFT.

## APÊNDICE B - IMPLEMENTANDO A INTERFACE AXI

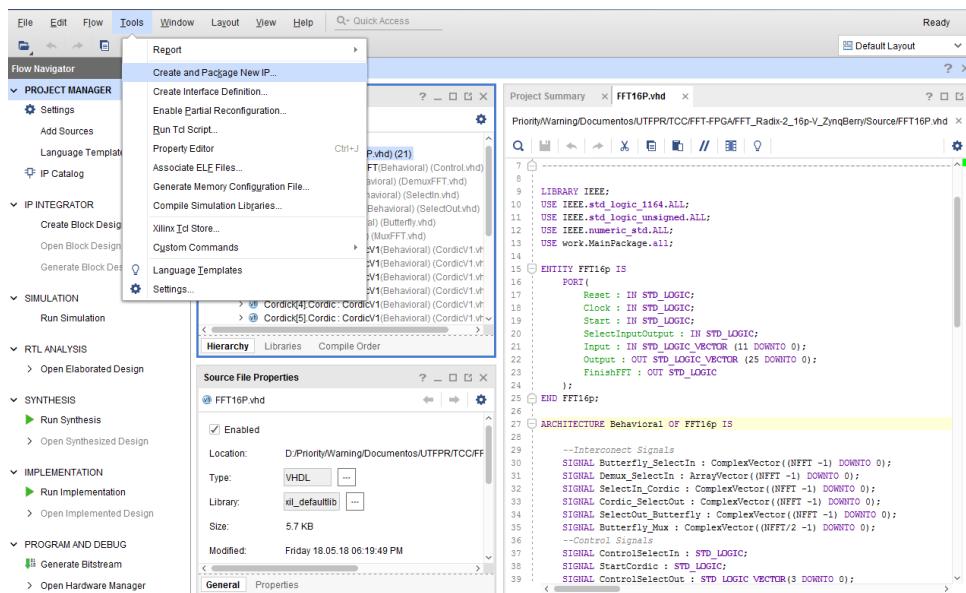
Como citado na Seção (2.5.2.1), na família de SoC Zynq-7000, o AXI é o meio padrão pelo qual diferentes componentes dentro do ambiente PL se conectam, e mais importante, é o meio pelo qual componentes em PL se comunicam com o processador em PS. Como a implementação da FFT, realizada na Seção (3.3), é feita em PL, e se deseja enviar e receber dados da FFT via UART, ao computador fora da placa ZynqBerry, é preciso incluir uma Interface AXI.

Uma Interface AXI para este projeto deve ser capaz de converter os sinais de saída da FFT para um formato padronizado de comunicação AXI. Segundo Xilinx (2018), existem 3 tipos básicos de interfaces AXI:

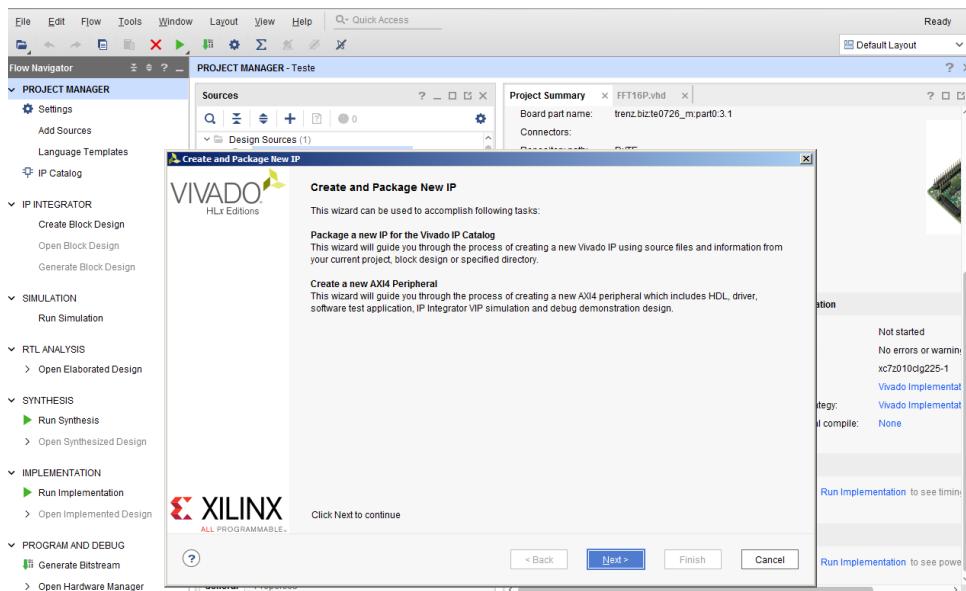
- **AXI-Standard:** Interface destinada a transferências de dados em alta velocidade e de maior volume. Utiliza endereçamento de memória para acessar dados, e portanto consome recursos de memória para ser implementado. Interface mais complexa, e oferece maiores opções de controle de dados, inclusive transferência no modo *burst*.
- **AXI-Lite:** Interface destinada a transferências de dados em baixa velocidade, e de menor volume. Consome espaço de memória apenas para controlar dados da transferência, como destino, origem e status da transmissão. Interface mais simples do que a AXI-Standard, não oferece controle de dados como o modo *burst*.
- **AXI-Stream:** Interface para transferência de dados em alta velocidade, porém não utiliza mapeamento de memória. Toda a transferência nesta interface é penteada em fluxo de dados contínuos, os quais não são armazenados pela interface.

Teoricamente qualquer uma das 3 opções de interface AXI seriam capazes de transmitir e receber os dados da FFT. Porém deve ter em mente que uma interface AXI também consome recursos da FPGA, os quais devem ser destinados principalmente a FFT. Logo uma interface como a AXI-Standard não seria adequado, já que não há necessidade de utilizar recursos como controle de dados. Neste projeto é de interesse ter acesso seletivo aos dados de saída da FFT, de modo que seja possível ao PS acessar os valores de amplitude de frequências específicas. Como a interface AXI-Stream não possui mapeamento de memória, não seria de interesse utilizar esta interface. Portanto, será utilizada a interface AXI-Lite.

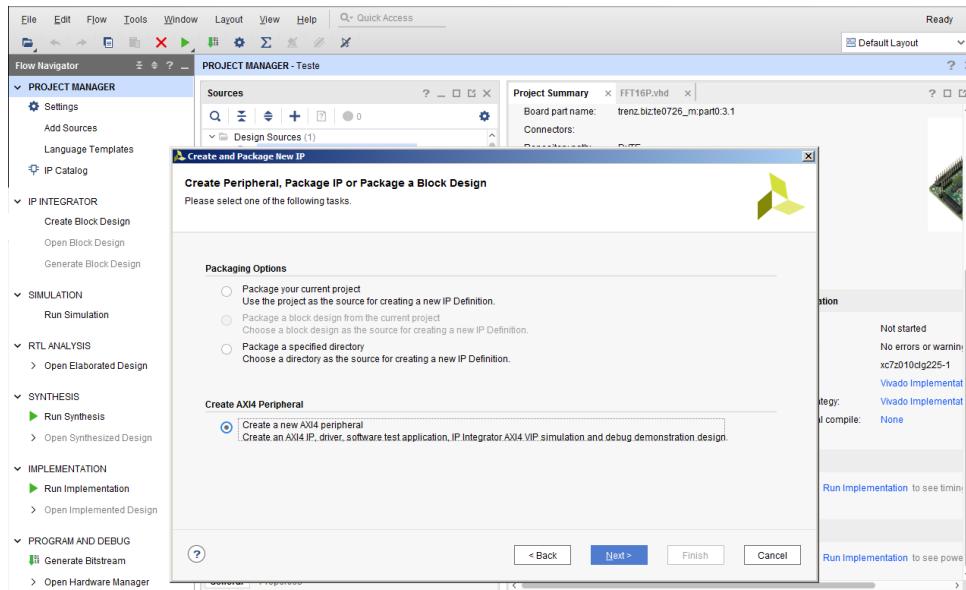
O fabricante do Zynq-7010 que equipa a placa ZynqBerry, utilizada neste trabalho, disponibiliza em sua IDE de desenvolvimento, o Vivado HLx 2017.4, uma ferramenta de criação de IPs. Utilizando esta ferramenta, partindo design em VHDL da FFT, foi criado um novo periférico AXI, como pode ser visto nas Figuras (51), (52), (53) e (54).



**Figura 51: Caminho até a ferramenta: Tools: Create and Package New IP**  
Fonte: Vivado HLx 2017.4

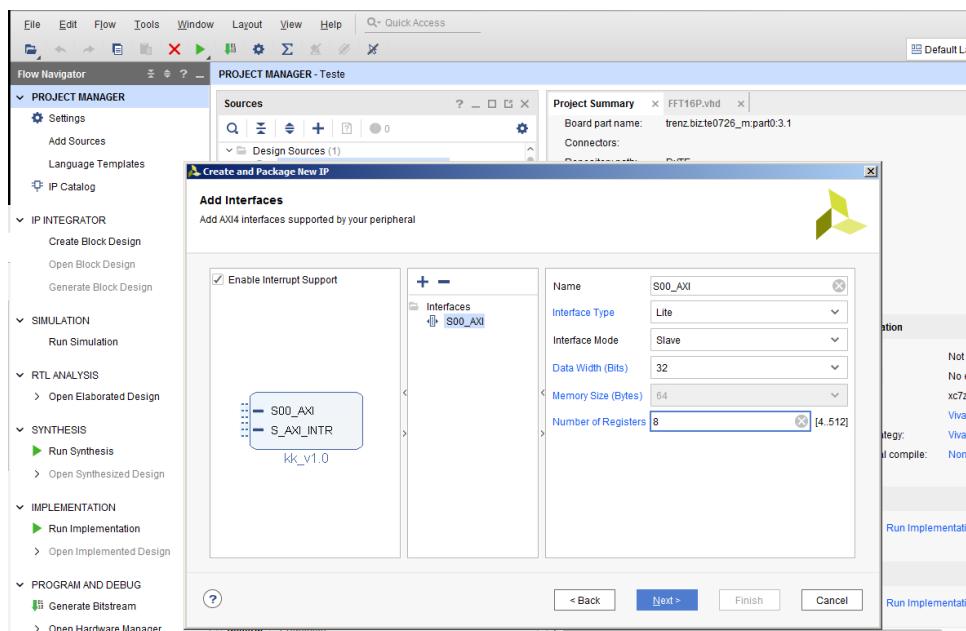


**Figura 52: Ferramenta de Criação e Empacotamento de Novas IPs**  
Fonte: Vivado HLx 2017.4



**Figura 53: Escolha da Opção: Create a New Peripheral**

Fonte: Vivado HLx 2017.4



**Figura 54: Configurações da nova IP**

Fonte: Vivado HLx 2017.4

Como pode ser observado na Figura (54), para a criação da Interface AXI foi escolhido utilizar uma interface do tipo *Lite*, no modo *Slave*, com 8 endereços de memória, cada endereço com 32 bits, e ainda escolhida a opção de habilitação de interrupções. Como é necessário transmitir apenas metade dos 16 pontos da FFT, ao fim do processo de cálculo, são necessários 8 endereços, sendo que cada endereço

de 32 *bits* é dividido de forma que, os 16 *bits* mais significativos são destinados a parte real do sinal, e os 16 textit{bits} menos significativos para a parte imaginária. Para o envio dos dados, basta preencher cada um dos 8 endereços de 32 *bits*, com o valor de dois pontos de 16 *bits* do sinal de entrada concatenados, considerando que os dados de entrada são puramente reais.

Após a criação do Periférico AXI, foi necessário editar o código Verilog do periférico, a fim de incluir o código da implementação em VHDL da FFT de 16 pontos. Após feitas tais alterações, o design é salvo, e o Vivado geral os arquivos fonte para uma nova IP, que contém a implementação da FFT. Esta IP pode ser reutilizada em qualquer outra IDE de desenvolvimento de FPGAs, desde que estas tenham suporte a AXI4.