UNIVERSIDADE FEDERAL DE JUIZ DE FORA
FACULDADE DE ENGENHARIA
ENGENHARIA ELÉTRICA

Leandro Silva Lima

Projeto de um Processador Embarcado Otimizado para Aplicação com Transformada de Fourier

Juiz de Fora
2016

Leandro Silva Lima

Projeto de um Processador Embarcado Otimizado para Aplicação com Transformada de Fourier

Monografia apresentada ao Departamento de Circuitos Elétricos da Faculdade de Engenharia Elétrica, da Universidade Federal de Juiz de Fora como requisito parcial a obtenção do grau de Bacharel em Engenharia Elétrica.

Orientador: Prof. Luciano Manhães de Andrade Filho, Dr.

Juiz de Fora

2016

Imprimir na parte inferior, no verso da folha de rosto a ficha disponível em:
http://www.ufjf.br/biblioteca/servicos/usando-a-ficha-catalografica/

Leandro Silva Lima

Projeto de um Processador Embarcado Otimizado para Aplicação com Transformada de Fourier

Monografia apresentada ao Departamento de Circuitos Elétricos da Faculdade de Engenharia Elétrica, da Universidade Federal de Juiz de Fora como requisito parcial a obtenção do grau de Bacharel em Engenharia Elétrica.

Aprovada em (dia) de (mês) de (ano)

BANCA EXAMINADORA

_____

Prof. Dr. Eng. Luciano Manhães de Andrade Filho - Orientador.

Universidade Federal de Juiz de Fora

_____

Prof. Dr. Eng. Leandro Rodrigues Manso Silva

Universidade Federal de Juiz de Fora

_____

M. Eng. Eder Barbosa Kapsisch

Universidade Federal de Juiz de Fora

AGRADECIMENTOS

Agradeço ao Orientado Professor Dr. Luciano de Andrade Manhães Filho que teve papel fundamental na elaboração desse trabalho e por ser o responsável por despertar em mim o interesse em estudar FPGA.

Agradeço, também, ao Professor Leandro Rodrigues Manso Silva por nunca se negar a ajudar em diversos momentos.

RESUMO

A presente monografia tem como objetivo descrever os procedimentos necessários para a modificação de um processador embarcado em uma FPGA, desenvolvido na UFJF, através da linguagem de descrição de hardware: Verilog. Para tal, técnicas de otimização foram utilizadas com o objetivo de que seja possível a aplicação da Transformada de Fourier minimizando os custos computacionais, ou seja, fazendo com quem tal processador seja capaz de poupar tempo de processamento.

Palavras-chave: FPGA. Processador Embarcado. Verilog. Transformada de Fourier. FFT. DFT. C. C++. C+/-. Assembly.

ABSTRACT

The aim of this monograph is to describe the procedures needed for the development of an embedded processor in a FPGA, developed at UFJF, using the hardware description language: Verilog. For that, optimization techniques were used in order to enable the application of the Fourier Transform minimizing the computational costs, making this processor able to save processing time.

Keywords: FPGA. Embedded Processor. Verilog. Fourier Transform. FFT. DFT. C. C++. C+/-. Assembly.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

FFT	Transformada Rápida de Fourier.
C++	Linguagem de Programação C++.
C++	Linguagem de Programação C.
C+/-	Linguagem de Programação C+/-.
FPGA	Field Programmable Gate Array.
LEs	Elementos Lógicos.
I/O	Input/ Output.
FM	Frequency Modulation.
PPLs	Phase Locked Loop.
LUT	Look up Table.
ULA	Unidade Lógica Aritmética

LISTA DE SÍMBOLOS

π	“pi” . Relação entre o perímetro de uma circunferência e diâmetro.
Ω	“Omega”. Frequência Digital.
O(n)	Ordem de um algoritmo.

SUMÁRIO

1 INTRODUÇÃO

Diversas aplicações em Engenharia Elétrica necessitam que sinais de diferentes naturezas sejam analisados no domínio da frequência, como, por exemplo, os sistemas de Telecomunicações, que desempenham um papel fundamental na transmissão de informações.

O processo de incorporar um sinal que contém algum tipo de informação em outro sinal é chamado de modulação. O processo reverso, de extrair o sinal que contém a informação é chamado de demodulação.

Existem vários métodos de modulação e demodulação de sinais. Entre eles está a modulação em frequência - FM - que transmite informações por meio de uma portadora variando a sua frequência instantânea. Logo é imprescindível uma boa análise no domínio da frequência dos sinais que serão transmitidos a fim de assegurar uma satisfatória troca de informações.

Outra área da Engenharia Elétrica que utiliza a análise de sinais no domínio da frequência é a Qualidade de Energia que está intimamente ligada a um conjunto de alterações que estão susceptíveis a ocorrer no sistema elétrico, que se manifestam na forma de algum tipo de distúrbio de tensão, corrente ou desvio de frequência, resultando em má operação dos equipamentos dos clientes ligados ao sistema elétrico.

Uma ferramenta matemática capaz de ajudar a analisar os distúrbios no Sistema Elétrico de Potência e os sinais que transmitem informação via Modulação em Frequência - FM - é a Transformada Rápida de Fourier (FFT), que desempenha papel importante na Engenharia Elétrica, principalmente em processamento de sinais.

“A Transformada Discreta de Fourier apresenta uma representação no domínio da frequência de uma função do tempo, mantendo exatamente as mesmas informações da função no tempo.” (Michelin, C., 1998). A Transformada Rápida de Fourier é uma implementação eficiente da Transformada Discreta de Fourier.

Existe ainda, outra Transformada capaz de decompor um sinal descrito no domínio do tempo em diferentes componentes no domínio da frequência. Essa Transformada é conhecida como Transformada de Wavelet.

Esses algoritmos que permitem a descrição de um sinal no domínio do tempo em um sinal descrito no domínio da frequência podem ser bastante complexos e com custos computacionais elevados. Se as aplicações que envolvem a análise de sinais

no domínio da frequência necessitam de processamento de dados online é fundamental que os algoritmos possam ser executados de forma rápida e segura.

Porém a resolução de problemas científicos como a análise de sinais no domínio da frequência necessita de rotinas computacionais que possuem grande quantidade de instruções e geram considerável demanda de recursos, além da precisão em cálculos. Por isso esses algoritmos são implementados em processadores embarcados em hardwares. Portanto é necessário escolher um hardware que consiga executar de forma concisa, com redução de tempo de processamento e potência dissipada os algoritmos implementados. Olhando para tais requisitos a FPGA, que é um hardware configurável se mostra como uma escolha eficaz, pois possui capacidade de processamento elevado.

1.1 MOTIVAÇÃO

Foi desenvolvido um processador na UFJF, utilizando a linguagem de descrição de hardware Verilog, para ser embarcado em uma FPGA. O processador possui ULA – Unidade Lógica Aritmética - de ponto fixo ou ponto flutuante, como também pilhas de memória para dados e instruções.

A motivação é utilizar esse processador para realizar o cálculo da FFT de sinais que necessitam ter seus espectros de frequência analisados.

1.2 OBJETIVOS

O objetivo deste trabalho é modificar a estrutura do processador com ULA de ponto fixo, já desenvolvido para aperfeiçoar os cálculos da rotina da FFT.

Por conseguinte, é necessário que ocorra o endereçamento de dados de forma mais eficaz e direta e que a ULA do processador seja modificada para que se torne capaz de realizar cálculos com números complexos de forma mais eficiente.

1.3 METODOLOGIA

A metodologia utilizada é voltada para a implantação de um método de endereçamento de dados mais direto para o cálculo da FFT. Esse método é conhecido como inversão de bits, além disso, é necessário que a ULA do processador seja

modificada para realizar cálculos com números complexos. A modificação da ULA se da através da utilização do software Quartus II.

Além disso, foi implementado uma forma mais direta de endereçamento de dois endereços consecutivos em *arrays*. Como tais endereços representam o armazenamento da parte real e imaginaria dos números complexos calculados com a FFT, a ULA foi modificada para realizar acesso a dois componentes (parte real e imaginaria) do número complexo de forma mais direta.

Uma última modificação foi permitir o aproveitamento do endereçamento dual das memórias internas da FPGA, de modo a realizar multiplicações com menos número de instruções, uma vez que os dois operados passaram a ser acessados da memória simultaneamente.

1.4 ESTRUTURA DA MONOGRAFIA

A monografia foi dividida em oito capítulos descritos a baixo:

O primeiro capítulo trata da introdução, objetivo, motivação e metodologia que foi adotada no desenvolvimento do trabalho.

O segundo capítulo tem como objetivo oferecer uma visão geral do que é uma FPGA.

O terceiro capítulo traz uma revisão conceitual do que é a Transformada de Fourier. Nesse capítulo estão presentes as equações que descrevem de forma matemática a Transformada de Fourier.

O capítulo quatro traz a rotina da FFT na linguagem de alto nível C.

O capitulo cinco aborda de forma suave os compiladores.

O sexto capítulo fala sobre o compilador desenvolvido na UFJF, o SAFO.

O sétimo capítulo aborda os processos de otimização do hardware.

O oitavo capítulo apresenta as conclusões.

2 FPGA

Uma FPGA é um dispositivo construído com semicondutores, por blocos lógicos e conexões que podem ser programadas. Devido ao fato de existirem blocos lógicos que podem ser interconectados uma FPGA permite programar circuitos de diversas naturezas e tamanhos.

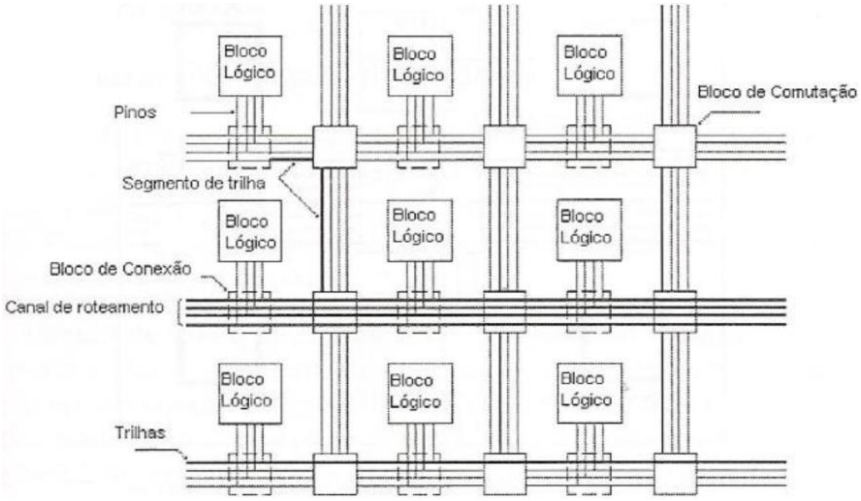
Uma hierarquia de interconexões programáveis permite que os blocos lógicos compreendidos na FPGA sejam conectados de acordo com a necessidade do projetista.

Uma grande vantagem da FPGA é que os blocos lógicos podem ser reconfigurados após a fabricação, permitindo assim que o usuário possa realizar a função lógica que deseja.

2.1 ARQUITETURA DA FPGA

A arquitetura da FPGA consiste de elementos lógicos (Logic Elements - LEs), blocos de memória, unidades multiplicadoras, bancos de I/O, além de PPLs universais.

Figura 1 - Estrutura de um FPGA

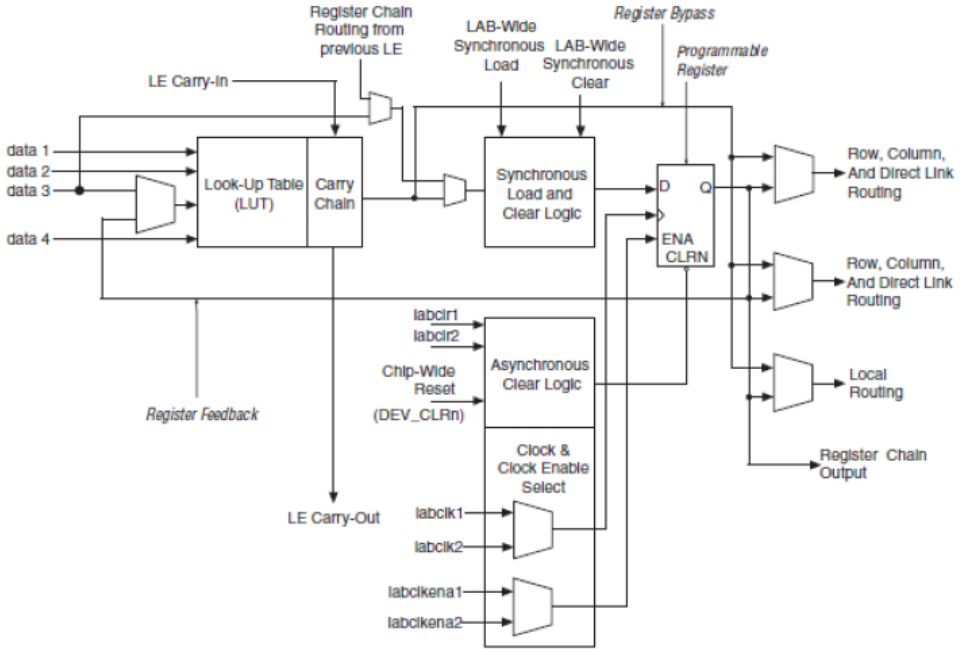


Fonte: www.altera.com/products/fpga

2.1.1 ELEMENTOS LÓGICOS (LES)

Os elementos lógicos (LEs) são as menores unidades que realizam alguma lógica nos dispositivos FPGA. Os blocos lógicos são formados, em geral, por uma tabela de buscas, conhecida como *look uptable* (LUTs) que pode realizar qualquer função de combinação de entradas de quatro variáveis, multiplexadores e registradores. Um exemplo de bloco lógico pode ser visto na Figura 2.

Figura 2 - Elementos de Um Bloco Lógico Programável.



Fonte: www.altera.com/products/fpga.

2.2 DESENVOLVIMENTO EM FPGA

As ferramentas existentes para o desenvolvimento de projetos com FPGA são bastante flexíveis e oferecem significativa liberdade para a especificação desses projetos, partindo da descrição esquemática e indo até a descrição comportamental.

As especificações comportamentais são criadas a partir das chamadas linguagens de descrição de hardware (Hardware Description Language - HDL) como: VHDL, Verilog, etc.

Em suma, uma linguagem descritiva de hardware é uma linguagem formal usada para modelar circuitos eletrônicos, e mais comumente, a lógica digital. As linguagens HDL permitem a validação de um projeto em FPGA, através de ferramentas de simulação como os *testbenches*. Existe ainda a possibilidade da visualização do arranjo dos elementos lógicos descritos pela HDL escolhida pelo usuário. O exposto mostra que as linguagens HDL permitem que o projetista possa ter o controle total, desde o inicio até a concepção final do projeto.

3 TRANSFORMADA DE FOURIER

"A transformada de Fourier é, em essência, uma ferramenta matemática que realiza a transição de sinais, do domínio do tempo para o da frequência, baseada na decomposição destes sinais em várias senóides e cossenoides." (Rolim, 2009, p.24).

3.1 TRANSFORMADA DISCRETA DE FOURIER

Muitas aplicações requerem reduzir o domínio de amostragem do sinal a ser estudado a um determinado intervalo, além disso, o crescimento expressivo dos dispositivos digitais fez com que as transformadas se adaptassem a era dos computadores, ou seja, que se tornassem discretas.

Uma das Transformadas discretas mais famosas e importantes é a Transformada Discreta de Fourier (DFT) que é a versão discreta da conhecida e aclamada Transformada de Fourier no tempo continuo.

Existe uma relação de dependência entre o sinal continuo que se deseja estudar e a DFT. Para que seja possível o cálculo da DFT de um sinal analógico x(t), usando um dispositivo digital, é necessário a conversão do mesmo para um sinal amostrado de base 2. Após isso, o sinal passa a ser representado por valores finitos e discretos compreendidos em uma determinada janela de tempo. A conversão é feita em intervalos de tempo conhecidos e a uma frequência de amostragem no mínimo duas vezes maior do que a frequência do sinal analógico. Ao final de todo esse processo, o sinal que antes era analógico, x(t), passa a ser uma sequência numérica x[n].

Em posse da sequência numérica x[n] é, então, possível calcular a DFT, seguindo a Equação 1.

Equação 1 – DFT

$$X(\Omega) = \sum_{n=0}^{N-1} x[n]\, e^{-j\Omega n}$$

Fonte: Lathi, 2010

Ω é uma variável continua que representa a frequência digital em n e está compreendida no intervalo [0, 2π]. A Equação 1 ainda não está adequada para ser utilizada por um computador, uma vez que é necessário um intervalo finito de valores

de Ω. Contudo feitas algumas modificações é possível reescrever a Equação 1 para que um computador possa resolvê-la, como pode ser visto abaixo na Equação 2.

Equação 2 – DFT Adequada ao Processamento Digital.

$$X(\Omega_k) = \sum_{n=0}^{N-1} x[n]\, e^{-j\Omega_k n}, k = 0,1, \ldots, M-1$$

Fonte: Lathi, 2010

Onde

Equação 3 – Frequência Digital.

$$\Omega_k = \frac{2\pi}{M} k$$

Fonte: Lathi, 2010.

O valor de M é comumente igual ao valor de N. Como os dispositivos eletrônicos digitais podem apenas trabalhar com um conjunto finito de números o sinal de entrada precisa ser quantizado para ser operado.

Sendo x[n], n = 1, 2,..., N-1 uma sequência numérica de tamanho N a Equação 2 pode ser reescrita como a Equação 4.

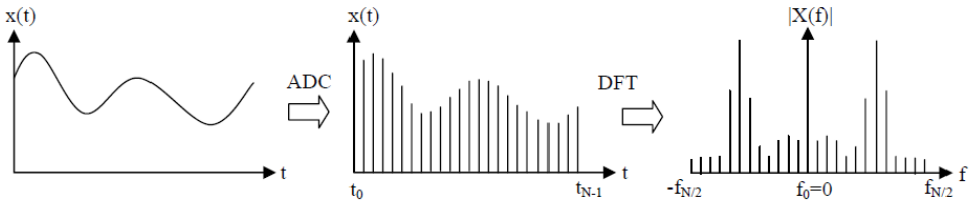
Equação 4 – Transformada Discreta de Fourier.

$$X(k) = \sum_{n=0}^{N-1} x[n]\, e^{-j\frac{2\pi}{N}kn}, \qquad k = 0,1, \ldots, N-1$$

Fonte: Lathi, 2010.

A Figura 3 exemplifica todo o processo de aquisição e adequação de um sinal para o cálculo da DFT.

Figura 3 - Processo para o cálculo da DFT.



Fonte: Dissertação de Mestrado por Arthur Umbelino Alves Rolim, UFPE, 2009.

3.1.1 EXEMPLO

Um sinal amostrado de tamanho n = 4, cujos componentes são [1, 2, 1, 0], de acordo com a Equação 4, possuem os componentes da Transformada Discreta de Fourier como mostrado na Equação 10.

Equação 5 – Sequência Numérica.

$$x[n] = \{1, 2, 1, 0\}$$

Fonte: Autor, 2016.

Equação 6 – DFT para k = 0.

$$k = 0, X[0] = \sum x[n]\, e^{0} = 1 + 2 + 1 + 0 = 4$$

Fonte: Autor, 2016.

Equação 7 – DFT para k = 1.

$$k = 1, X[1] = \sum x[n]\, e^{-\frac{j2\pi}{4}} = 1 + 2e^{-\frac{j\pi}{2}} + e^{-j\pi} = -2j$$

Fonte: Autor, 2016.

Equação 8 – DFT para k = 2.

$$k = 2, X[2] = \sum x[n]\, e^{-\frac{j2\pi 2}{4}} = 1 + 2e^{-j\pi} + e^{-j2\pi} = 0$$

Fonte: Autor, 2016.

Equação 9 – DFT para k = 3.

$$k = 3, X[3] = \sum x[n]\, e^{-\frac{j2\pi 3}{4}} = 1 + 2e^{-\frac{j\pi 3}{2}} + e^{-j3\pi} = 2j$$

Fonte: Autor, 2016.

Equação 10 – Componentes da DFT.

$$X[k] = \{4, -2j, 0, 2j\}$$

Fonte: Autor, 2016.

Uma rotina simples no MATLAB pode afirmar os resultados acima obtidos, como pode ser visto na Tabela 1.

Tabela 1 - Rotina em MATLAB.

```
clc; clear all; close all;
x = [1 2 1 0];% definição do vetor x de 4 posições
X = fft(x); % Construção do vetor X(k)
disp('O veotr x(k) é: ')
for k= 1: length(x)
    X(k)
end
O veotrx(k) é:
ans =
    4
ans =
  0.0000 - 2.0000i
ans =
    0
ans =
0.0000 + 2.0000i
```

Fonte: Autor, 2016.

3.2 TRANSFORMADA RÁPIDA DE FOURIER

Os grupos de algoritmos que computam de maneira extremamente veloz e eficiente os componentes da DFT são conhecidos como FFTs, Transformadas Rápidas de Fourier.

Embree e Danieli (1998) observam que a rotina da Transformada Rápida de Fourier (FFT) é capaz de substituir uma DFT de tamanho considerável pelo cálculo de várias outras DFTs menores. Contanto, essa substituição está condicionada a fatoração do número N em números múltiplos de dois. Isso é necessário, já que o número de operações requisitadas para o cálculo direto de uma DFT de N pontos é proporcional a N^2, logo o número de operações diminui rapidamente, de acordo com a partição da DFT em várias outras DFTs menores.

"A Transformada Discreta de Fourier pode, de fato, ser computada em $O(N)\ N\log_2 N$ operações com um algoritmo chamado Transformada Rápida de Fourier (FFT)." (PRESS; TEUKOLKY; VETTERLING; FLANNERY, 1992, p.504).

Tabela 2 - Comparação DFT e FFT.

N(Tamanho)	N^2(DFT)	$N\log_2 N$ (FFT)	Vantagem
2	4	2	2
4	16	8	2
8	64	24	2,67
16	256	64	4
32	1024	160	6,4
64	4096	384	10,67
128	16384	896	18,29
512	65536	2048	32
1024	262144	4068	56,89

Fonte: Autor, 2016.

E importante dizer que a FFT não é mais uma transformada e sim uma ferramenta computacional que possibilita avaliar os componentes de uma DFT de forma mais econômica, como pode ser visto acima na Tabela 2.

3.2.1 ALGORITMO DE COOLEY - TUKEY

Um dos algoritmos existentes para o cálculo da Transformada Rápida de Fourier é o algoritmo de Cooley – Tukey. Essa rotina segue a ideia de dividir uma transformada discreta em outras transformadas menores de forma recursiva, com o objetivo de reduzir significativamente o esforço computacional. (COLLEY; TUKEY; 1965).

O algoritmo de Cooley – Turkey além de ser $O(N)\, N\log_2 N$ não utiliza vetores auxiliares para o cálculo da DFT, comprovando assim, sua eficácia, uma vez que propicia a otimização de posições de memória do sistema em todo o decorrer do processo de cálculo. Outra vantagem do algoritmo é que os passos de interação são conhecidos.

3.3 DECIMAÇÂO NO TEMPO

Sabendo que a Transformada de Fourier obedece à Equação 4:

Equação 4 – Transformada Discreta de Fourier.

$$X(k) = \sum_{n=0}^{N-1} x[n]\, e^{-j\frac{2\pi}{N}kn}, \qquad k = 0,1,\dots,N-1$$

Fonte: Lathi, 2010.

Como N é par, o somatório $\sum_{n=0}^{N-1} x[n]\ e^{-j\frac{2\pi}{N}kn}$ pode ser dividido em dois, uma parte representando os termos pares: n = 2n' e outra parte representando os termos impares: n = 2n' + 1. Em que n' = 1, 2,..., (N/2)-1. Logo:

Equação 11 – Decimação no Tempo.

$$X(k) = \sum_{n=0}^{N-1} x[n]\, e^{-j\frac{2\pi}{N}kn}$$

$$X(k) = \sum_{n'=0}^{\frac{N}{2}-1} x[2n']\, e^{-j\frac{2\pi}{N}k(2n')} + \sum_{n'=0}^{\frac{N}{2}-1} x[2n'+1]\, e^{-j\frac{2\pi}{N}k(2n'+1)}$$

$$X(k) = \sum_{n'=0}^{\frac{N}{2}-1} x[2n']\, e^{-j\frac{2\pi}{\frac{N}{2}}k(n')} + W^k \sum_{n'=0}^{\frac{N}{2}-1} x[2n'+1]\, e^{-j\frac{2\pi}{\frac{N}{2}}k(n')}$$

$$X(k) = X_k^e + W^k X_k^o$$

$$X(k+1) = X_k^e - W^k X_k^o$$

$$W_N(k) = e^{j\frac{2\pi k}{N}}$$

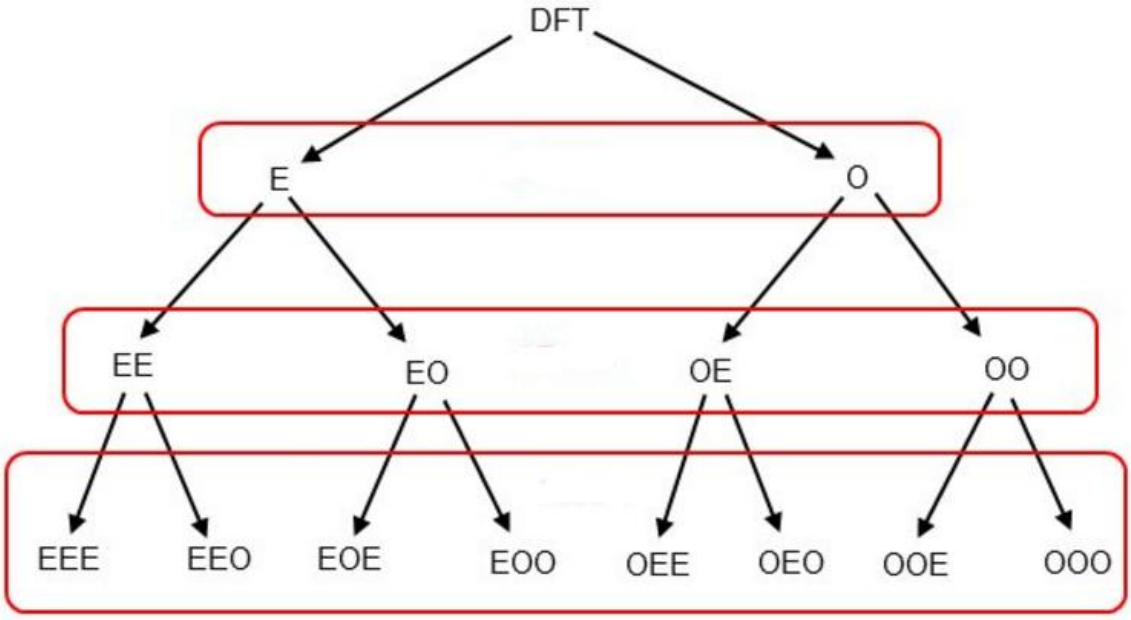
Fonte: Numerical Recipes in C. The Art of Scientific Computing, 1998.

Na Equação 11 temos que X_k^e representa o k-ésimo componente da Transformada de Fourier de tamanho N/2 formado pelos componentes pares da sequência original x[n], enquanto X_k^o corresponde aos componentes impares. É importante notar que k varia de 0 a N e não apenas até N/2. Contudo as transformadas X_k^e e X_k^o são periódicas em k com tamanho N/2, logo para obter X(k), X_k^e e X_k^o são repetidas, cada uma, por dois ciclos.

O processo descrito acima pode ser usado de forma recursiva, ou seja, tendo resolvido a questão de computar a transformada de X(k) em duas transformadas menores, X_k^e e X_k^o, é possível fazer a mesma redução de X_k^e decompondo-o em outras duas partes: X_k^{ee} e X_k^{eo}, o mesmo vale para X_k^o.

O processo de dividir componentes do domínio do tempo em amostras pares e impares é conhecido como processo de subdivisão de Denielson – LanczosLemma.

Figura 4 - Subdivisão de Denielson - LanczosLemma.

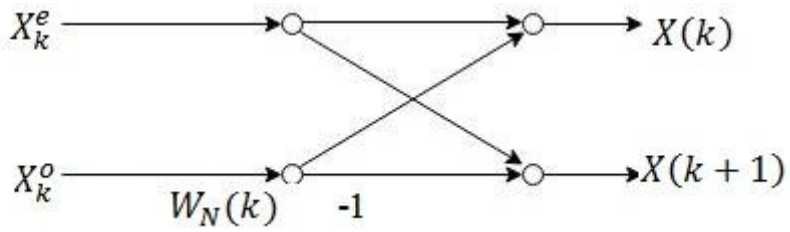


Fonte: Autor, 2016.

É importante ressaltar que N deve ser uma potência de dois, pois casos em que isso não se aplica o cálculo dos componentes da transformada se tornam complicados. Se a restrição a N for respeitada deve-se aplicar Denielson – LanczosLemma até a obtenção de uma DFT de dois pontos, com isso o número de operações necessárias será proporcional a $O(N)N\log_2 N$.

Ao final do processo de Denielson – LanczosLemma é possível obter dois elementos de X(k) a partir de um elemento de X_k^e e um elemento de X_k^o com apenas uma multiplicação e duas adições complexas. Essas operações são chamadas de *butterfly*, que são aplicadas a dois pontos da série de cada vez, tendo como consequência, uma redução no número de operações requeridas.

Figura 5 – Buterfly.



Fonte: Autor, 2016.

Para aplicar o algoritmo de decimação no tempo é necessário fazer uso da técnica bit – invertido aplicado ao índice dos vetores de dados de entrada. Essa técnica consiste em fazer com que os bits menos significativos de um número se tornem os mais significativos.

O uso do bit – invertido é de extrema importância, pois os índices do vetor de dados de entrada são divididos sequencialmente em termos pares e impares até que se obtenha uma única combinação de elementos pares e impares.

Considerando uma serie com oito amostras temos:

Tabela 3 - Subdivisão Sequencial em Termos Pares e Ímpares.

Subdivisão			
0	1	2	Índice
0	0	0	
1	2	4	
2	4	2	
3	6	6	
4	1	1	
5	3	5	
6	5	3	
7	7	7	

Separação em índices pares em cor vermelha
Separação em índices ímpares em cor azul

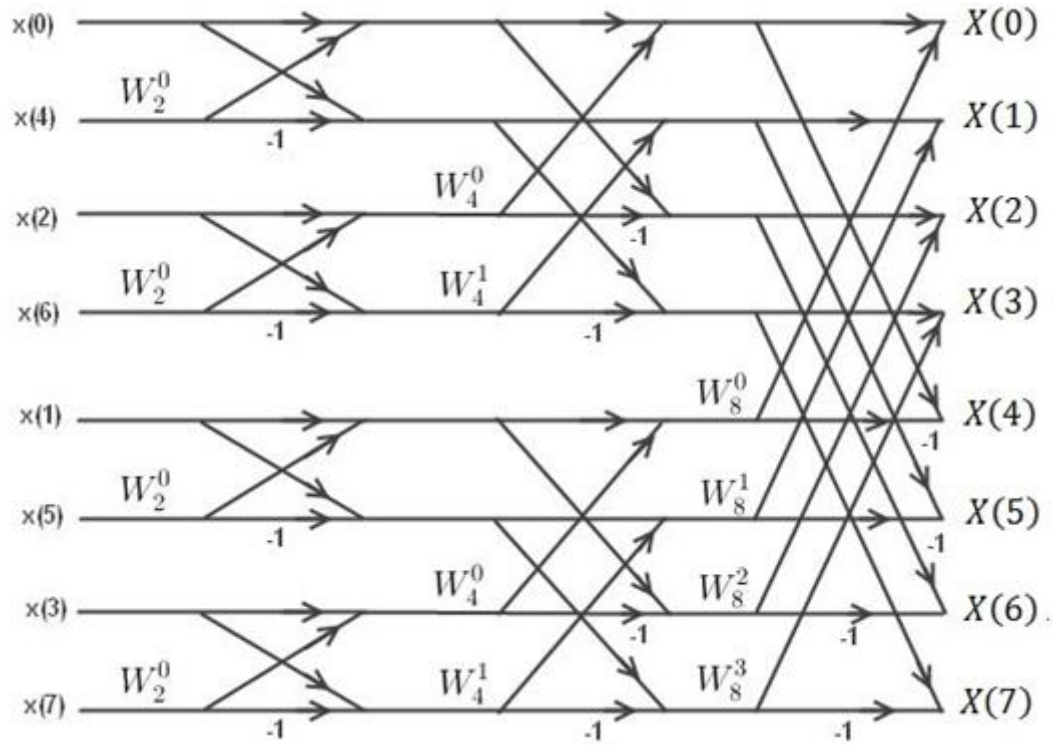
Fonte: Autor, 216.

Tabela 4 - Ordenação por Bit-Invertido.

Índice	0	1	2	3	4	5	6	7
Equivalente em binário	000	001	010	011	100	101	110	111
Bit - invertido	000	100	010	110	001	101	011	111
Índice em bit - invertido	0	4	2	6	1	5	3	7

Fonte: Autor, 2016.

Figura 6 - Buterfly para Oito Amostras.



Fonte: Numerical Recipes in C. The Art of Scientific Computing, 1998.

4 ALGORITMO PARA FFT EM C

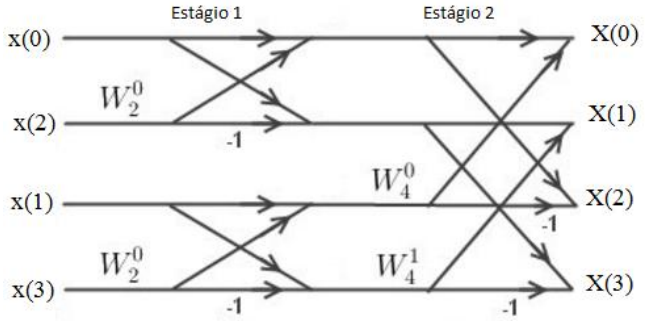
A rotina correspondente a FFT é descrita em um compilador, portanto faz-se necessário o estudo de um código escrito em uma linguagem de alto nível com o objetivo de fomentar a base teórica e aumentar os conhecimentos a cerca de algoritmos.

O algoritmo que calcula a FFT possui como entradas o vetor data [1 ...2*nn] que será alvo da FFT, o número de amostras complexas, nn, e a variável isign. Se isign for 1 a Transformada Direta de Fourier será calculada, se isign for igual a -1, então a Transformada Inversa de Fourier será calculada.

Primeiramente é aplicada ao vetor de entrada uma sequência de comandos que irá fazer com que o vetor *data* tenha os argumentos organizados segundo a técnica do bit – invertido.

Em seguida a rotina entra em um *loop* externo que percorre os estágios de cálculos da FFT. Por exemplo, para um vetor *data* com 4 componentes complexos os estágios de cálculos serão como na figura 7.

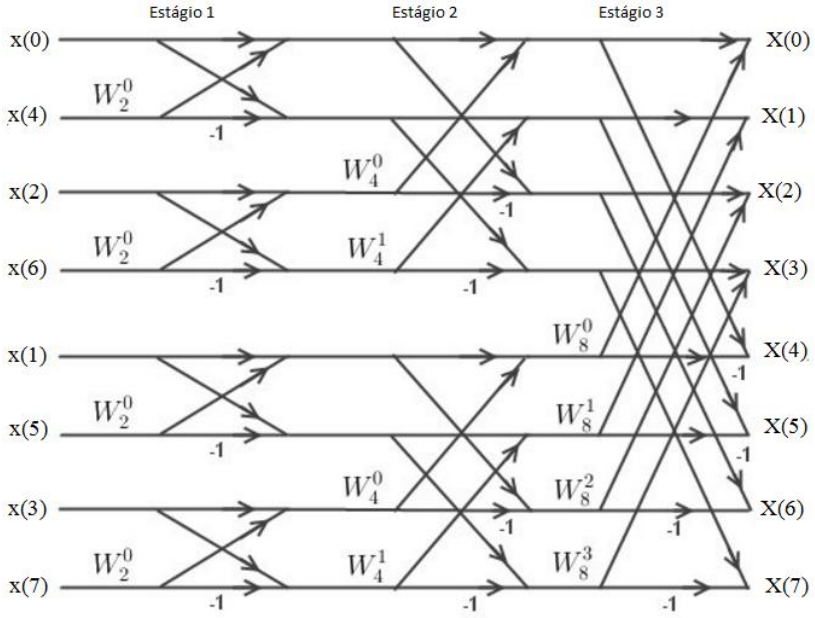
Figura 7 - Estágios para Quatro Amostras.



Fonte: Autor, 2016.

Se *data* possuir 8 componentes complexos os estágios de cálculos serão como na figura 8 e assim por diante. Logo o número de estágios de multiplicações e adições é $\log_2 nn$.

Figura 8 - Estágios para Oito Amostras.



Fonte: Autor, 2016.

Outro *loop* é definido, esse tem o objetivo de determinar o índice do vetor *data* que será acessado e também atualizar algumas variáveis que são necessárias para que em seguida outro *loop* possa realizar, de fato, o cálculo necessário para a obtenção dos parâmetros da Transformada de Fourier.

Por fim, um último *loop* é definido para que possa ser efetuado o cálculo dos componentes da FFT. Nesse *loop* são calculadas variáveis auxiliares denominadas *Twiddle Factor* para que então os termos do vetor *data* possam ser atualizados.

Figura 9 - Sequenciamento para Cálculo da FFT.

Fonte: Autor, 2016.

Como *nn* é o número de amostras complexas o tamanho real do vetor data [1..2*nn] é duas vezes o número de amostras complexas, ou seja, data [1] corresponde a componente real na frequência f_0, ao passo que data [2] representa a componente complexa da amostra na f_0, data [3] é a componente real na frequência f_1, enquanto data [4] é a componente complexa emf_1, e assim até data [2*nn-1] que é a parte real em f_{nn-1} e data [2*nn] como a parte imaginária em f_{nn-1}. O algoritmo retorna um vetor *data* como os componentes da FFT com o mesmo número de amostras complexas *nn* do vetor de entrada.

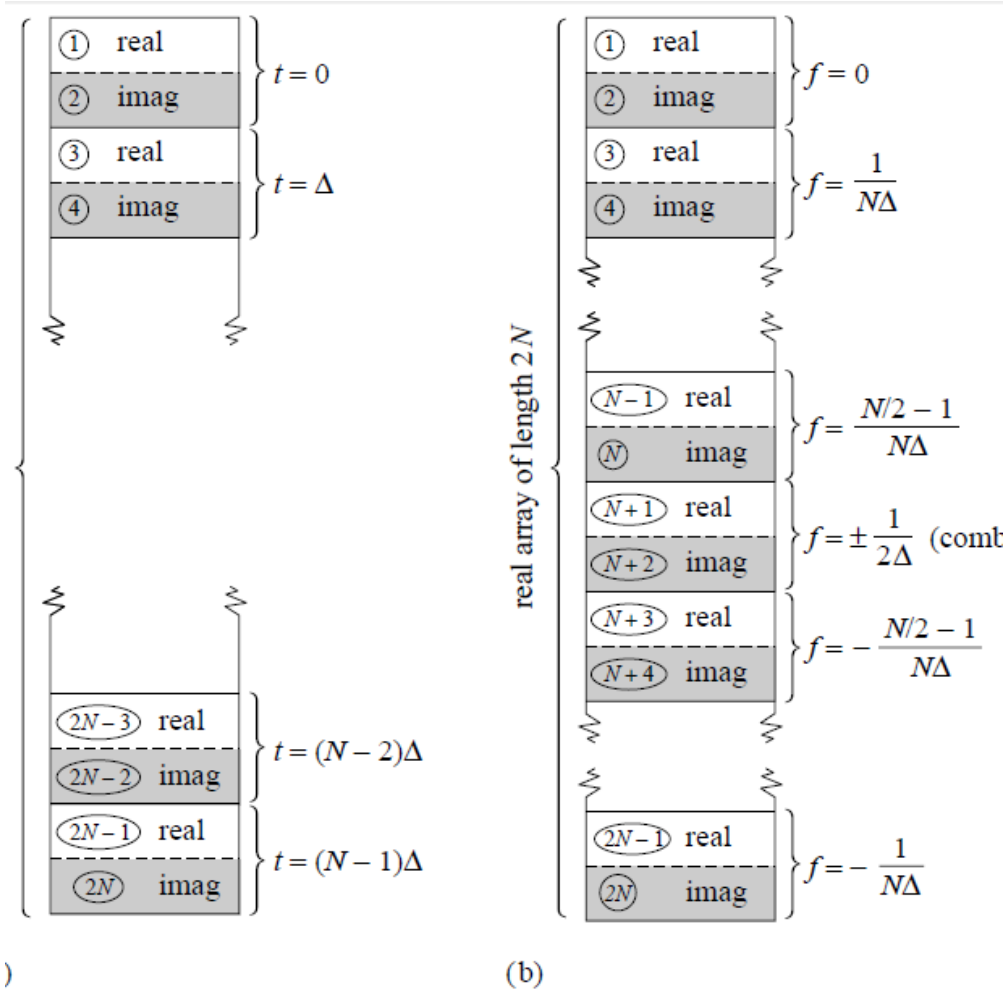
O vetor que contém o espectro complexo de Fourier para N valores de frequência é organizado da seguinte forma:

- Partes reais e imaginárias são agrupadas de forma alternada, começando com um termo real seguido por um imaginário e assim sucessivamente até o fim do vetor.
- As componentes do espectro na frequência zero estão alocadas logo no inicio do vetor, data [1] e data [2].

- O vetor *data* é preenchido até as frequências positivas mais altas, culminando na frequência mais positiva que tem o mesmo valor em modulo da frequência mais negativa.
- As frequências negativas seguem a partir da segunda frequência mais negativa até a menor frequência depois do zero.

O exposto anteriormente é ilustrado na Figura 10.

Figura 10 - – Vetores de Entrada e Saída da Rotina FFT.
a) Vetor de entrada de tamanho N. b) Vetor de saída de tamanho N.



Fonte: Numerical Recipes in C. The Art of Scientific Computing, 1998.

"O algoritmo usado para o cálculo da FFT é baseado no algoritmo escrito por N. M. Brenner". (PRESS; TEUKOLKY; VETTERLING; FLANNERY, 1992, p.506).

Tabela 5 - Algoritmo FFT.

```
void  FFT (float data[], unsigned long nn,intisign)
{
    unsigned long n,mmax,m,j,istep,i;
    double wtemp,wr,wpr,wpi,wi,theta;
floattempr,tempi;
n=nn * 2;j=0;
for (i=0;i<n/2;i+=2)
  {
     if (j >i)
     {
swap(data[j],data[i]); //swap the real part
          swap(data[j+1],data[i+1]); //swap the complex part
// checks if the changes occurs in the first half
          // and use the mirrored effect on the second half
          if((j/2)<(n/4))
          {
                    swap(data[(n-(i+2))],data[(n-(j+2))]); //swap the real part
//swap the complex part
swap(data[(n-(i+2))+1],data[(n-(j+2))+1]);
}
}
     m=n/2;
     while (m >= 2 && j >= m)
     {
          j -= m;
          m = m/2;
     }
      j += m;
  }
  //Danielson-Lanczos routine
```

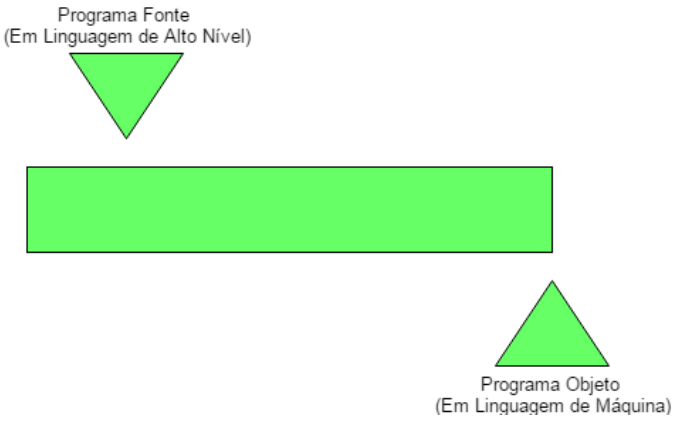
```
mmax=2;
   //external loop
   while (n >mmax)
   {
istep = mmax<<  1;
        theta=isign*(6.28318530717959/mmax);
wtemp=sin(0.5*theta);
wpr = -2.0*wtemp*wtemp;
wpi=sin(theta);
wr=1.0;
wi=0.0;
        //internal loops
        for (m=1;m<mmax;m+=2)
        {
           for (i= m;i<=n;i+=istep)
{
j=i+mmax;
tempr=wr*data[j-1]-wi*data[j];
tempi=wr*data[j]+wi*data[j-1];
data[j-1]=data[i-1]-tempr;
data[j]=data[i]-tempi;
data[i-1] += tempr;
data[i] += tempi;
}
wr=(wtemp=wr)*wpr-wi*wpi+wr;
wi=wi*wpr+wtemp*wpi+wi;
}
                                mmax=istep;    }}
```

Fonte: Numerical Recipes in C. The Art of Scientific Computing, 1998.

5 COMPILADORES

Um Compilador é um programa que traduz um programa escrito em linguagem de alto nível para um programa equivalente em código de máquina para o processador.

Figura 11 - Papel do Compilador.

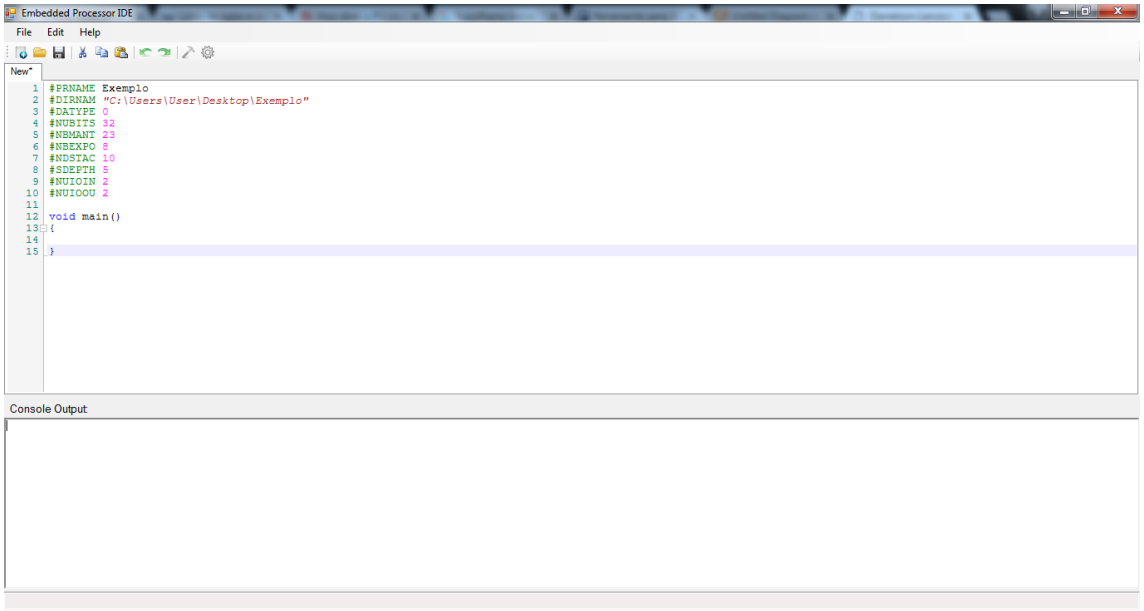


Fonte: Autor, 2016.

Existem diversos compiladores, entre eles o Clang, Visual C++, Dev C++, Code::BLocks entre outros.

Nesse projeto o compilador usado é o SAFO que foi desenvolvido pelo Professor Dr. Luciano Manhães de Andrade Filho, cujo principal objetivo é traduzir um código em linguagem de alto nível para a linguagem Assembly, que um processador descrito em Verilog embarcado em uma FPGA consegue assimilar e realizar o que lhe fora determinado.

Figura 12 - Interface do Compilador SAFO.



Fonte: UNIVERSIDADE FEDERAL DE JUIZ DE FORA, 2016.

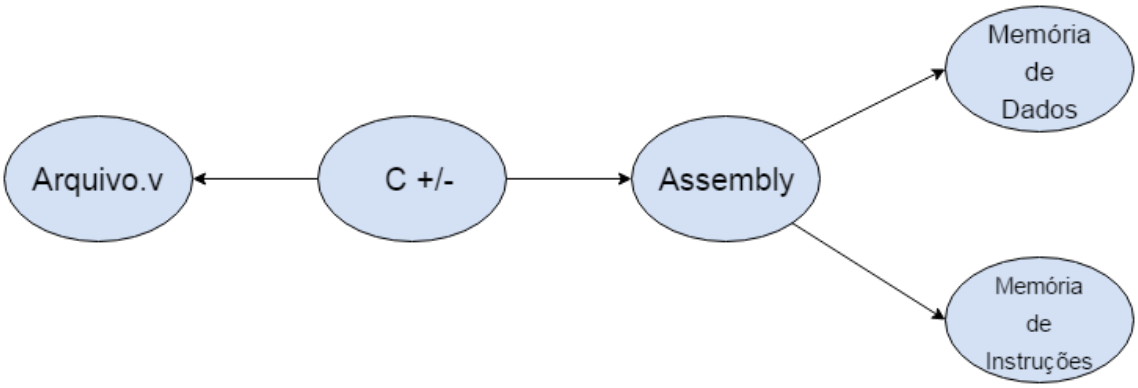
6 COMPILADOR SAFO

O SAFO possui sua própria linguagem de alto nível, batizada de C +/-. O C+/- é bastante similar ao C, uma vez que possui algumas das funcionalidades presentes no C e C++ como a função de repetição *while*, as funções de comparação *if* e *else* e também suporta operações de subtração, adição, multiplicação e subtração.

Após o código ser escrito no SAFO, ser salvo e compilado, são gerados quatro arquivos: dois arquivos de extensão .mif, um arquivo .v e uma arquivo .asm.

Os arquivos de extensão .mif são arquivos de inicialização das memórias do processador, que é baseado em uma arquitetura Harward, enquanto os arquivos .v são arquivos Verilog que parametrizam o processador. Já os arquivos .asm são arquivos do tipo Assembly e são usados como arquivos temporários para geração dos arquivos .mif por meio de um compilador assembler. A figura 13 ilustra o processo de geração desses arquivos pelo SAFO.

Figura 133 - Arquivos Gerados Pelo SAFO.



Fonte: Autor, 2016.

O Assembly é uma notação textual de um código de máquina que uma arquitetura de computador usa para executar tarefas predefinidas.

6.1 ALGORITMO PARA FFT EM C+/-

Como as linguagens C e C+/- são bem parecidas não fica difícil adaptar um código escrito para ser executado em um compilador C ou C++ em código executável no SAFO, porém é preciso tomar algumas precauções, já que o SAFO não possui a biblioteca *math.h*, logo todas as variáveis que são definidas por operações de seno e cosseno devem ser declaradas como vetores preenchidos previamente.

A Tabela 6 mostra o código apresentado na Tabela 5 adaptado para a linguagem C+/-, para o exemplo de uma FFT de 16 pontos em ponto fixo.

Tabela 6 - Algoritmo FFT em C+/-.

```
int data[16];
int sin1[3];
int sin2[3];
int mmax,m,j,istep,i,wtemp,wr,wpr,wpi,wi,tempr,tempi,tmp,ind,n,a,aux;
void main()
{
   sin1[0] = 256; //256 1x2^8
   sin1[1] = 181;//181; // 0.7071067811 x 2^8;
   sin1[2] = 98;//98;  // 0.3826834323 x 2^8;
```

```
   n    = 16;
   sin2[0] = 0;
   sin2[1] = 256;//256; // 1x2^8;
   sin2[2] = 181;//181; // 0.7071067811 x 2^8;

data[0] = 1;  data[8] = 1;
data[1] = 0;  data[9] = 0;
data[2] = 2;  data[10 ] = 2;
data[3] = 0;  data[11] = 0;
data[4] = 1;  data[12] = 1;
data[5] = 0;  data[13] = 0;
data[6] = 2;  data[14] = 2;
data[7] = 0;  data[15] = 0;

j=0;i=0;
while(i<n/2)
  {
     if (j >i)
     {
          //swap the real part
          //swap(data[j],data[i]);
tmp      = data[j];
data[j]   = data[i];
data[i]   = tmp;
          //swap the complex part
//swap(data[j+1],data[i+1]);
tmp      = data[j+1];
data[j+1]   = data[i+1];
data[i+1]   = tmp;
          // checks if the changes occurs in the first half
          // and use the mirrored effect on the second half
          if((j/2)<(n/4))
          {
                    //swap the real part
```

```
//swap(data[(n-(i+2))],data[(n-(j+2))]);
tmp      = data[(n-(j+2))];
data[(n-(j+2))]   = data[(n-(i+2))];
data[(n-(i+2))]   = tmp;
                        //swap the complex part
//swap(data[(n-(i+2))+1],data[(n-(j+2))+1]);
tmp      = data[(n-(j+2))+1];
data[(n-(j+2))+1]   = data[(n-(i+2))+1];
data[(n-(i+2))+1]   = tmp;
}
}
m=n/2;
while (m >= 2 && j >= m)
      {
         j = j- m;
           m = m/2;
      }
        j = j+ m;
i = i+ 2;
   }



mmax = 2;
ind  = 0;
   while (mmax< n)
   {
istep = mmax*2;
wtemp = sin1[ind];
wpr   = -2*wtemp*wtemp;
wpi   = sin2[ind];
wr    = 256;
wi    = 0;
ind   = ind+1;
      m = 1;
```

```
      while (m <mmax)
      {
i = m;
while(n >= i)
{
j = i + mmax;

tempr = wr*data[j-1] - wi*data[j];
tempi = wr*data[j] + wi*data[j-1];

data[j-1]  = (256*data[i-1] - (tempr>> 8)) >> 3;
data[j]    = (256*data[i]   - (tempi>> 8)) >> 3;
data[i-1]  = (256*data[i-1] + (tempr>> 8)) >> 3;
data[i]    = (256*data[i]   + (tempi>> 8)) >> 3;

i = i + istep;
         }

wtemp = wr;
wr = (((wtemp>> 4)*(wpr>> 4) - (wi>> 4)*(wpi>> 4)) >> 8) + wr;
wi = (((wi>> 4)*(wpr>> 4) + (wtemp>> 4)*(wpi>> 4)) >> 8) + wi;
         m   = m+2;
      }
mmax = istep;
   }
   out(0,data[0]);  out(0,data[1]);
   out(0,data[2]);  out(0,data[3]);
   out(0,data[4]);  out(0,data[5]);
   out(0,data[6]);  out(0,data[7]);
   out(0,data[8]);  out(0,data[9]);
   out(0,data[10]);  out(0,data[11]);
   out(0,data[12]);  out(0,data[13]);
   out(0,data[14]);  out(0,data[15]);}
```

Fonte: Autor, 2016.

O código apresentado na Tabela 6 é composto por três partes. Na primeira parte ocorre a atribuição de valores aos vetores auxiliares sin1 e sin2 e ao vetor *data* que contêm as amostras complexas de dados que serão alvo da FFT.

O algoritmo para o cálculo da FFT exige que as amostras complexas sejam reordenadas de acordo com a técnica do bit – invertido. O procedimento consiste em varrer todo o vetor de dados através de uma função de repetição e então trocar os termos do vetor, de acordo com o exposto na Tabela 4.

O procedimento de reordenação do vetor de dados é dividido em dois. Um procedimento exclusivo para a reordenação dos termos reais das amostras complexas e outro procedimento que irá reorganizar os componentes imaginários das amostras complexas. A parte de reorganização das amostras complexas é definida como a segunda parte do algoritmo apresentado na Tabela 6.

Na terceira parte do algoritmo apresentado na Tabela 6 é que de fato ocorre o cálculo da FFT, essa parte é conhecida como rotina de Danielson-Lanczos.

Nessa parte algumas variáveis auxiliares são declaradas, porém existem algumas que merecem atenção, são elas:

- mmax que representa o tamanho máximo de amostras que serão analisadas;
- wpi que representa a parte imaginária de $W_N(k) = e^{j\frac{2\pi k}{N}}$;
- wpr que representa a parte real de $W_N(k) = e^{j\frac{2\pi k}{N}}$;
- tempr é a parte real de $W^k X_k^o$;
- tempi é a parte imaginária de $W^k X_k^o$.

Tabela 6- Componentes Reais e Imaginárias.

data[j]=data[i] - tempr;	data[i] =data[i] + tempi;	Componentes Imaginárias
data[j-1]=data[i] - tempi;	data[i-1] =data[i-1] + tempr;	Componentes Reais

Fonte: Autor, 2016.

Os elementos da primeira linha da Tabela 7 representam os termos $X(k+1)$ na Equação 11, ao passo que os termos da segunda linha da Tabela 7 representam os termos $X(k)$ na Equação 11.

A Tabela 8 mostra um comparativo do número de instruções geradas para executar o algoritmo com o número de elementos lógicos consumidos pela FPGA.

Tabela 8 – Instruções x Elementos Lógicos.

Número de Instruções	Elementos Lógicos
521	497

Fonte: Autor, 2016.

A parcela do algoritmo apresentado na Tabela 6 que consome mais processamento é a parte correspondente ao cálculo dos coeficientes da *buterfly*, ou seja, cálculos complexos. Esse trecho de código é apresentado na Tabela 9.

Tabela 9 – Cálculo dos Componentes da Buterfly, Cálculos Complexos.

```
tempr = wr*data[j-1] - wi*data[j];
tempi = wr*data[j] + wi*data[j-1];
```

Fonte: Autor, 2016.

A sequência de instruções, em Assembly, correspondente ao trecho de código apresentado na Tabela 9 gerou um total de 32 instruções e pode ser conferido na Tabela 10.

Tabela 10 – Sequência de Instruções Para o Cálculo dos Componentes da Buterfly.

LOAD fftwr
PLD fftj
PLD 1
SSUB
PUSH
SRF
LOAD fftdat
SMLT
PLD fftwi
PLD fftj
PUSH
SRF
LOAD data
SMLT
SSUB
SET ffttempr
LOAD fftwr
PLD fftj
PUSH
SRF
LOAD data
SMLT
PLD fftwi
PLD fftj
PLD 1
SSUB
PUSH
SRF
LOAD data
SMLT
SADD
SET ffttempi

Fonte: Autor, 2016.

7 OTIMIZAÇÂO

Como apresentado na Tabela 8, foi consumido um número razoável de elementos lógicos e foi gerada, também, uma quantidade considerável de instruções.

Com o intuito de diminuir o número de instruções geradas e tornar o processador de ponto fixo capaz de realizar operações complexas de forma mais eficiente, tornando-o mais robusto e eficiente, a estrutura do hardware foi modifica.

7.1 REORDENAÇÃO DOS DADOS E SUBTRAÇÂO COM A PILHA

O algoritmo apresentado na seção 6.1 é eficiente, porém gera 521 instruções e consome 497 elementos lógicos. Objetivando otimizar o acesso aos dados para o cálculo da FFT, diminuindo assim a quantidade de instruções geradas, foi desenvolvido um comando que dispensa toda a parte de reordenação (bit – invertido) dos elementos do vetor de amostras complexas alvo do algoritmo da FFT.

O comando que substitui toda a parte do bit – invertido é o: “|] ”. Quando esse comando é escrito no compilador SAFO, é então gerado um comando em Assembly que o decodificador de instruções do processador entende que deve enviar para a ULA os dados que serão processados obedecendo a lógica do bit – invertido, como mostra a Tabela 4.

A instrução em Assembly gerada é a SRFR, logo quando aparece um comando SRFR significa que foi encontrado em alguma parte do código o comando “|] ”.

O *opcode* correspondente ao comando “|]” no decodificador de instruções do processador é o 17.

Quando o *opcode* do decodificador de instruções vale 17 acontecem as seguintes operações:

- A ULA não executa nenhuma instrução;
- O sinal *brev* vale 1;
- O sinal *srf* vale 1;
- Ocorre um “pop” na pilha de dados.

O sinal *srf* é utilizado para acessar dados de uma posição de um vetor, enquanto o sinal *brev* é o responsável por inverter a ordem dos bits dos dados. Uma vez

que os sinais *srf* e *brev* são 1 ao mesmo tempo, é garantido o acesso aos dados de um vetor de acordo com a lógica do bit – invertido.

Abaixo segue um exemplo de como o exposto a cima acontece.

Figura 14 - Exemplo.

Vetor X.

$$X = [20\ 30\ 40\ 50\ 60\ 70\ 80\ 90]$$

Acesso Normal ao Conteúdo do Vetor X.

$X[0] = 20$ $X[4] = 60$
$X[1] = 30$ $X[5] = 70$
$X[2] = 40$ $X[6] = 80$
$X[3] = 50$ $X[7] = 9$

Acesso ao Conteúdo do Vetor X com o comando "|]".

$X|0] = 20$ $X|4] = 30$
$X|1] = 60$ $X|5] = 70$
$X|2] = 40$ $X|6] = 50$
$X|3] = 80$ $X|7] = 90$

Fonte: Autor, 2016.

Outra modificação feita no processador foi a inserção da operação de subtração com a pilha de dados, SSUB. As únicas modificações feitas na ULA para que o procedimento de subtração com a pilha de dados fosse criado foram a introdução de mais um operando, e a introdução de uma saída a mais no multiplexador de operações aritméticas presente na ULA, a saída *sub*.

Quando o compilador SAFO identifica um padrão "exp – exp", em que "exp" pode ser um valor numérico ou uma variável, é gerado um comando em Assembly denominado SSUB.

O *opcode* correspondente à instrução SSUB no decodificador de instruções do processador é o 24.

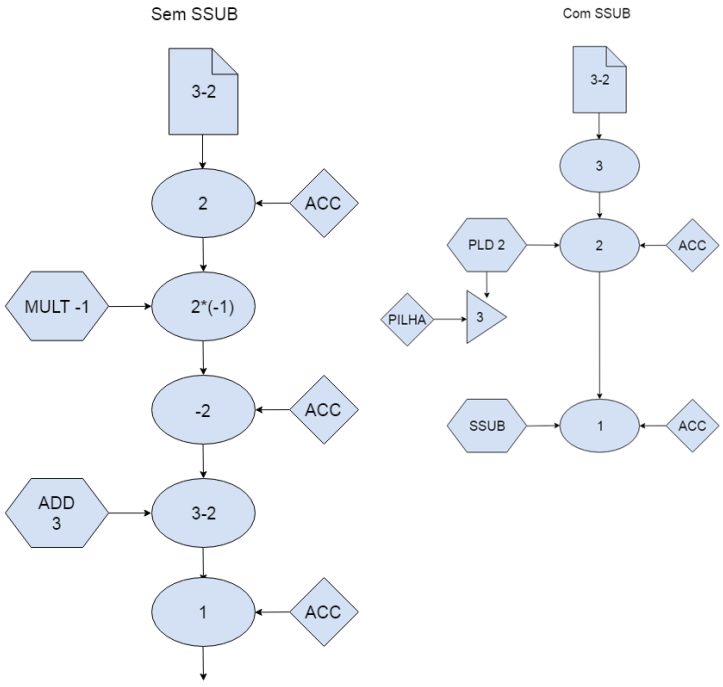
Antes da introdução da subtração com a pilha, para que uma operação de subtração acontecesse ocorria em primeiro lugar uma multiplicação seguida por uma operação de adição.

Por exemplo, se fosse necessário realizar a operação matemática “3 – 2”, primeiro era carregado no ACC – acumulador - o valor 2, em seguida a instrução MULT – instrução de multiplicação – multiplicava o argumento presente no acumulador por menos 1, por fim o resultado dessa operação é somado – Instrução ADD – ao número 3, resultando em 1.

A mesma operação exemplificada acima realizada após a introdução da instrução SSUB ocorre da seguinte maneira: O valor 3 é carregado no acumulador, em seguida ele é armazenado no topo da pilha de dados e então o número 2 é carregado no acumulador. Prontamente o comando SSUB realiza a subtração do dado presente no topo da pilha de dados, 3, com o número presente no acumulador, 2. Resultando em 3 – 2 = 1.

Um esquemático do antes e depois da introdução da operação de subtração com a pilha está presente na Figura 15 e as respectivas instruções em Assembly estão na Tabela 11.

Figura 15 - Exemplo.



Fonte: Autor, 2016.

Tabela 11 – Comparativo.

Sem SSUB	Com SSUB
LOAD 2	LOAD 3
MULT -1	PLD 2
ADD 3	SSUB
SET a	SET a
LOAD 0	LOAD 0
PLD a	PLD a
OUT	OUT

Fonte: Autor, 2016.

Na Tabela 12 está presente o resultado da introdução do comando "|]" e da instrução de subtração com a pilha de dados em relação ao número total de instruções geradas para executar o algoritmo e ao consumo de elementos lógicos pela FPGA.

Tabela 12 – Instruções x Elementos Lógicos.

Número de Instruções	Elementos Lógicos
408	569

Fonte: Autor, 2016.

Ao utilizar o comando "|]" a parte do código correspondente a determinação dos componentes da *buterfly*, que é a parte que exige mais esforço e gasta mais processamento, fica como apresentado na Tabela 13.

Tabela 13 – Cálculo dos Componentes da Buterfly, Com o Comando "|]".

```
tempr = wr*data|j-1] - wi*data|j];
tempi = wr*data|j] + wi*data|j-1];
```

Fonte: Autor, 2016.

A sequência de instruções, em Assembly, correspondente ao trecho de código apresentado na Tabela 13, incorporando o comando "|]" e a instrução de subtração com pilha de dados, gerou um total de 32 instruções e pode ser conferido na Tabela 14.

Tabela 14 – Sequência de Instruções Para o Cálculo dos Componentes da Buterfly.

LOAD fftwr
PLD fftj
PLD 1
SSUB
PUSH
SRFR
LOAD data
SMLT
PLD fftwi
PLD fftj
PUSH
SRFR
LOAD data
SMLT
SSUB
SET ffttempr
LOAD fftwr
PLD fftj
PUSH
SRFR
LOAD data
SMLT
PLD fftwi
PLD fftj
PLD 1
SSUB
PUSH
SRFR
LOAD data
SMLT
SADD
SET ffttempi

Fonte: Autor, 2016.

7.2 INDEXAÇÂO

A Tabela 13 mostra um padrão de acesso ao conteúdo do vetor de dados. O acesso ao conteúdo ocorre sempre seguindo os índices “i” e “j” ou o valor desses índices menos 1, com o propósito de reduzir as instruções geradas para o cálculo dos componentes da *buterfly* e aumentar a velocidade de processamento do hardware, uma nova instrução foi criada.

A instrução criada, o comando “^”, foi introduzida no processador com o objetivo de substituir toda operação correspondente a subtração de uma variável menos 1.

Em suma o comando “^” pega um valor armazenado na memória de dados ou um valor externo, realiza a operação de subtração de tal valor menos 1 e carrega o resultado de toda essa operação na entrada do acumulador, para que no próximo *clok* esse valor esteja presente na saída do próprio acumulador.

A modificação feita no decodificador de instruções foi bastante simples, houve apenas a inclusão de dois novos *opcode* de número 25 e 26. O *opcode* 25, cujo correspondente em Assembly é a instrução LOADM1, faz a operação “^” com os dados provenientes da memória de dados, ao passo que o *opcode* 26, cujo correspondente em Assembly é a instrução PLDM1, realiza os mesmos procedimentos, porém com os dados presentes na pilha de dados.

A ULA do processador foi modificada para suportar a operação “^”. As modificações feitas foram:

- Aumento do número de bits do operando da ULA de 3 para 4 bits;
- Adição de uma saída a mais para o multiplexador de operações aritméticas, a saída *ldm1*.

A Tabela 15 apresenta o resultado da introdução do comando “^” com relação ao número de instruções exigidas para realizar o código que calcula os componentes da FFT e também o número elementos lógicos utilizados pela FPGA.

Tabela 15 – Instruções x Elementos Lógicos.

Número de Instruções	Elementos Lógicos
397	577

Fonte: Autor, 2016.

Ao utilizar o comando “^”, a parte do código correspondente à determinação dos componentes da *buterfly*, que é a parte que exige mais esforço e gasta mais processamento, fica como apresentado na Tabela 16.

Tabela 16 – Cálculo dos Componentes da Buterfly, Com o Comando “^”.

```
tempr = wr*data|j^] - wi*data|j];
tempi = wr*data|j] + wi*data|j^];
```

Fonte: Autor, 2016.

A sequência de instruções, em Assembly, correspondente ao trecho de código apresentado na Tabela 16, incorporando o comando “^”, gerou um total de 28 instruções e pode ser conferido na Tabela 17.

Tabela 17 – Sequência de Instruções Para o Cálculo dos Componentes da Buterfly.

LOAD fftwr
PLDM1 fftj
PUSH
SRFR
LOAD data
SMLT
PLD fftwi
PLD fftj
PUSH
SRFR
LOAD data
SMLT
SSUB
SET ffttempr
LOAD fftwr
PLD fftj
PUSH
SRFR
LOAD data
SMLT
PLD fftwi
PLDM1 fftj
PUSH
SRFR
LOAD data
SMLT
SADD
SET ffttempi

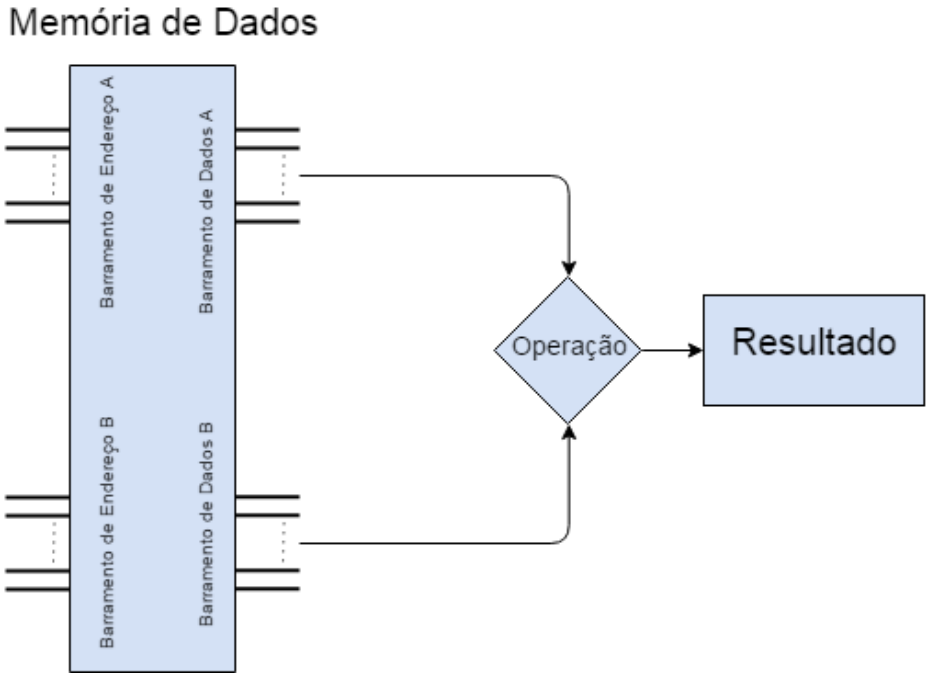
Fonte: Autor, 2016.

7.3 ACESSO DIRETO A WR E WI

O acesso a variável “wr” e “wi” pode ser otimizada, desde que a posição de memória ocupada por essas variáveis seja conhecida.

Uma vez que a posição ocupada pelas constantes “wr” e “wi” na memória de dados é conhecida não é mais necessário guardar os valores dessas variáveis no topo da pilha de dados, basta realizar as operações que envolvam “wr” e “wi”, presentes na Tabela 16, diretamente com os dados vindos da memória de dados, como mostra a Figura 16.

Figura 16 - Operação Com Acesso Direto à wr e wi.



Fonte: Autor, 2016.

A fim de tornar o acesso ao valor das variáveis “wr” e “wi” direto, foi definido que, para o código da FFT, a constante “wr” deve ocupar a primeira posição da memória de dados, enquanto “wi” deve ocupar a segunda posição da memória de dados.

O circuito de memória interno da FPGA apresenta barramento duplo de leitura e endereçamento de dados. Tendo em mãos esse fato foi definido que o barramento de dados B é exclusivo para o acesso aos valores representados pelas constantes “wr” e “wi”, à medida que o outro barramento será exclusivo para o acesso a outras constantes e variáveis presentes no código.

O controle de qual elemento será colocado no barramento de dados B, exclusivo para “wr” e “wi”, é feito por um multiplexador introduzido no circuito do processador.

A saída desse multiplexador é ligada diretamente no barramento de endereço B, exclusivo pra “wr” e “wi”, da memória de dados. O multiplexador é controlado pela saída da memória de instrução.

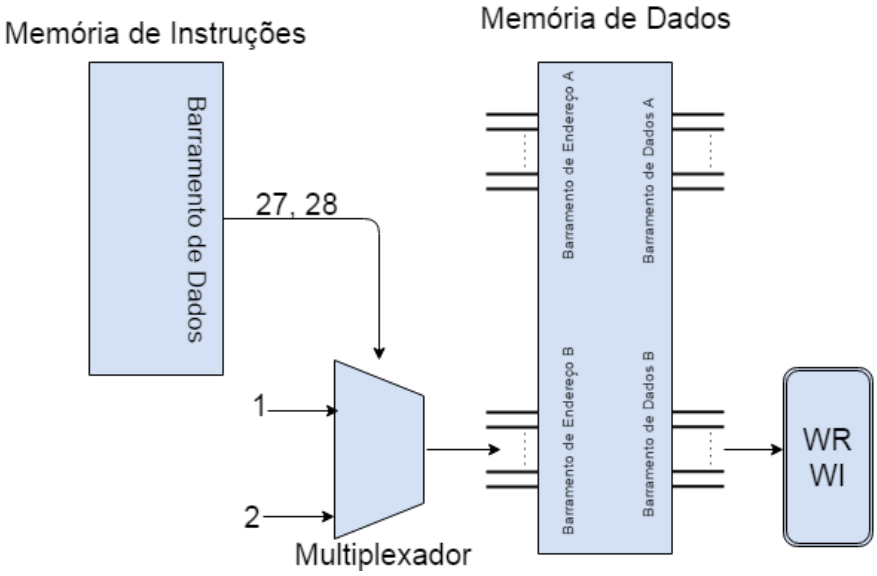
As instruções responsáveis por realizar as operações diretamente com “wr” e “wi” são: LWR e LWI, respectivamente. O símbolo que representa a instrução LWR é o “@” já o símbolo que representa a instrução LWI é o “$”.

Para eliminar todas as operações que necessitam utilizar a pilha de dados durante os cálculos presentes na Tabela 16 foram introduzidas mais duas operações no decodificador de instruções do hardware, justamente as instruções LWR que corresponde ao *opcode* 27, que realiza a multiplicação direta de “wr” pelo valor vindo do barramento de dados A e carrega o resultado dessa operação no acumulador, e a instrução LWI que corresponde ao *opcode* 28, responsável por realizar a multiplicação direta de “wi” pelo valor vindo do barramento de dados A, e ao final carrega o resultado dessa operação no acumulador.

Em resumo o que acontece é o seguinte: Se é requisitado ao processador realizar uma operação envolvendo a multiplicação de “wr” por algum valor ou variável, o *opcode* do decodificador de instruções será 27, sendo esse o sinal de controle do multiplexador introduzido entre as memórias de instrução e dados, consequentemente a saída do multiplexador será 1, já que foi requerido o acesso ao conteúdo da variável “wr”, que possui posição fixa - primeira posição - na memória de dados.

Se é requisitado ao processador realizar uma operação envolvendo a multiplicação de “wi” por algum valor ou variável, o *opcode* do decodificador de instruções será 28, sendo esse o sinal de controle do multiplexador introduzido entre as memórias de instrução e dados, consequentemente a saída do multiplexador será 2, já que foi solicitado o acesso à variável “wi”, que possui posição fixa - segunda posição - na memória de dados. A Figura 17 exemplifica todo o processo.

Figura 17 – Acesso Direto à wr e wi.



Fonte: Autor, 2016.

Além da modificação do decodificador de instruções e da introdução de um multiplexador, foi necessário alterar a estrutura da ULA do processador.

Uma das alterações na ULA foi a introdução de mais uma entrada. Até então a ULA possuía duas entradas, porém como agora existe uma maneira de acessar diretamente “wr” e “wi”, essas variáveis devem entrar diretamente na ULA vindas do barramento de dados B da memória de dados.

Entretanto foi preciso alterar a estrutura do multiplexador de operações aritméticas presentes na ULA, adicionando a ele uma saída a mais, a saída *lwri*, que leva o resultado da operação envolvendo “wr” e “wi” com outros dados à entrada do acumulador.

A Tabela 18 apresenta o total de instruções e elementos lógicos como resultado da introdução do multiplexador entre as memórias de instrução e dados, a modificação na ULA e os novos comandos, LWR e LWI.

Tabela 18 – Instruções x Elementos Lógicos.

Número de Instruções	Elementos Lógicos
393	635

Fonte: Autor, 2016.

Ao utilizar os comandos “@” e “$” o trecho da FFT correspondente à determinação dos componentes da *buterfly*, que é a parte que consome mais processamento, fica como apresentado na Tabela 19.

Tabela 19 – Cálculo dos Componentes da Buterfly, Com os Comandos “@” e “$”.

```
tempr = wr@data|j^] – wi$data|j];
tempi = wr@data|j] + wi$data|j^];
```

Fonte: Autor, 2016.

A sequência de instruções, em Assembly, correspondente ao trecho de código apresentado na Tabela 19, incorporando os comandos “@” e “$”, gerou um total de 20 instruções e pode ser conferido na Tabela 20.

Tabela 20 – Sequência de Instruções Para o Cálculo dos Componentes da Buterfly.

LOADM1 fftj
PUSH
SRFR
LWR data
PLD fftj
PUSH
SRFR
LWI data
SSUB
SET ffttempr
LOAD fftj
PUSH
SRFR
LWR data
PLDM1 fftj
PUSH
SRFR
LWI data
SADD
SET ffttempi

Fonte: Autor, 2016.

7.4 INDEXAÇÂO COM O ACUMULADOR

Após todo o processo de modificação do hardware percebeu-se que era possível realizar a indexação do vetor *data* sem a necessidade de utilizar a pilha de dados. O trecho de código apresentado na Tabela 21 mostra o processo de indexação do vetor *data* .

Tabela 21 – Indexação do Vetor Data.

LOADM1 fftj
PUSH
SRFR
LWR data

Fonte: Autor, 2016.

De acordo com a Tabela 21, para acessar uma posição do vetor *data*, primeiro é carregado no acumular o valor correspondente à posição que se deseja acessar, em seguida o valor é armazenado no topo da pilha de dados para que por fim, através do comando SRFR, seja possível acessar o conteúdo do vetor *data* da forma adequada.

Como a posição do vetor *data* que se deseja acessar já está presente no acumulador o processo de armazenamento na pilha de dados não é necessário.

Objetivando reduzir o número de instruções e aproveitar o fato de que o índice de acesso ao vetor *data* já está presente no acumulador uma nova instrução foi criada, a instrução SRFRA.

Em suma a instrução SRFRA, usa o valor presente no acumulador para realizar a indexação do vetor *data*.

Para que a instrução SRFRA tenha efeito no processador, foi preciso adicionar mais um *opcode* no decodificador de instruções, o *opcode*, 29. Outra modificação foi a introdução de mais uma saída no decodificador de instruções, a saída *st_ac*. Essa saída possui apenas um bit, valendo 1 apenas quando é necessário indexar vetores com dados vindos da pilha de dados.

Quando o *opcode* do decodificador de instruções é 29 acontecem às seguintes operações:

- A ULA não executa nenhuma instrução;

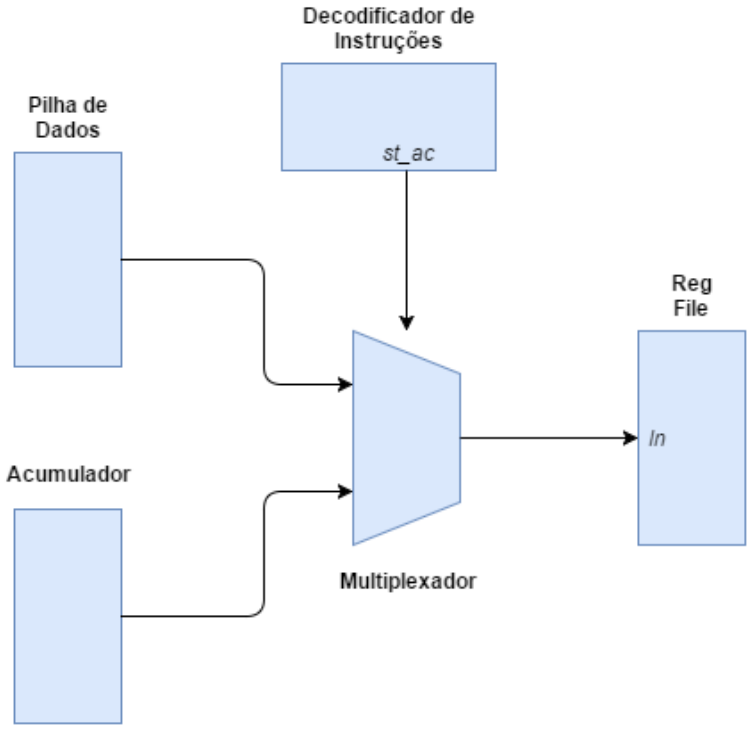
- O sinal *brev* vale 1;
- O sinal *srf* vale 1;
- O sinal *st_ac* vale 0;
- Não ocorre “pop” na pilha de dados.

Outra modificação necessária foi a inserção de um multiplexador no *core* do processador, com o objetivo de controlar qual índex será usado para acessar dados do vetor *data*.

A saída desse multiplexador é ligada na entrada “in” do *reg file*. O multiplexador é controlado pela saída *st_ac* do decodificador de instruções.

Os dados de entrada do multiplexador podem vir diretamente da pilha de dados ou do acumulador, de acordo com o sinal de controle. Se o sinal de controle, a saída *st_ac*, do decodificador de instruções vale 0 os dados que saem do multiplexador são os dados presentes no acumulador, ao passo que se *st_ac* for 1 os dados presentes na saída do multiplexador são os dados vindos da pilha de dados, como mostra a Figura 18.

Figura 18 – Multiplexação de Índex do Vetor Data.



Fonte: Autor, 2016.

A Tabela 22 apresenta o total de instruções e elementos lógicos como resultado da introdução da instrução SRFRA, que resultou na modificação do decodificador de instruções, e na introdução de um novo multiplexador no core do processador.

Tabela 22 – Instruções x Elementos Lógicos.

Número de Instruções	Elementos Lógicos
369	637

Fonte: Autor, 2016.

A sequência de instruções, em Assembly, resultante da implantação do comando SRFRA gerou um total de 16 instruções para o calculo dos componentes da *buterfly*, e pode ser conferido na Tabela 23.

Tabela 23 – Sequência de Instruções Para o Cálculo dos Componentes da Buterfly.

LOADM1 fftj
SRFRA
LWR data
PLD fftj
SRFRA
LWI data
SSUB
SET ffttempr
LOAD fftj
SRFRA
LWR data
PLDM1 fftj
SRFRA
LWI data
SADD
SET ffttempi

Fonte: Autor, 2016.

8 CONCLUSÃO

As modificações feitas na ULA, *core*, decodificador de instruções e na memória de dados do processador de ponto fixo desenvolvido na UFJF proporcionaram uma vultuosa redução na quantidade de instruções geradas pelo algoritmo da FFT implementada no compilador SAFO.

Em contra partida houve um aumento da quantidade de elementos lógicos consumidos pela FPGA, uma vez que a cada nova instrução criada, um novo circuito digital era incorporado ao processador.

É interessante que no futuro, a estrutura do processador possa ser modificada ainda mais para que ocorram, de fato, operações com números complexos, além disso, a remodelagem feita no processador de ponto fixo deve ser estendida ao processador de ponto flutuante e talvez seja válido descrever o mesmo processador

utilizando a arquitetura Von Neumann. Também, no futuro, é possível fazer uma comparação entre o número de *clocks* e a quantidade de elementos lógicos consumidos pelo processador modificado e a primeira versão.

O estudo da estrutura de um processador descrito usando a linguagem de descrição de hardware Verilog, para ser embarcado em uma FPGA é bastante empolgante e desafiador, uma vez que abre as portas para a implantação de diversas aplicações que necessitam de processamento online de alto desempenho.

O processador modificado pode ser usando na academia, servindo de ferramenta para simulações e até mesmo como objeto de estudo de alunos de graduação em Engenharia Elétrica.

Como o processador desenvolvido na UFJF é inovador, pode-se considerar a possibilidade de torná-lo comercial, tendo em vista que existe a necessidade de hardwares que apresentem alto desempenho de processamento de sinais.

REFERÊNCIAS

HART, Daniel W.**Eletrônica de Potência: análise e projetos de circuitos.** Porto Alegre: AMGH Editora Ltda., 2012.

PRES, William H. TEUKOLKY, Saul A.;VETTERLING, William T.;FLANNERY, Brian P. **Numerical Recipes in C. the art of scientifc computing.** New York: CAMBRIDGE UNIVERSITY PRESS. 1998.

Michelin, C. (1998).**Transformada de Fourier e Transformada de Fourier Rápida.** Ribeirão Preto. 125p. Dissertação (Mestrado) – Faculdade de Medicina de Ribeirão Preto, Universidade de São Paulo.

ALEXANDRINI, Fábio; BOECHEL, Tiago; SCHIQUETTI, Felipe Augusto; ROSA, Luís Ricardo; PROBST, Rodrigo Becker.**Desenvolvimento do Compilador da Linguagem básico.** Resende, 2014.

LATHI.B.P. **Lineares Systems and Signals.**Oxford University Press, 2004.

EMBREE, P.; DANIELI, D. **C++ Algorithms for Digital Signal Processing.**Prentice Hall PTR.SegundaEdição, 1998.

COLLEY, James W; TUKEY John W.**An Algorithm for the Machine Calculation of Complex Fourier Series.** MathComput, 1965.