ESTE EXEMPLAR CORRESPONDE A REDAÇÃO FINAL DA
TESE DEFENDIDA POR Danilo Morais
Pagano E APROVADA
PELA COMISSÃO JULGADORA EM 28/02/2012
ORIENTADOR

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA MECÂNICA
COMISSÃO DE PÓS-GRADUAÇÃO EM ENGENHARIA MECÂNICA

Danilo Morais Pagano

Proposta de uma arquitetura de processamento de sinais utilizando FPGA

Campinas, 2012

40/2012

Danilo Morais Pagano

Proposta de uma arquitetura de processamento de sinais utilizando FPGA

Dissertação apresentada ao Curso de Mestrado da Faculdade de Engenharia Mecânica da Universidade Estadual de Campinas, como requisito para a obtenção do título de Mestre em Engenharia Mecânica.

Área de Concentração: Mecânica dos Sólidos e Projetos Mecânicos

Orientador: Prof. Dr. Eurípedes Guilherme de Oliveira Nóbrega

Campinas
2012

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA E ARQUITETURA - BAE - UNICAMP

P14p

Pagano, Danilo Morais
Proposta de uma arquitetura de processamento de sinais utilizando FPGA / Danilo Morais Pagano – Campinas, SP: [s.n.], 2012.

Orientador: Eurípedes Guilherme de Oliveira Nóbrega.
Dissertação de Mestrado – Universidade Estadual de Campinas, Faculdade de Engenharia Mecânica.

1. Processamento de sinais - Tecnicas digitais.. 2. Fourier, Transformada de.. 3. FPGA. I. Nóbrega, Euripedes Guilherme de Oliveira. II. Universidade Estadual de Campinas. Faculdade de Engenharia Mecânica. III. Título.

Título em Inglês:	Proposal to an architecture for signal processing using FPGA
Palavras-Chave em Inglês:	Digital signal processing; Fourier Transform; FPGA.
Área de Concentração:	Mecânica dos Sólidos e Projetos Mecânicos
Titulação:	Mestre em Engenharia Mecânica
Banca Examinadora:	Peter Jürgen Tatsch e Luis Otávio Saraiva Ferreira.
Data da Defesa:	28/02/2012
Programa de Pós-Graduação:	Engenharia Mecânica

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA MECÂNICA
COMISSÃO DE PÓS-GRADUAÇÃO EM ENGENHARIA MECÂNICA
DEPARTAMENTO DE MECÂNICA COMPUTACIONAL

DISSERTAÇÃO DE MESTRADO

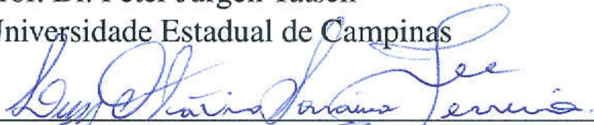
Proposta de uma arquitetura de processamento de sinais utilizando FPGA

Autor: Danilo Morais Pagano
Orientador: Prof. Dr. Eurípedes Guilherme de Oliveira Nóbrega

A Banca Examinadora composta pelos membros abaixo aprovou esta Dissertação:

Prof. Dr. Eurípedes Guilherme de Oliveira Nóbrega, Presidente
Universidade Estadual de Campinas

Prof. Dr. Peter Jürgen Tatsch
Universidade Estadual de Campinas



Prof. Dr. Luis Otávio Saraiva Ferreira
Universidade Estadual de Campinas

Campinas, 28 de fevereiro de 2012

Aos meus pais, minhas irmãs, minha namorada e meus amigos.

AGRADECIMENTOS

Agradeço inicialmente a Deus por ter me guiado durante esta fase de minha vida, oferecendo-me forças para seguir em frente sem desistir.

Aos meus pais, Donizeti dos Santos Pagano e Rosimeiry P. de Morais Pagano pelo apoio e incentivo, sempre me fortalecendo e amparando nos momentos difíceis.

Às minhas irmãs, Lívia e Letícia, que sempre estiveram ao meu lado e me tiveram como um exemplo a ser seguido.

À minha namorada, Daiana Bueno, pela compreensão, paciência e apoio, contribuindo enormemente para a realização deste trabalho.

Ao meu orientador, Prof. Dr. Eurípedes Guilherme de Oliveira Nóbrega, que me aceitou como aluno de mestrado, acreditou e confiou no meu trabalho, conduzindo-me pelos melhores caminhos a serem seguidos.

A todos os professores que participaram desta minha caminhada, sempre dispostos a ajudar, transferindo o conhecimento e auxiliando na minha formação.

A todos os meus amigos pelo incentivo, auxílio e companheirismo, principalmente aos colegas do departamento que também contribuíram com ideias, palpites e discussões.

À Biosensor Ind. e Com. LTDA pelo apoio financeiro e pela confiança depositada no meu trabalho.

A grandeza não consiste em receber honras, mas em merecê-las. Aristóteles

RESUMO

Esta dissertação apresenta um sistema para processamento digital de sinais através de dispositivos de *hardware* reconfigurável . Uma implementação do algoritmo FFT foi adotada como meio para avaliar o desempenho da arquitetura proposta para o sistema. O processamento digital de sinais tradicionalmente tem um alto custo computacional, pois os algoritmos são implementados em software, o que pode não atender as restrições de tempo de aplicações reais. O objetivo principal deste trabalho é desenvolver uma arquitetura para adquirir os sinais através de módulos de aquisição de dados distribuídos em uma rede e processá-los usando um FPGA. Um microcontrolador da FreeScale Semiconductors® foi adotado como módulo de aquisição de dados, executando um sistema operacional de tempo real (RTOS) para garantir os requisitos temporais. Foi implementado o processador *soft-core* NIOS 2 da Altera® executando também um RTOS com recursos de comunicação em rede, incluindo um periférico escrito em VHDL para o processamento da FFT usando uma estrutura de *pipeline* baseada em estágios e comunicação direta ao barramento do processador. A versão em *hardware* do algoritmo obteve uma redução de até 2000 vezes no tempo de processamento da FFT comparado com a mesma versão implementada em *software*, alcançando um tempo de processamento de 3.9 microssegundos para sinais discretizados em 256 pontos, quando usado 100MHz de *clock*. A quantidade de pontos pode ser facilmente aumentada alterando-se apenas o núcleo do periférico desenvolvido, e os resultados permitem adotar a arquitetura proposta para aplicações em tempo real de processamento digital de sinais.

Palavras-Chave: Processamento de sinais - Tecnicas digitais.; Fourier, Transformada de.; FPGA.

ABSTRACT

This work presents a digital signal processing system based on reconfigurable hardware. Implementation of the FFT algorithm is used as a mean to assess the adopted configuration performance. Digital signal processing algorithms are in general software implemented, incurring high computational cost, which may not attend the real-time constraints of real applications. The main objective of this work is to develop an FPGA based architecture to process signals acquired through a distributed network of data acquisition modules. A microcontroller from FreeScale Semiconductors® was adopted as data acquisition module, running a real-time operating system (RTOS) to guarantee timing requirements. The soft-core processor NIOS 2 from Altera®, also running an RTOS with network communication capabilities, was implemented including a peripheral module written in VHDL for the computation of the FFT, which uses a pipeline-based stage structure and directly communicates with the processor bus. The hardware version of the algorithm achieved a reduction up to 2000 times in the FFT processing time compared to the same version implemented in software, reaching a processing time of 3.9 microseconds for 256 points sampled signals when using 100MHz of clock. The number of points can be easily increased just changing the core of the developed peripheral module, and the results permit to expect adequate real-time application of digital signal processing adopting the proposed configuration.

Keywords: Digital signal processing; Fourier Transform; FPGA.

Lista de Figuras

1.1 Processador NIOS 2 no reconhecimento de folhas. 3

1.2 Processador NIOS 2 no reconhecimento de pessoas. 4

1.3 Projeto de *hardware* do filtro digital de imagens. 5

1.4 Projeto de *hardware* do leitor de cartões. 6

1.5 Diagrama de blocos do processador com instruções personalizadas. 7

1.6 Processador multi-núcleos desenvolvido. 7

1.7 Projeto de *hardware* do sistema de aquisição de dados. 8

1.8 Projeto de *hardware* simplificado do sistema de reconhecimento da fala. 9

1.9 Diagrama de blocos simplificado do processador. 10

2.1 Histórico do surgimento e utilização de tecnologias de construção de circuitos integrados. A parte branca indica que apesar da tecnologia estar disponível, ela ainda não tinha sido recebida com entusiasmo pelos profissionais do setor. 14

2.2 Estrutura interna de um FPGA. 17

2.3 Níveis de abstração em linguagens de descrição de *hardware*. 19

2.4 Exemplo de projeto em VHDL. 20

2.5 Estrutura interna do soft-core NIOS2. 24

2.6 Barramento Avalon *Memory Map* e seus módulos internos. 26

2.7 Comando de leitura e de escrita de dados em uma interface Avalon MM. 26

2.8 Transmissão de dados através da interface Avalon ST. 27

2.9 Kit de desenvolvimento NEEK. 28

2.10 Periféricos do kit de desenvolvimento NEEK. 29

2.11 Microcontrolador MCF51CN128. 30

2.12 Kit de desenvolvimento TOWER. 31

2.13 Conceitos básicos de um sistema operacional. 34

2.14 Componentes do núcleo e componentes adicionais do MQX. 36

2.15 Sinal representado no domínio do tempo e no domínio da frequência. 41

2.16 Operação borboleta. 44

2.17 Etapas do algoritmo FFT DIT. 45
2.18 Operação borboleta da dizimação em frequência. 47
2.19 Etapas do algoritmo FFT DIF para um sinal de 8 pontos. 47
2.20 Exemplo do algoritmo R2MDC com 8 pontos. 49

3.1 Funcionamento do sistema: da aquisição à visualização. 52
3.2 Estrutura do sistema de aquisição em rede. 52
3.3 Diagrama de fluxo de dados do sistema. 53
3.4 Diagrama de partição do módulo de aquisição de dados. 54
3.5 Diagrama de interação do módulo de aquisição de dados. 55
3.6 Diagrama de estados SDL da tarefa de controle do conversor AD. 56
3.7 Formato do comando enviado do NEEK para o TOWER. 57
3.8 Formato da resposta enviada do TOWER ao NEEK. 57
3.9 Diagrama de estados SDL da tarefa de monitoramento da rede e diagrama de estados simplificados. 58
3.10 Diagrama de sequência do módulo de aquisição de dados. 59
3.11 Placa do NEEK com o processador desenvolvido e os periféricos externos ao FPGA de forma simplificada. 60
3.12 Organização do *software* desenvolvido. 61
3.13 Janela do SoPC Builder com o processador construído. 62
3.14 Diagrama de blocos do processador desenvolvido. 63
3.15 Tarefas do módulo de comunicação 66
3.16 Estrutura de dados da tela e de um botão. 67
3.17 Diagrama do fluxo de dados no módulo em desenvolvimento. 70
3.18 Fluxo de dados em um pipeline de 8 estágios. 70
3.19 Entradas e saídas de um estágio. 71
3.20 Detalhamento da estrutura interna de um estágio. 71
3.21 Operação borboleta no algoritmo DIF. 73
3.22 Sinais de entrada e saída do pipeline. 73
3.23 Controlador para o pipeline FFT. 74
3.24 Controlador para o pipeline FFT. 77
3.25 Tela de configuração dos sinais do periférico desenvolvido. 79

4.1 Diagrama de fluxo de dados do teste realizado na primeira situação. 82
4.2 Sinal original de 4Hz, FFT calculada pelo FPGA e FFT calculada pelo Matlab®. . . . 83
4.3 Sinal original de 32Hz, FFT calculada pelo FPGA e FFT calculada pelo Matlab®. . . . 83
4.4 Sinal original de 64Hz, FFT calculada pelo FPGA e FFT calculada pelo Matlab®. . . . 84
4.5 Diagrama de fluxo de dados do teste realizado na segunda situação. 85
4.6 Sinal original em forma de senoide, FFT calculada pelo FPGA e FFT calculada pelo Matlab®. 85
4.7 Sinal original de onda quadrada de 10Khz, FFT calculada pelo FPGA e FFT calculada pelo Matlab®. 86
4.8 Sinal original de senoide de 10KHz, FFT calculada pelo FPGA e FFT calculada pelo Matlab®. 86
4.9 Sinal original de senoide de 20KHz, FFT calculada pelo FPGA e FFT calculada pelo Matlab®. 87
4.10 Diagrama de fluxo de dados do teste realizado na terceira situação. 87
4.11 À esquerda, tela da área de trabalho com o atalho para as demais telas. À direita, a tela do aplicativo de aquisição de dados. 88
4.12 Diferentes espectros de frequências para diferentes formas de onda, ambas com frequência de 1KHz. 88
4.13 Foto do NEEK com sinal senoidal de 5KHz e de 10KHz. 89

A.1 Configuração genérica do *pipeline*. 103
A.2 *Pipeline* com todos os sinais envolvidos. 104
A.3 Estrutura de uma fila de dados. 105
A.4 Estrutura do multiplexador auto-controlado. 105
A.5 Operação borboleta DIF. 105
A.6 Entradas e saídas do gerador de fatores. 106
A.7 Elementos de um estágio. 106
A.8 Sequência simplificada do *pipeline*. 107
A.9 Sequência detalhada do *pipeline*. 108
A.10 Simplificação para facilitar a simulação. 108
A.11 Passos 0 a 4 a simulação. 109
A.12 Passo 5 da simulação. 110

A.13 Passo 6 da simulação. 110
A.14 Passo 7 da simulação. 110
A.15 Passo 8 da simulação. 111
A.16 Passo 9 da simulação. 111
A.17 Passo 10 da simulação. 112
A.18 Passo 11 da simulação. 112
A.19 Passo 12 da simulação. 112
A.20 Passo 13 da simulação. 113
A.21 Relação dos passos com a execução do algoritmo DIF. 113

Lista de Tabelas

2.1 Principais diferenças entre as versões do processador NIOS 2. 23
2.2 Tabela de ordenação *bit*-reverso. 45

4.1 Tempos médios do processamento a 100MHz. 89
4.2 Utilização dos elementos lógicos do FPGA. 90
4.3 Utilização de elementos lógicos pelos principais periféricos do processador. 90

D.1 Comparação dos resultados para executar 1 multiplicação, em milissegundos e por pulsos de *clock*. 129
D.2 Tempos de execução do algoritmo FFT para 256 pontos em ponto flutuante 129

Lista de Abreviaturas e Siglas

Abreviaturas

art.	artigo
cap.	capítulo
comb.	combinacional
mux.	multiplexador
reg.	registrador
sec.	seção

Siglas

ADC	Conversor Analógico-Digital
API	*Application Programming Interface*
ASIC	*Application Specific Integrated Circuit*
CI	Circuito Integrado
CPLD	*Complex Programmable Logic Device*
DFT	*Discrete Fourier Transform*
DIF	*Decimation In Frequency*
DIT	*Decimation in Time*
DMA	*Direct Memory Access*
DMC	Departamento de Mecânica Computacional
DSP	*Digital Signal Processing*
DRAM	*Dynamic Random Access Memory*
EL	Elementos lógicos
FEM	Faculdade de Engenharia Mecânica
FFT	*Fast Fourier Transform*
FIFO	*First In First Out*
FPGA	*Field Programmable Gate Array*

Gbps	Giga *bits* por segundo
HAL	*Hardware Abstraction Layer*
LCD	*Liquid Crystal Display*
LE	*Logic Elements*
LUT	*Look Up Table*
MAC	*Midia Access Controller*
MIPS	Milhões de Instruções Por Segundo
MM	*Memory Map*
NEEK	NIOS 2 *Embedded Evaluation Kit*
PLD	*Programmable Logic Device*
PLL	*Phased Locked Loop*
R2^2SDC	*Radix*2^2 *single-path delay commutato*r
R2MDC	*Radix-2 multipath delay commutator*
R2SDC	*Radix-2 single-path delay feedback*
R4MDC	*Radix-4 multipath delay commutator*
R4SDC	*Radix-4 single-path delay commutator*
R4SDF	*Radix-4 single-path delay feedback*
RAM	*Random Access Memory*
RISC	*Reduced Instruction Set Computer*
RTCS	*Real Time Communication Suite*
RTOS	*Real-Time Operating System*
RTL	*Register Transfer Level*
SDL	*Specification and Description Language*
SDRAM	*Synchronous Dynamic Random Access Memory*
SGDMA	*Scatter-Gather Direct Memory Access*
SoPC	*System On a Programmable Chip*
SPLD	*Simple Programmable Logic Device*
SRAM	*Static Random Access Memory*

SSRAM	*Synchronous Static Random Access Memory*
ST	*Stream*
TCP	*Transmission Control Protocol*
TOWER	TWR-MCF51CN128
TSE	*Triple Speed Ethernet*
UART	*Universal asynchronous receiver/transmitter*
UDP	*User Datagram Protocol*
ULA	Unidade Lógica Aritmética
UNICAMP	Universidade Estadual de Campinas
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Intergraded Circuit*

SUMÁRIO

1 INTRODUÇÃO **1**

1.1 Trabalhos relacionados 2

1.2 Objetivo 9

1.3 Organização do trabalho 10

2 FUNDAMENTOS **13**

2.1 Introdução ao FPGA 13

2.1.1 Linguagens para programação de FPGAs 18

2.1.2 Fluxo de projeto para FPGA 21

2.2 Microprocessadores em *hardware* reconfigurável 22

2.2.1 Processador *Soft-core* NIOS 2 e SoPC *Builder* 23

2.2.2 Kits de desenvolvimento 28

2.3 Microcontrolador *Coldfire* FreeScale® 29

2.3.1 O microcontrolador MCF51CN128 29

2.3.2 Kits de desenvolvimento 30

2.4 Sistemas operacionais de tempo real 31

2.4.1 Características principais 32

2.4.2 RTOS uC/OS2 34

2.4.3 RTOS MQX 35

2.5 Representação numérica de ponto fixo 36

2.5.1 Números racionais positivos 37

2.5.2 Complemento de 2 em números binários 38

2.5.3 Números racionais sinalizados 38

2.5.4 Regras básicas para aritmética em ponto fixo 38

2.6 Processamento espectral de sinais 40

2.6.1 Transformada Fourier Discreta 41

2.6.2 Transformada Fourier Rápida 42

2.6.3 Dizimação no tempo 43

2.6.4 Dizimação na frequência . . . 46
2.6.5 Algoritmos seriais para a FFT . . . 48

3 DESENVOLVIMENTO DA ARQUITETURA . . . 51
3.1 Visão geral do sistema . . . 51
3.1.1 Aquisição de dados . . . 54
3.1.2 Processamento de sinais usando FPGA . . . 59
3.2 Algoritmo FFT em FPGA . . . 68
3.2.1 O fluxo de dados . . . 69
3.2.2 O controlador do fluxo de dados . . . 74
3.2.3 Implementação do módulo personalizado para o cálculo da FFT . . . 76
3.2.4 Integração do módulo implementado com o NIOS 2 . . . 79

4 RESULTADOS . . . 81
4.1 Testes realizados e seus resultados . . . 81
4.1.1 Situação 1 . . . 81
4.1.2 Situação 2 . . . 84
4.1.3 Situação 3 . . . 87
4.2 Tempo de execução . . . 89
4.3 Utilização de elementos lógicos . . . 90

5 CONCLUSÕES E PERSPECTIVAS . . . 91
5.1 Perspectivas . . . 93

REFERÊNCIAS . . . 95

APÊNDICE A - FUNCIONAMENTO DO ALGORITMO FFT . . . 103
A.1 Simulação para oito pontos . . . 107

APÊNDICE B - CÓDIGOS VHDL . . . 115
B.1 Tipos . . . 115
B.2 Comutador . . . 115
B.3 Registrador . . . 116
B.4 FIFO . . . 117

B.5 *Butterfly* DIF . . . 118

B.6 Estágio . . . 119

B.7 *Pipeline* . . . 120

APÊNDICE C - ESTRUTURA DE DADOS DO MÓDULO DE GERENCIAMENTO DE TELAS 123

APÊNDICE D - DESEMPENHO DE UMA MULTIPLICAÇÃO NO PROCESSADOR NIOS 2 E SEU EFEITO NA FFT EM SOFTWARE . . . 129

1 INTRODUÇÃO

Nas últimas décadas, as constantes mudanças e avanços na tecnologia têm transformado a forma de projetar circuitos digitais. A presença dos *chips* reconfiguráveis faz com que algoritmos complexos possam ser embarcados e que a prototipagem aconteça em tempo reduzido, além de poder ter o seu comportamento alterado rapidamente. A existência atual de *hardware* reprogramável de grande capacidade permite que algoritmos de alta complexidade computacional possam ser implementados de forma concorrente, em oposição à regra de uma instrução por vez por processador, conhecida como "gargalo" de Von Neumann (SEBESTA, 2002).

A proposta deste trabalho é atuar no processamento de sinais, desenvolvendo uma arquitetura que possa ser utilizada com facilidade para a implementação concorrente dos respectivos algoritmos. Sinais estão presentes em muitas situações do dia-a-dia do ser humano. De acordo com Mello (2011), um sinal pode ser definido como uma função que carrega uma informação. Um exemplo é a comunicação por sinal de voz, onde há o sinal gerado pelo trato vocal e o sinal recebido pelo sistema auditivo. Apesar do sinal ser transmitido, a forma como ele é processado é inerente ao receptor. O processamento de sinais lida com a representação, transformação e manipulação dos sinais e da informação que eles contêm. Até a década de 60, a tecnologia para processamento de sinais era basicamente analógica. A evolução de computadores e microprocessadores juntamente com diversos desenvolvimentos teóricos causou um grande crescimento na tecnologia digital, surgindo o processamento digital de sinais (PDS). Um aspecto fundamental do processamento digital de sinais é que ele é baseado no processamento de sequências de amostras. Para tanto, o sinal contínuo no tempo é convertido numa sequência de amostras, ou seja, convertido em um sinal discreto no tempo. Após o processamento digital, a sequência de saída pode ser convertida de volta a um sinal contínuo no tempo.

Para realizar o processamento de um sinal podem ser usadas diversas ferramentas: um computador pessoal, um microcontrolador, um FPGA (*Field Programmable Gate Array*), dentre outros. A proposta deste trabalho é utilizar um FPGA para processar dados. Foi desenvolvido um sistema de aquisição de dados distribuído em uma rede local, onde diversos dispositivos realizam a aquisição de um sinal e o transmitem pela rede para um dispositivo central (um FPGA) que o processa e exibe o resultado numa tela de LCD (*Liquid Crystal Display*).

1.1 Trabalhos relacionados

Existem diversos trabalhos que utilizam processadores *soft-core* embarcados em FPGA combinados com módulos personalizados para otimizar o processamento. Esses projetos geralmente buscam a execução em tempo real, ou seja, dentro de um tempo determinado. Para tanto, são propostas arquiteturas que permitem que o processador fique livre do processamento computacional mais complexo, através da criação de rotinas concorrentes diretamente em *hardware*. Estas rotinas geralmente se transformam ou em instruções personalizadas diretamente no núcleo do processador ou em periféricos acoplados ao barramento interno do processador. As duas implementações otimizam o processo e aumentam o desempenho do sistema. Ao utilizar instruções personalizadas, é necessário escrever um programa em linguagem C e ocupar o processador com esta tarefa. Utilizar periféricos é uma solução que pode desocupar consideravelmente o processador, deixando-o livre para executar outras funções, permitindo otimizações mais eficazes diretamente em *hardware*.

As implementações utilizando esta metodologia de projeto, que inclui projeto de *hardware* em FPGA e utilizam processadores *soft-core* executando um determinado algoritmo, têm levado a resultados de alto desempenho em termos de tempo de execução. A flexibilidade presente nesta metodologia permite criar aplicações para diversas áreas do conhecimento, como por exemplo, controle robótico, processamento de sinais, processamento de imagens e inteligência artificial. Geralmente implementações em *hardware* são melhores que implementações em *software* em termos de tempo de processamento e consumo de energia. Alguns trabalhos relacionados com esta metodologia serão apresentados nos próximos parágrafos.

O processamento de imagens digitais é um assunto que se estende à ciência da computação, teoria da informação, geometria e outras aplicações. Liao et al. (2010) propõem um sistema de reconhecimento de folhas de plantas utilizando um projeto misto de *software* e *hardware*, onde utiliza instruções personalizadas diretamente no núcleo do processador para acelerar o tempo de execução. Neste trabalho foi adicionada uma instrução de comparação personalizada para realizar a identificação das folhas, resultando em redução no tempo de processamento em até 7 vezes. A Figura 1.1 apresenta um diagrama de blocos simplificado do processador.

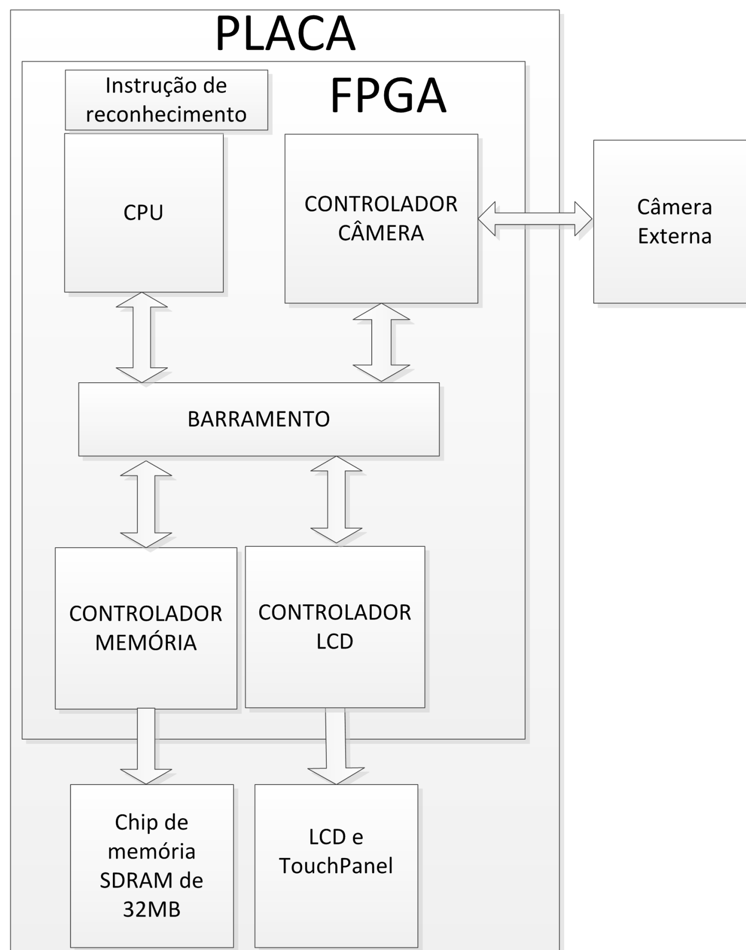


Figura 1.1 - Processador NIOS 2 no reconhecimento de folhas.
Fonte: Adaptado de Liao et al. (2010).

Reconhecimento e rastreamento de pessoas em imagens é uma tarefa complexa. Um método de reconhecimento foi proposto por Mayya et al. (2010), que utiliza um processador NIOS 2 e instruções personalizadas para acelerar o reconhecimento das pessoas. O algoritmo foi desenvolvido em linguagem C, e foi executado pelo processador que utilizou variáveis de ponto fixo em *software*. Obteve-se uma taxa de quatro quadros por segundo. Ao invés de utilizar uma câmera para capturar as imagens, foi utilizada uma memória *Flash* com as imagens e vídeos previamente armazenados. A Figura 1.2 apresenta o processador desenvolvido.

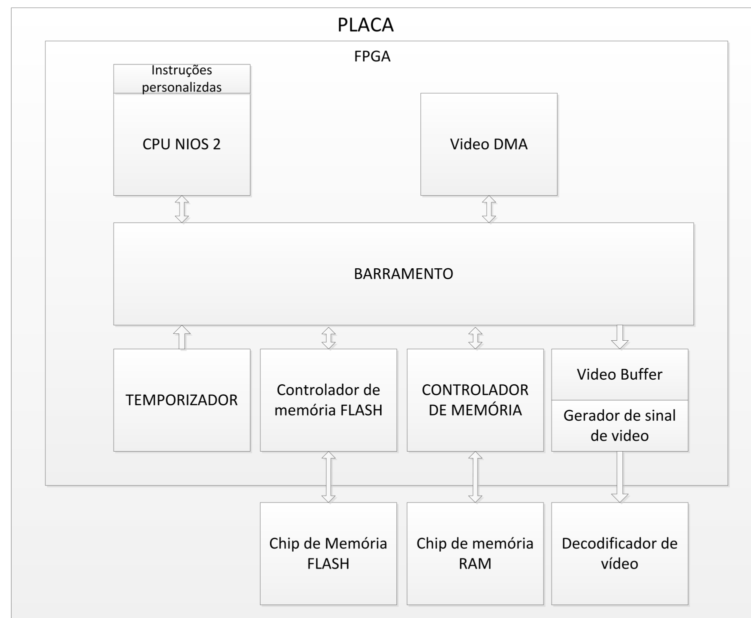


Figura 1.2 - Processador NIOS 2 no reconhecimento de pessoas.
Fonte: Adaptado de Mayya et al. (2010).

A execução de filtros digitais de sinais é uma tarefa complexa e pode consumir um alto tempo na execução. No trabalho de Boudabous et al. (2005), o foco está na aceleração do algoritmo do filtro digital de distância direcional aplicado na remoção de ruídos em imagens. Foi realizada a comparação entre a execução do algoritmo puramente em *software* e uma segunda versão composta de um programa para movimentar os dados e instruções personalizadas para acelerar a execução do algoritmo, utilizando o processador NIOS 2 da Altera®. Obteve-se uma redução de aproximadamente 20 mil vezes no tempo de execução e todos os 288 multiplicadores em *hardware* presentes no FPGA foram utilizados.

Uma implementação parecida com a anterior, proposta pelo mesmo autor, é o filtro digital por vetor de medianas, apresentada em Boudabous et al. (2007). A contribuição deste trabalho está na implementação do algoritmo em um módulo periférico ao processador, utilizado para realizar os cálculos. O módulo proposto foi criado utilizando uma linguagem de descrição de circuitos. Nele foram instanciados 36 vezes o componente de cálculo da norma entre dois pixels, um componente que realiza o acúmulo destes resultados e por fim um componente que minimiza a norma calculada. Esta complexidade computacional foi removida do processador, que agora deve enviar os dados para o periférico e realizar a leitura do resultado. Com isto, 96% dos multiplicadores presentes no FPGA foram utilizados, gerando uma redução de aproximadamente 48 vezes na execução do

algoritmo. A Figura 1.3 apresenta o processador desenvolvido.

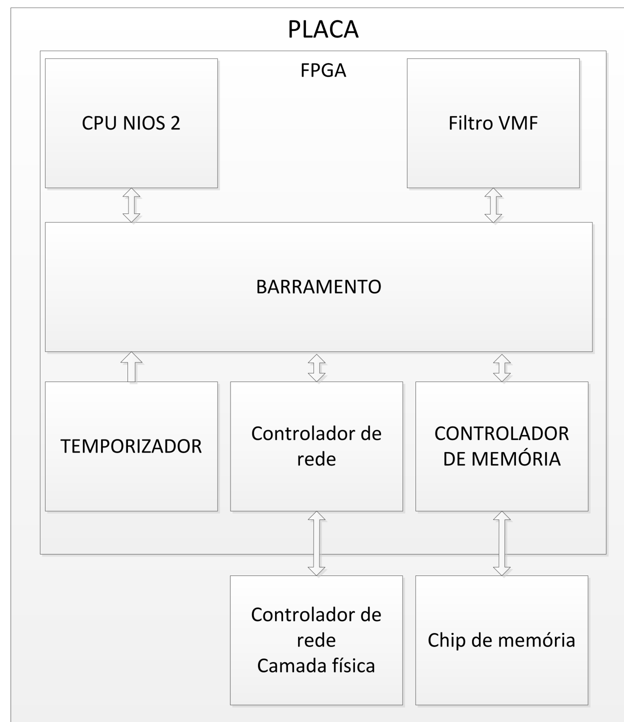


Figura 1.3 - Projeto de *hardware* do filtro digital de imagens.
Fonte: Adaptado de Boudabous et al. (2007).

Além de processamento de imagens, outras aplicações para um sistema composto por um FPGA foram avaliadas. Todas elas possuem um processador e implementações em *hardware* para realizar otimizações de processamento. Yang et al. (2011) apresentam uma implementação de um leitor de imagens de um cartão de memória com saída para um sinal de vídeo. Trata-se de um projeto em que o foco está na leitura dos dados do cartão implementado em linguagem C. Porém foi preciso desenvolver o *hardware* no FPGA que faz o acesso e o processamento dos dados contidos no cartão. A Figura 1.4 apresenta o processador desenvolvido.

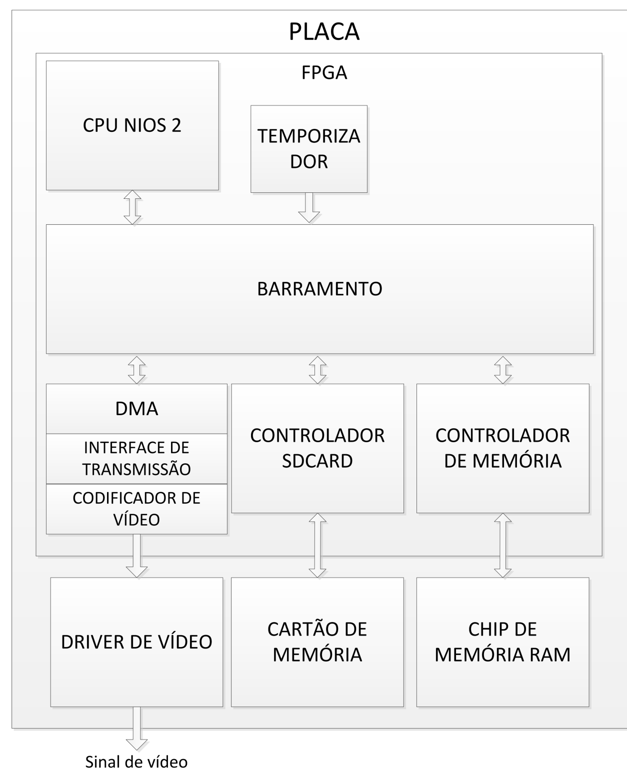


Figura 1.4 - Projeto de *hardware* do leitor de cartões.
Fonte: Adaptado de Yang et al. (2011).

Ramakrishna et al. (2011) sugerem a utilização de instruções personalizadas para realizar o cálculo do histograma e da saturação de uma imagem. É feita uma comparação do algoritmo sem e com o uso de instruções personalizadas no processador NIOS 2. Observa-se um ganho no tempo de execução ao utilizar instruções personalizadas. A Figura 1.5 apresenta a estrutura interna do processador desenvolvido.

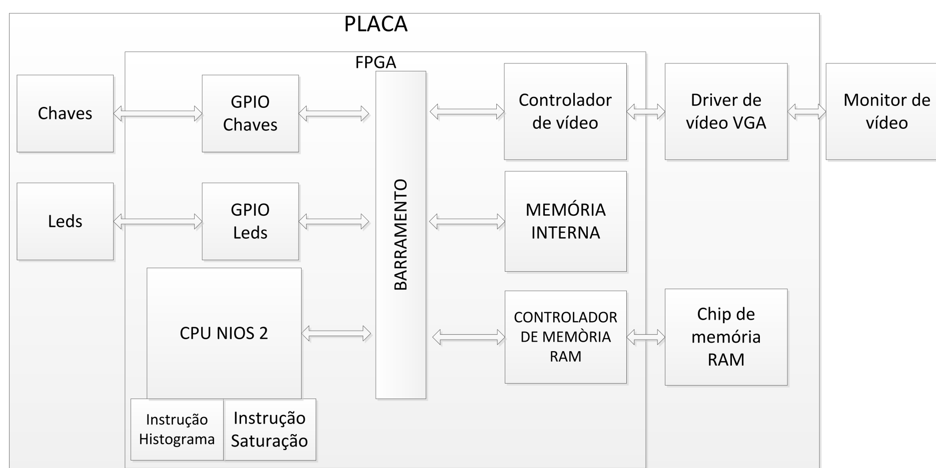


Figura 1.5 - Diagrama de blocos do processador com instruções personalizadas.
Fonte: Adaptado de Ramakrishna et al. (2011).

A Lógica nebulosa tem sido muito utilizada para resolver problemas de controle. A implementação de lógica nebulosa em FPGA tem a vantagem de ser flexível e de execução rápida. Kung et al. (2009) implementam um processador com quatro núcleos NIOS 2 para controlar uma mesa XYZ (3 graus de liberdade). Um dos núcleos calcula a trajetória do movimento, enquanto os demais controlam os motores. Cada núcleo de controle dos motores recebe o comando de posicionamento do núcleo principal e realiza o controle da posição utilizando lógica nebulosa. Portanto, o funcionamento paralelo dos quatro núcleos aumentam o poder computacional do sistema. A Figura 1.6 mostra o processador de forma simplificada.

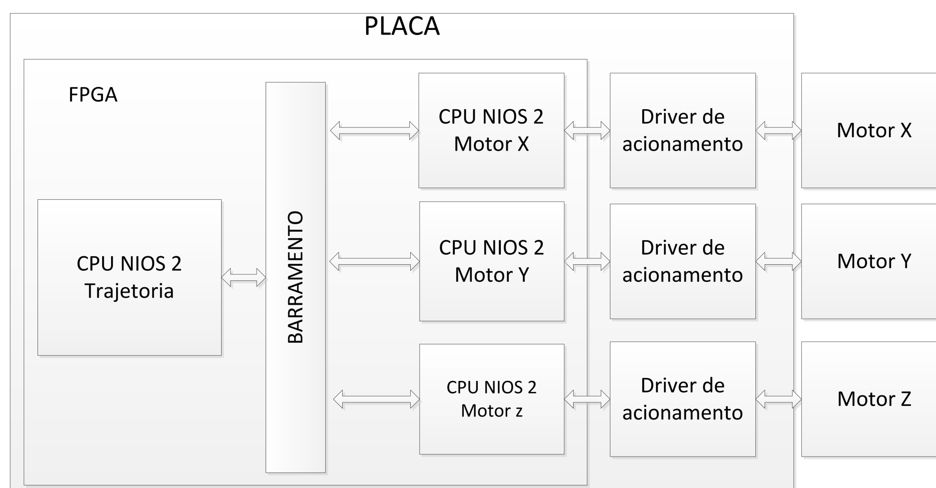


Figura 1.6 - Processador multi-núcleos desenvolvido.
Fonte: Adaptado de Kung et al. (2009).

Sistemas de aquisição de dados utilizam placas de aquisição de dados ou microprocessadores para controlar a aquisição, armazenamento e transmissão dos dados. Wei e Guidong (2010) apresentam alguns pontos fracos destes sistemas e propõem o uso de um FPGA para realizar em *hardware* o controle da aquisição e o uso do processador NIOS 2 para realizar o ajuste do sinal antes de enviá-lo para um computador através da porta USB. O projeto é flexível, com ênfase na portabilidade e facilidade de configuração do conversor analógico-digital externo ao FPGA. A Figura 1.7 mostra este processador.

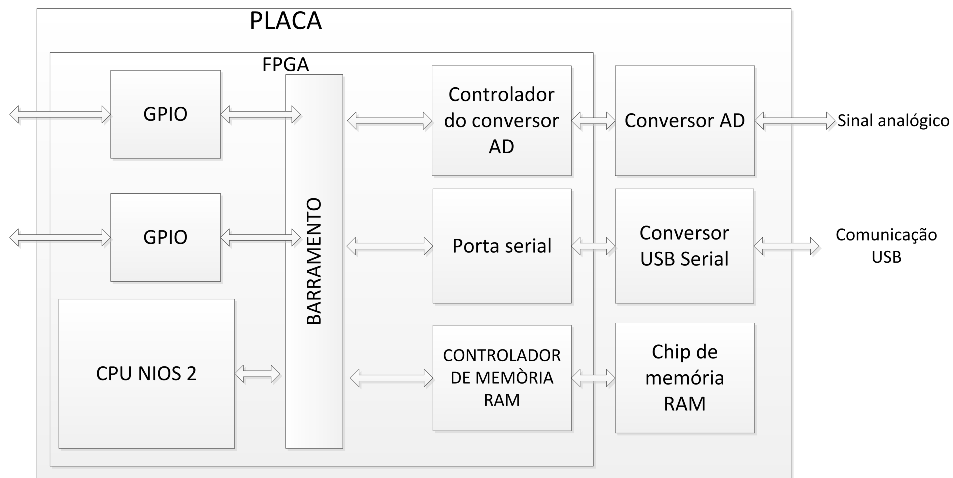


Figura 1.7 - Projeto de *hardware* do sistema de aquisição de dados.
Fonte: Adaptado de Wei e Guidong (2010).

O reconhecimento da fala pode ser utilizado em inúmeras aplicações. Reconhecer uma palavra envolve uma sequência de processamentos, que podem ser otimizados através do uso de um FPGA. Zhang et al. (2011) apresentam uma proposta de sistema embarcado em um FPGA para reconhecimento da fala. O projeto foi dividido em 2 partes: uma utiliza o *hardware* para aplicar o algoritmo da transformada rápida de Fourier externo a um processador NIOS 2 e a outra aplica os demais processos matemáticos para realizar o reconhecimento. Obteve-se um ganho de 17 vezes no tempo de reconhecimento. A Figura 1.8 apresenta o diagrama de blocos simplificado do sistema.

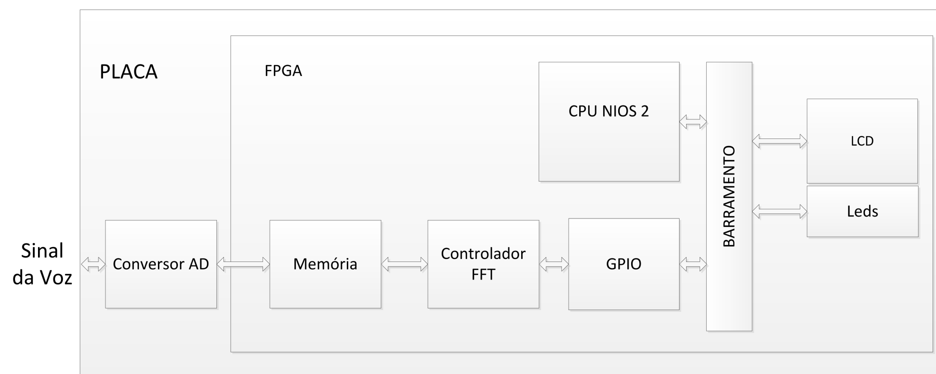


Figura 1.8 - Projeto de *hardware* simplificado do sistema de reconhecimento da fala.
Fonte: Adaptado de Zhang et al. (2011).

1.2 Objetivo

O principal objetivo deste trabalho foi desenvolver uma arquitetura para adquirir e processar sinais digitais usando um FPGA como módulo central, e módulos de conversão analógico-digital distribuídos em uma rede local. Os resultados obtidos no processamento poderão ser transmitidos para um sistema de arquivamento e monitoramento remoto por operadores, além de possibilitar a operação do sistema de processamento diretamente sobre o FPGA. Neste caso, existe a necessidade de uma interface homem-máquina para apresentação dos dados processados e para controle da aquisição remota dos sinais. Deste modo, a unidade central representada pelo FPGA necessitou de inteligência embarcada, por isso foi utilizado um microcontrolador de boa capacidade de processamento para as funções de comunicação. Deve-se ressaltar que este microcontrolador foi projetado em linguagem de descrição de circuitos e sintetizado no FPGA. Porém, como não é o foco do sistema, adotou-se a princípio uma propriedade intelectual do fabricante do FPGA.

Deste modo, pode-se dividir o trabalho realizado em três partes principais:

- módulo de aquisição de dados;
- módulo de processamento digital de sinais;
- interface homem-máquina.

Para tanto, foi necessário projetar e implementar uma arquitetura no processador que permita executar todas as funções necessárias. O diagrama de blocos da Figura 3.11 apresenta um esquema

simplificado, onde pode-se visualizar o processador que foi projetado.

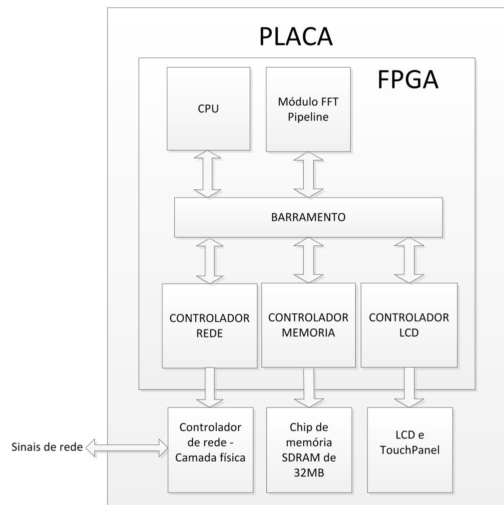


Figura 1.9 - Diagrama de blocos simplificado do processador.

Com o propósito de concentrar os esforços e prover uma metodologia de teste do sistema a desenvolvido foi adotada uma ferramenta de processamento digital de sinais que possa explorar os recursos disponíveis no sistema, a transformada rápida de Fourier.

1.3 Organização do trabalho

O capítulo 1 apresenta uma introdução ao trabalho realizado, onde comenta-se a respeito de trabalhos relacionados que utilizam processadores *soft-core* e módulos em *hardware* para otimizar o processamento em FPGA. Estes trabalhos vão de sistemas de aquisição de dados a processamento de imagens.

O capítulo 2 apresenta os fundamentos utilizados no trabalho. São apresentados os dispositivos programáveis, sua composição e utilização; o microprocessador *soft-core* NIOS 2 e seu barramento; os sistemas operacionais de tempo real; a representação numérica de ponto fixo e sua aritmética; e os algoritmos da transformada rápida de Fourier.

O capítulo 3 aborda o desenvolvimento do projeto. Primeiro é apresentada uma visão geral da arquitetura. Depois é apresentado o módulo de aquisição de dados detalhadamente. Finalmente é apresentado o algoritmo implementado em *hardware* para transformada de Fourier e sua integração

ao processador NIOS 2.

O capitulo 4 expõe os resultados obtidos nos experimentos comparando a implementação dos algoritmos puramente em *software* e utilizando o módulo FFT desenvolvido em *hardware*. É levantada a discussão sobre a utilização de elementos lógicos no FPGA.

O capítulo 5 apresenta as conclusões obtidas pelos experimentos e aponta futuros trabalhos a serem desenvolvidos.

O apêndice A apresenta uma simulação passo a passo do algoritmo proposto para o cálculo da FFT.

2 FUNDAMENTOS

Este capítulo aborda os conhecimentos necessários para a realização do projeto proposto. Na primeira seção será abordado o funcionamento dos FPGAs, as linguagens de descrição de *hardware* existentes, o fluxo de um projeto em FPGA, o uso de processadores *soft-core* e a ferramenta de desenvolvimento utilizada. Na segunda seção será apresentado o microcontrolador MCF51CN128 da FreeScale Semiconductors que foi utilizado neste projeto, abordando suas características técnicas, seus periféricos e o uso do sistema operacional de tempo real MQX, além da ferramenta de desenvolvimento. Na terceira seção será apresentada a representação numérica em ponto fixo, utilizada nos cálculos implementados no FPGA. Por fim, será apresentada a transformada de Fourier discreta e suas variações de implementação através da transformada rápida com dizimação no tempo e na frequência.

2.1 Introdução ao FPGA

Antes da criação dos FPGAs, as tecnologias disponíveis para desenvolvimento de circuitos eram os transistores, circuitos integrados (memórias, registradores, e outros), microprocessadores, SPLDs (*Simple Programmable Logic Device*), CPLDs (*Complex Programmable Logic Device*) e ASICs (*Application Specific Integrated Circuits*). A Figura 2.1 mostra as datas aproximadas do surgimento destas tecnologias. Por exemplo, embora o primeiro FPGA tenha surgido por volta de 1984, os projetistas de *hardwares* só começaram a utilizar esta tecnologia no início de 1990 (MAXFIELD, 2004).

Após a invenção dos transistores, os circuitos digitais tornaram-se cada vez mais complexos. O número de transistores utilizados nos circuitos digitais aumentava rapidamente. Portanto, surgiu a necessidade de reduzir a eletrônica envolvida e desta forma surgiu o circuito integrado (CI). A cada novo CI lançado, a complexidade dos projetos aumentava exponencialmente, possibilitando o desenvolvimento de aplicações que até então eram difíceis de serem implementadas (MAXFIELD, 2004).

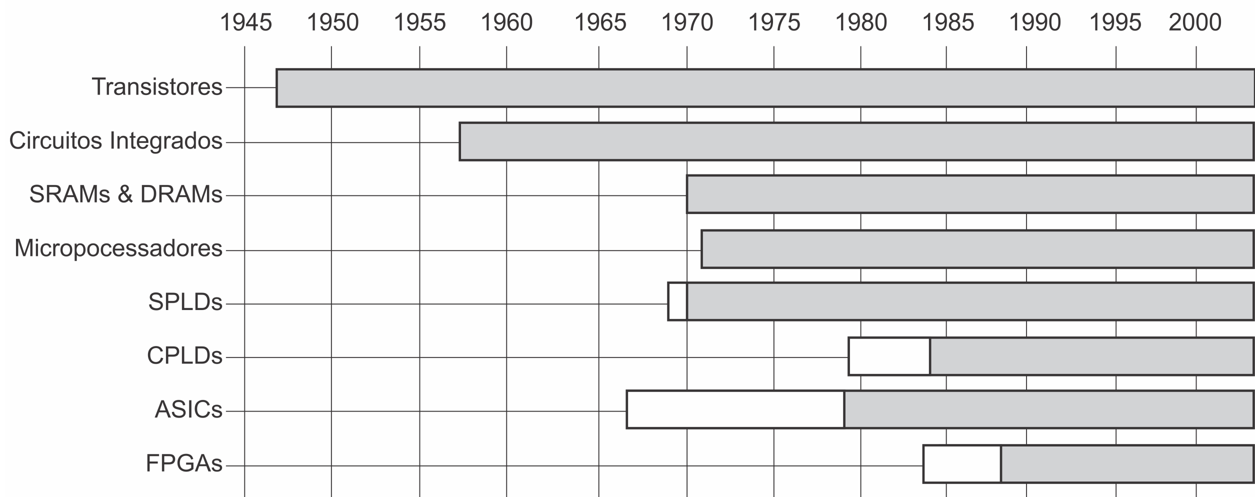


Figura 2.1 - Histórico do surgimento e utilização de tecnologias de construção de circuitos integrados. A parte branca indica que apesar da tecnologia estar disponível, ela ainda não tinha sido recebida com entusiasmo pelos profissionais do setor.
Fonte: (MAXFIELD, 2004)

No final de 1960 e início de 1970 surgiram novas tecnologias na área de circuitos integrados digitais, o que impulsionou o seu processo de expansão. Em 1970 foi anunciada a primeira DRAM (*Dynamic Random Access Memory*). A memória RAM dinâmica é formada por pares de transistores e capacitores que consomem pouco espaço no silício. O atributo dinâmico é utilizado porque o capacitor perde a informação ao longo do tempo, de modo que cada célula da DRAM deve ser periodicamente atualizada para manter a informação armazenada. A memória SRAM (*Static Random Access Memory*) difere da memória DRAM porque uma vez que a informação foi carregada para dentro da célula SRAM os dados permanecerão inalterados, a menos que seja especificamente alterado ou a energia elétrica seja removida do sistema (MAXFIELD, 2004; MOREIRA, 2009).

Ainda na década de 70 surgiu o projeto de um microprocessador que incorporava as funções de uma unidade de processamento central em um único circuito integrado. Os microprocessadores substituíram milhões de transistores o que diminuiu o tamanho dos dispositivos. Eles são utilizados principalmente para processamentos complexos e dentre algumas aplicações estão presentes nos computadores e aparelhos eletrônicos. As tecnologias SRAM e de microprocessadores são largamente utilizadas na maioria dos FPGAs (MAXFIELD, 2004; MOREIRA, 2009).

Os dispositivos lógicos programáveis (*Programmable Logic Devices, PLDs*) foram os primeiros circuitos integrados cuja arquitetura interna (determinada pelo fabricante) foi projetada de

tal forma que podia ser configurado ou programado em campo, no local de trabalho do desenvolvedor. O termo PLD pode ser atribuído a duas subcategorias: os SPLDs e os CPLDs. Os SPLDs referem-se aos primeiros PLDs fabricados, os quais originalmente continham um número modesto de unidades lógicas equivalentes e eram bastante simples. Os CPLDs são dispositivos que contêm uma determinada quantidade de funções SPLD que compartilham uma matriz de interconexões programáveis. Se comparado aos FPGAs, esses dispositivos contêm um número relativamente limitado de unidades lógicas que podem ser utilizadas para implementar funções menores e menos complexas (MAXFIELD, 2004; MOREIRA, 2009).

Durante todo este processo, existia ainda a fabricação de *chips* dedicados para uma aplicação, chamados de ASIC. Esses dispositivos podem conter centenas de milhares de unidades lógicas, e podem ser otimizados em termos de desempenho e de área ocupada. Em geral, o ASIC é concebido e construído para fim industrial e produção em larga escala. Portanto, as opções disponíveis para a construção de circuitos integrados eram ou a utilização de PLDs, altamente configuráveis, mas sem suporte a funcionalidades complexas, ou a utilização de ASICs para aplicações específicas (MAXFIELD, 2004; MOREIRA, 2009).

Em meados de 1985, uma companhia chamada Xilinx® introduziu a ideia de combinar o controle de usuário e o *time-to-market* do PLD com a densidade e os benefícios de arranjo de unidades lógicas. Isto gerou uma nova classe de circuito integrado chamado de FPGA. As FPGAs ocupam o meio-termo entre PLDs e ASICs, porque suas funcionalidades podem ser personalizadas na "programação em campo"como acontece com os PLDs, e além disso, podem conter milhares de unidades lógicas utilizadas para a execução de funcionalidades extensas e complexas que anteriormente só poderiam ser realizadas utilizando ASIC. O custo de um projeto em FPGA é menor do que o de um projeto em ASIC, apesar de componentes ASICs serem mais baratos quando utilizados na produção de larga escala. Ao mesmo tempo, implementar mudanças no projeto é mais fácil em FPGA e o tempo necessário para a especificação, desenvolvimento e implementação até a finalização do projeto é menor (MAXFIELD, 2004; MOREIRA, 2009).

Os FPGAs são circuitos integrados digitais que contêm blocos configuráveis de lógica com interconexões programáveis entre eles. Os projetistas podem configurar esses dispositivos de maneira que executem diferentes tarefas. A parte do nome FPGA referenciada como *Field Programmable* menciona o fato de que sua programação acontece "em campo", ou seja, no local do usuário, diferentemente de dispositivos cuja programação é feita apenas pelo fabricante. Isto significa que

os FPGAs podem tanto ser configurados no laboratório como podem ser modificados depois de já estarem em seus locais definitivos de operação. O termo *Gate Array* está relacionado ao conceito de arranjo de unidades lógicas. As unidades lógicas são componentes padrão formados por transistores que podem ser configurados independentemente e interconectados a partir de uma matriz de trilhas condutoras e chaves programáveis (MAXFIELD, 2004; ALTERA, 2011p).

Durante o início da década de 1990, o tamanho e a sofisticação dos FPGAs começaram a aumentar. O grande mercado eram os consumidores das áreas de telecomunicação e redes, as quais envolviam o processamento de grandes blocos de dados. Mais tarde, no final da década de 1990, a utilização de FPGA em indústria automotiva e em aplicações industriais passou a ter um grande crescimento. No início do ano 2000, já se encontrava disponível FPGAs de alto desempenho contendo milhares de unidades lógicas. Algumas das características destes dispositivos são a possibilidade de incorporar os núcleos de microprocessadores embarcados, além dos pinos de alta velocidade para serem usados como entradas e saídas. O resultado é que atualmente os FPGAs podem ser utilizados para implementar praticamente qualquer circuito digital, incluindo *software* e dispositivos de comunicação como rádio, radar, processamento de imagem e outras aplicações de processamento digital de sinais (MAXFIELD, 2004; ALTERA, 2011p).

Atualmente, os FPGAs são compostos por memória SRAM embarcada configurável (pode ser reprogramada quantas vezes for necessário) para realizar as conexões, blocos de entrada e saída de dados e blocos lógicos. Sendo mais específico, um FPGA contém componentes lógicos programáveis, chamados de elementos lógicos (LEs) e uma hierarquia de interconexões reconfiguráveis que permitem que os elementos lógicos possam estar fisicamente conectados. Um FPGA que utiliza células SRAM deve ser reprogramado toda vez que é ligado, pois o dispositivo FPGA é composto de uma memória volátil que perde as conexões ao ser desligada. Pode-se configurar LEs para executar funções complexas de combinações, ou portas lógicas meramente simples como AND e XOR. Na maioria dos FPGAs, os blocos lógicos também incluem elementos de memória, que podem ser simples *flip-flops* ou blocos completos de memória (MAXFIELD, 2004; ALTERA, 2011p).

Os blocos lógicos programáveis, presentes nos FPGAs, podem ser vistos como componentes que podem ser configurados independentemente. Atualmente, o bloco lógico programável é composto de vários recursos que podem implementar diversas funções aritméticas e lógicas. Estes recursos variam de acordo com o fabricante do FPGA, mas de maneira geral incluem LUTs (*Lookup Table*), registradores que podem atuar como *flip-flops*, memórias, multiplicadores e acumulado-

res programáveis. Cada FPGA é formado por um grande número de blocos lógicos programáveis interconectados através de uma matriz de trilhas condutoras e chaves programáveis baseados em SRAM. Para especificar a funcionalidade de cada bloco lógico programável, assim como seletivamente fechar as chaves da matriz de interconexão, é necessário gerar um arquivo binário que indique as configurações do FPGA. Esse arquivo binário é gerado a partir de ferramentas de *software* seguindo um determinado fluxo de projeto. A hierarquia de chaves programáveis em conjunto com os blocos lógicos programáveis permite que o sistema possa ser interconectado de acordo com a necessidade do projetista. A Figura 2.2 apresenta os blocos lógicos, as chaves de interconexão física interna e os blocos de entrada e saída de dados. (SKLIAROVA; FERRARI, 2003; ALTERA, 2011p; MAXFIELD, 2004)

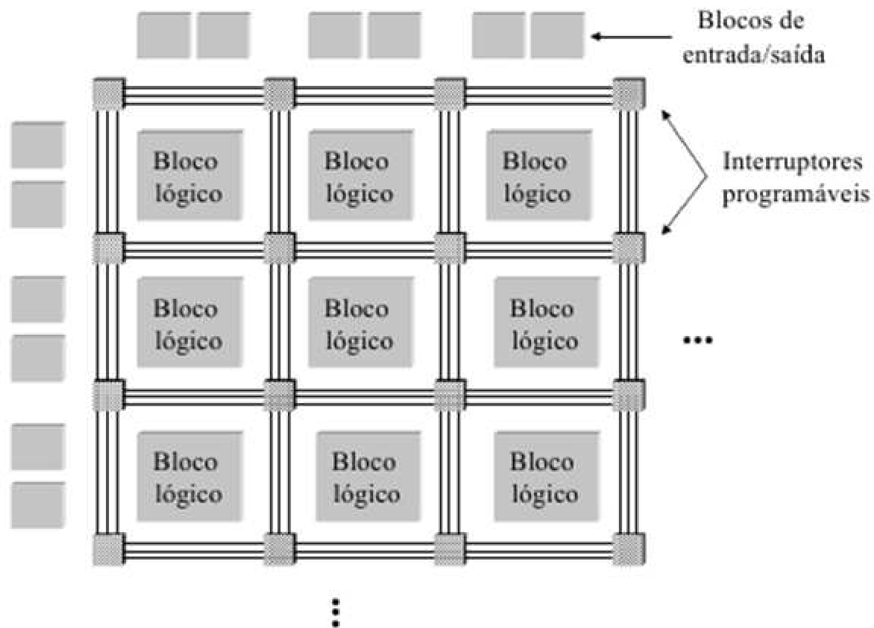


Figura 2.2 - Estrutura interna de um FPGA.
Fonte: (SKLIAROVA; FERRARI, 2003)

2.1.1 Linguagens para programação de FPGAs

Existem diversas maneiras de desenvolvimento e implementação de projeto de circuitos para FPGA. Os métodos tradicionais de fazer a programação de um FPGA incluem o uso de linguagem esquemática ou linguagem de descrição de *hardware*. A linguagem utilizada descreve as funções lógicas a serem sintetizadas no FPGA.

A linguagem esquemática apresenta diretamente as conexões e os elementos digitais que serão utilizados no processo, enquanto que as linguagens de descrição descrevem o funcionamento do *hardware* independente dos recursos digitais que serão utilizados. Logo, para utilizar este método de programação, é preciso ter em mente todo o *hardware* que deverá ser projetado (MOREIRA, 2009).

Normalmente utilizam-se linguagens de alto nível para descrever o funcionamento de um circuito através de um arquivo de texto ao invés de uma descrição gráfica no nível de lógica digital. As linguagens de descrição de *hardware* são utilizadas para descrever uma representação funcional e comportamental do circuito através de notação adequada e padronizada que independe do fabricante de *hardware* (MOREIRA, 2009).

As linguagens de descrição de *hardware*, HDL(*Hardware Description Language*) são bastante versáteis, por isso é importante entender de que maneira essa linguagem pode ser utilizada como parte do fluxo de projeto de um circuito digital. As funcionalidades de um circuito digital podem ser representadas em três diferentes níveis de abstração e a linguagem pode oferecer suporte a esses níveis de abstração, como mostrado na Figura 2.3. O nível mais alto de abstração é o comportamental, que permite descrever o comportamento do circuito através de laços e processos. Neste nível de abstração também é possível compor equações através de multiplicações e somas. O próximo nível de abstração possibilita descrever o funcionamento do circuito em termos de lógica combinacional (por exemplo, *if, then, else*) e booleana. Este nível de abstração também engloba a representação do circuito no nível de registros de transferências (RTL, *Register Transfer Level*), que consiste basicamente em uma representação por registradores interligados por lógica combinacional. O nível mais baixo de abstração de um HDL é o estrutural, que consiste em uma representação direta do circuito através de portas lógicas (CARDOSO; ARANTES, 2007; MAXFIELD, 2004).

A principal motivação para a utilização de linguagens de descrição de *hardware* é que estas

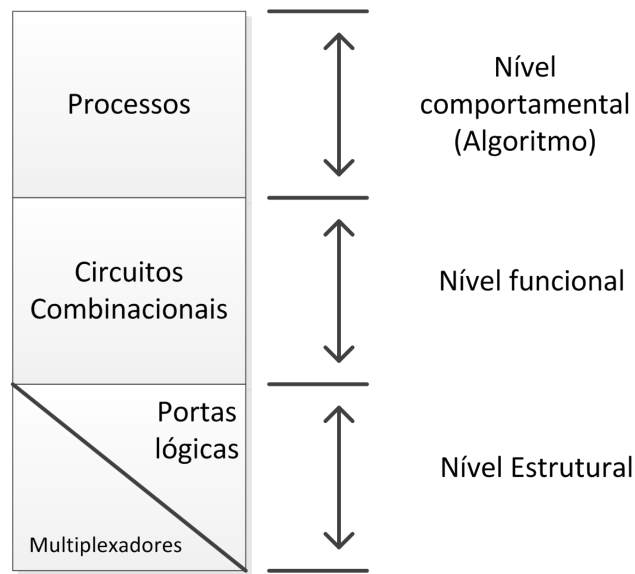


Figura 2.3 - Níveis de abstração em linguagens de descrição de *hardware*.
Fonte: Adaptado de Cardoso e Arantes (2007)

são linguagens padronizadas independentemente do fornecedor de *hardware* ou da tecnologia empregada, portanto é portátil e reutilizável. Uma vez que o código foi escrito, ele pode ser utilizado para implementar um circuito em um dispositivo programável ou pode ser utilizado no processo de fabricação de um *chip*. Atualmente, muitos *chips* comerciais são projetados utilizando essa abordagem (PEDRONI, 2004).

VHDL

Em 1980 o Departamento de Defesa dos Estados Unidos iniciou um programa para documentar o comportamento de ASICs que compunham os equipamentos vendidos às Forças Armadas americanas e desenvolveu uma linguagem para descrever o *hardware* de um *chip*. Isto quer dizer que a linguagem foi desenvolvida para substituir os complexos manuais que descreviam o funcionamento dos ASICs. Esta linguagem foi chamada de HDL (*Hardware Description Language*) e posteriormente de VHDL (VHSIC *Hardware Description Language*, onde VHSIC significa *Very High Speed Intergraded Circuit*). Ela foi certificada como um padrão IEEE em 1987 e vem recebendo revisões desde então (ALTERA, 2011n; ALTERA, 2011f).

De acordo com Mentor (1991), um projeto desenvolvido utilizando VHDL é composto de:

- Entidades (*Entities*)
- Arquiteturas (*Architecture*)
- Configurações (*Configuration*)
- Bibliotecas (*Packages*)

A entidade é responsável pela descrição das portas de entrada e saída dos circuitos, que servem para formalizar uma comunicação com outros circuitos. A arquitetura é responsável por descrever o comportamento dos circuitos, ou seja, o algoritmo utilizado para desempenhar sua funcionalidade. As arquiteturas podem possuir estruturas concorrentes e estruturas sequenciais que são descritas como um algoritmo. As estruturas do código são sempre concorrentes (paralelas), sendo que a lógica de conexão das mesmas especificam o comportamento final do circuito (PERRY, 2002).

Na Figura 2.4, como exemplo, temos um projeto completo, executando uma operação lógica conjuntiva entre os sinais de entrada. Observe que na declaração da entidade, denominada de "PORTA AND", encontra-se a especificação dos sinais e sua direção, entrada ou saída. Na declaração da arquitetura, denominada "comportamento", está o comportamento da entidade, a qual realiza a lógica e atribui o resultado no sinal de saída. A figura apresenta ainda o *hardware* gerado a partir do código VHDL (MENTOR, 1991).

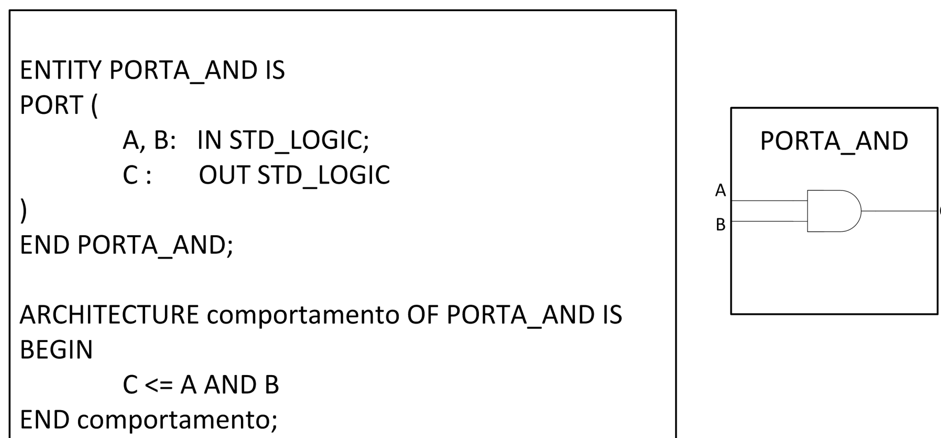


Figura 2.4 - Exemplo de projeto em VHDL.

2.1.2 Fluxo de projeto para FPGA

A mudança em desenvolver um projeto com base no esquema digital para o desenvolvimento com base no HDL trouxe melhorias significativas. Essa mudança permitiu aos desenvolvedores implementar os módulos em um padrão comportamental que era independente da tecnologia. No entanto, haviam muitos aspectos no projeto digital com o uso do HDL que poderiam ser melhorados. A fim de aperfeiçoar o desenvolvimento de projeto em circuitos integrados houve um esforço por parte da comunidade acadêmica e da indústria para definir um fluxo principal de projeto. O fluxo tradicional de projeto de circuitos para FPGA pode ser dividido em quatro fases distintas: especificação, verificação, implementação e depuração do sistema. É importante salientar que pode haver variação na especificação de cada fase do fluxo de projeto de acordo com o fabricante da FPGA (CARDOSO; ARANTES, 2007; MAXFIELD, 2004).

Na fase de especificação, o circuito é descrito através de uma linguagem de descrição. Através desta descrição, o *software* fornecido pelo fabricante do FPGA irá criar as conexões internas, resultando na lógica descrita. Nesta fase deve-se atentar ao modo de descrever os circuitos, pois apesar das linguagens oferecerem diversos recursos, nem todos podem ser sintetizados (convertidos em um circuito lógico) (CARDOSO; ARANTES, 2007).

Na fase de verificação, o programa escrito em linguagem de descrição de *hardware* é simulado através de ferramentas oferecidas pelo fabricante. O simulador é um *software* que confirma as funcionalidades de um circuito. Quando um projeto utiliza uma linguagem padronizada, por exemplo VHDL, há garantias de que o projeto pode ser reutilizado. Se um fornecedor de FPGA alterar suas bibliotecas, apenas é necessário fazer a compilação da fase de síntese. Depois de completar a fase de síntese é necessário verificar se o circuito projetado está trabalhando conforme o esperado. Esse é o objetivo da fase de verificação. No caso do simulador encontrar algum problema, o projeto deve voltar para a etapa de especificação. O projetista deverá corrigir a descrição do *hardware* e executar a etapa de verificação. Estimativas indicam que os projetistas utilizem 50% do tempo de desenvolvimento nas fases de especificação e verificação até que o projeto funcione como o esperado. Após o projeto funcionar apropriadamente, inicia-se a fase de implementação (CARDOSO; ARANTES, 2007; MAXFIELD, 2004).

Nesta fase, o circuito descrito é compilado e sintetizado no FPGA. Ela é realizada necessa-

riamente com o uso do ambiente de programação do fabricante do FPGA. Esta fase compreende as etapas de tradução, posicionamento e roteamento, analisador de tempo e *download* do código para o FPGA. Durante a tradução, o *software* analisa o código e gera os elementos de *hardware* necessários para criar o circuito dentro do FPGA. Feito isto, o *software* parte para a etapa de posicionamento destes elementos e roteamento dos sinais existentes entre eles. Então, através de restrições criadas pelo usuário (ou utilizando as configurações padrão), parte para a análise do tempo de propagação dos sinais. Enfim, cabe ao usuário fazer o *download* dos dados gerados pelo *software* para o FPGA (CARDOSO; ARANTES, 2007; ALTERA, 2011e).

Ao fazer o *download* do código para o FPGA, dá-se início à fase de depuração. Nessa fase, é preciso verificar se o projeto está funcionando no dispositivo real conforme havia sido planejado. Encontrar problemas nessa fase significa que a especificação do projeto para o dispositivo em uso não foi realizada de maneira apropriada ou que alguns aspectos dos sinais, cuja origem ou destino sejam o dispositivo, não foram considerados. Desta forma, deve-se coletar os dados do problema e corrigir a modelagem do projeto voltando a fase de especificação.

2.2 Microprocessadores em *hardware* reconfigurável

Os FPGAs tem sido amplamente empregados para a criação de circuitos digitais específicos. Contudo, existem outros recursos que podem ser aproveitados, como por exemplo, o uso de microprocessadores totalmente escritos em linguagem de descrição de *hardware*. Estes têm a vantagem de permitir que seja criado um processador com *hardware* totalmente dedicado à aplicação com a vantagem de utilizar circuitos adicionais criados pelo próprio usuário. Isto abre portas para que o sistema tenha um módulo em hardware, que agregado a um processador pode permitir que este execute funções complexas de forma muito mais rápida.

2.2.1 Processador *Soft-core* NIOS 2 e SoPC *Builder*

O processador *soft-core* NIOS 2 (ALTERA, 2011k) é um processador escrito totalmente em linguagem de descrição de *hardware*, por isto é chamado de *soft-core* ou seja, o núcleo do processador pode ser alterado. Trata-se de um núcleo que possui arquitetura *Harvard* RISC de 32-bits. Foi desenvolvido internamente pela Altera® e pode ser personalizado de acordo com a necessidade, sendo possível alterar desde suas configurações até suas instruções aritméticas. Sua única limitação é a quantidade de elementos lógicos presentes no FPGA em uso, visto que ele é livre de *royalties*.

Existem três versões disponíveis: econômico, padrão e rápido. A Tabela 2.1 apresenta as principais diferenças entre elas.

Tabela 2.1 - Principais diferenças entre as versões do processador NIOS 2.

Item	Econômico	Padrão	Rápido
Performance a 100MHz	15 MIPS	64 MIPS	113 MIPS
Elementos lógicos consumidos	600 a 700	1200 a 1400	1400 a 1800
Multiplicador em *hardware*	Não possui	3 ciclos	1 ciclo

É possível adicionar lógica personalizada diretamente na unidade lógica aritmética (ULA) do processador; veja na Figura 2.5 o bloco de *Custom Instructions*. Assim, pode-se desenvolver suas próprias instruções, como instruções de ponto flutuante.

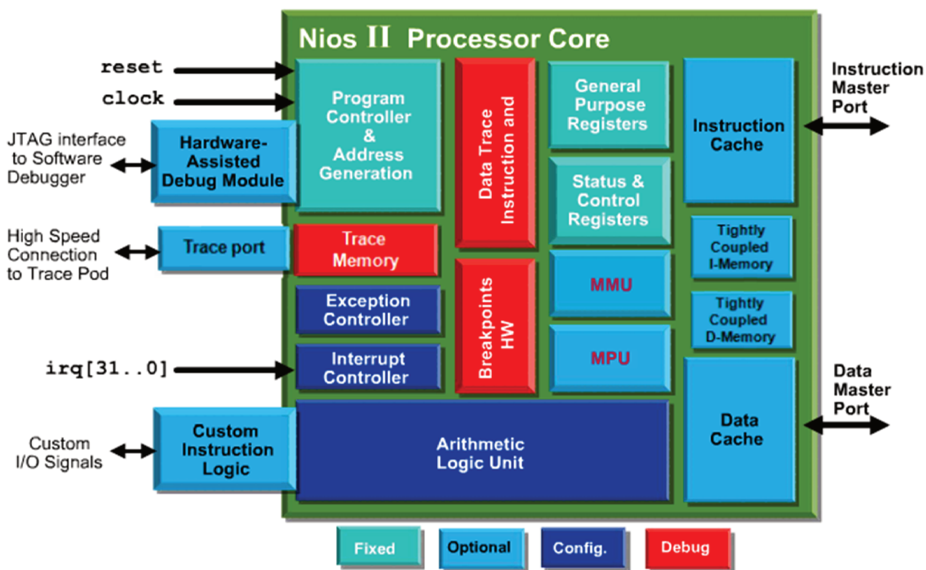


Figura 2.5 - Estrutura interna do soft-core NIOS2.
Fonte: (ALTERA, 2011k)

Para desenvolver projetos com FPGAs da Altera®, utiliza-se o *software* Quartus 2 IDE. Trata-se de uma ferramenta para compilação, síntese e análise de projetos escritos em linguagem de descrição de *hardware* ou diagramas esquemáticos, fornecida gratuitamente. Para desenvolver o processador NIOS 2, utiliza-se o *software* SoPC *Builder* (*System on a Programmable Chip*), uma extensão do Quartus 2 IDE. Ele inclui além do núcleo processador NIOS 2, diversos periféricos (ALTERA, 2011c). No processo de desenvolvimento, os periféricos desejados são instanciados no SoPC *Builder*, montando o processador de acordo com sua necessidade. Feito isto, segue-se para a etapa de instanciar o processador desenvolvido pelo SoPC *Builder* dentro do projeto em desenvolvimento no Quartus 2 IDE, utilizando qualquer uma das linguagens aceitas pelo ambiente de desenvolvimento. O projeto deve ser compilado para enfim ser sintetizado no FPGA (ALTERA, 2011l).

Com o circuito sintetizado no FPGA, deve-se desenvolver o *software* que será executado pelo processador. Isso é realizado utilizando o NIOS 2 IDE *Eclipse* e a linguagem C ou C++ para criar o código, contando-se com o apoio de sistemas operacionais de tempo real e bibliotecas de acesso ao *hardware* (HAL, *Hardware Abstraction Layer*).

Barramento Avalon e Periféricos

Como pode ser visto na Figura 2.6, todas as conexões entre o processador e um periférico são realizadas através do sistema de interconexão, chamado *System Interconnect Fabric Avalon Bus*. Existem dois tipos diferentes de barramentos de dados disponíveis (*Memory Map e Stream*), variando na forma de conexão e manipulação dos dados. Além dos barramentos de dados e suas interfaces, ainda existem as interfaces de *reset*, *clock*, interrupção e sinais externos ao sistema. A interface de interrupção envia uma requisição de interrupção ao processador. A interface de sinais externos, chamada de *Conduit*, tem o objetivo de enviar ou receber sinais externos ao sistema, como por exemplo os sinais de uma comunicação de dados. O funcionamento de cada interface está descrito no manual de especificação do barramento Avalon (ALTERA, 2011a).

De acordo com a especificação deste barramento (ALTERA, 2011a), o tipo *Memory Map* (MM) é baseado em endereços de memória. Toda conexão de um periférico ao barramento MM é realizada através de uma interface MM. Esta interface pode ser de 2 tipos: mestre ou escravo. A interface mestre acessa a interface escrava através do barramento. A interface escrava sempre possui um endereço base e um endereço final, que engloba todos os seus registradores internos. Estes registradores internos podem ser endereços de pequenas memórias internas ao módulo, registradores de controle, de estado, ou qualquer outro registrador que possa ser acessado através de um endereço. A Figura 2.6 apresenta o barramento Avalon MM com seus módulos internos e diversos periféricos do tipo MM, tanto mestres quanto escravos.

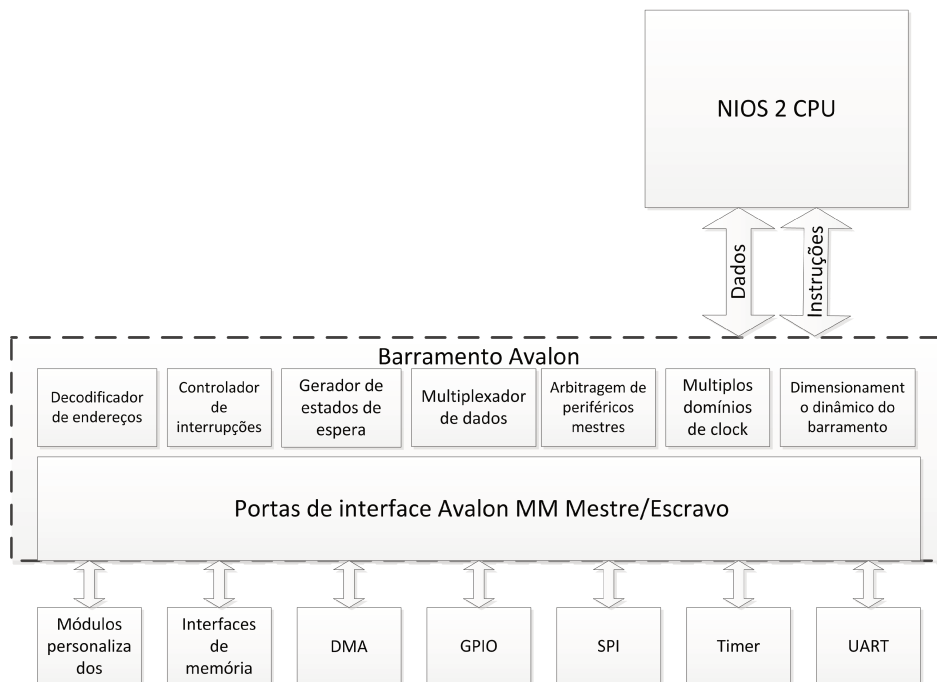


Figura 2.6 - Barramento Avalon *Memory Map* e seus módulos internos.
Fonte: Adaptado de Altera (2011a)

Existem vários sinais envolvidos em uma transação de dados entre uma interface mestre e uma interface escrava. A Figura 2.7 apresenta alguns destes sinais e suas formas de onda durante um comando de leitura e um comando de escrita de dados.

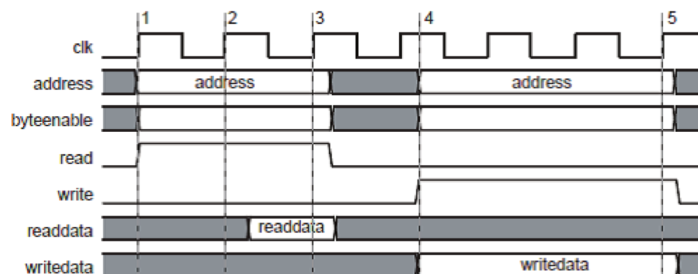


Figura 2.7 - Comando de leitura e de escrita de dados em uma interface Avalon MM.
Fonte: (ALTERA, 2011a)

O barramento de fluxo de dados, chamado de *Stream*, tem o objetivo de trocar dados em alta velocidade através de uma interface mais simples. Esta interface pode ser de dois tipos: geradora de dados (*Source*) ou receptora de dados (*Sink*). A interface geradora de dados sempre está conectada a uma interface receptora de dados. Existem diversos periféricos que utilizam esta interface para

transmitir dados de forma mais rápida e sem ocupar o processador para tal tarefa, como os módulos de acesso direto à memória (DMA). A Figura 2.8 apresenta alguns dos sinais envolvidos numa transmissão de dados com a interface *Stream* (ALTERA, 2011a).

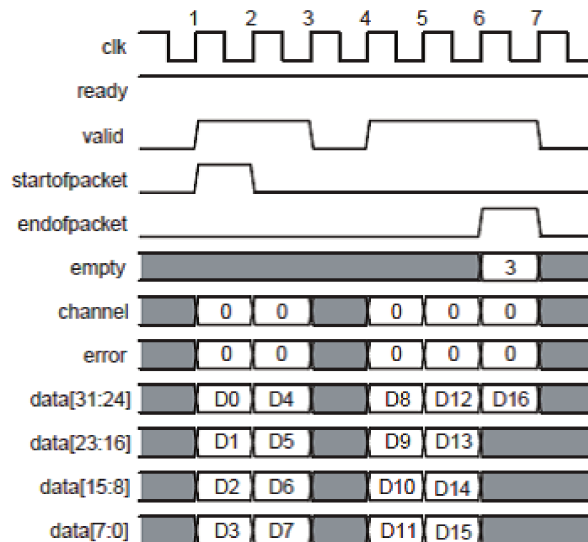


Figura 2.8 - Transmissão de dados através da interface Avalon ST.
Fonte: (ALTERA, 2011a)

O SoPC *Builder* possui uma variedade de periféricos que podem ser adicionados ao sistema em construção. Dentre eles os mais comumente utilizados são: temporizador, memórias *On-Chip*, memórias FIFO, medidor de desempenho, acesso direto à memória (DMA), controladores de memória (SDRAM, SRAM, FLASH), processador NIOS2, controlador de rede *ethernet*, comunicação serial síncrona e assíncrona, pinos de entrada e saída de uso geral. Cada um destes módulos existentes possui além de sua descrição em *hardware* e importação ao SoPC *Builder*, um modelo no ambiente de programação em C/C++. Neste ambiente é necessário, através de instruções para o processador, configurar e utilizar estes módulos. É preciso conhecer detalhadamente cada periférico utilizado no sistema em criação, para que no momento do desenvolvimento do *software*, ele seja corretamente utilizado (ALTERA, 2011l).

2.2.2 Kits de desenvolvimento

A elaboração de todo o projeto foi realizado sobre uma placa de desenvolvimento completa para FPGA, denominada Altera NIOS II *Embedded Evaluation Kit* (NEEK) *Cyclone* III *Edition* (conforme a Figura 2.9), que inclui ferramentas de desenvolvimento de *hardware* e *software*, documentação, acessórios e periféricos necessários para criar sistemas embarcados (ALTERA, 2011h; ALTERA, 2011i).

Figura 2.9 - Kit de desenvolvimento NEEK.
Fonte: (ALTERA, 2011h)

A placa de desenvolvimento inclui um FPGA Altera *Cyclone* III EP3C25F324 pré-configurado com um projeto de *hardware* armazenado na memoria *flash*, assim como várias demonstrações armazenadas no cartão de memória. Toda vez que o sistema é ligado, o *hardware* armazenado na memória *flash* é gravado no FPGA através de um CPLD dedicado a esta função. Após esta gravação, um programa seletor de aplicações é executado sob este *hardware* recém-gravado, permitindo que seja executado um projeto armazenado no cartão de memória. Pode-se criar uma aplicação e armazená-la no cartão de memória, ou utilizar as ferramentas de desenvolvimento para gravar o projeto diretamente no FPGA utilizando um cabo USB. A Figura 2.10 apresenta a estruturação do NEEK, que possui duas placas principais (*Cyclone* III FPGA *Starter Board* e LCD *Multimedia Daughtercard*), onde cada uma delas possui uma variedade de periféricos (ALTERA, 2011h; ALTERA, 2011i).

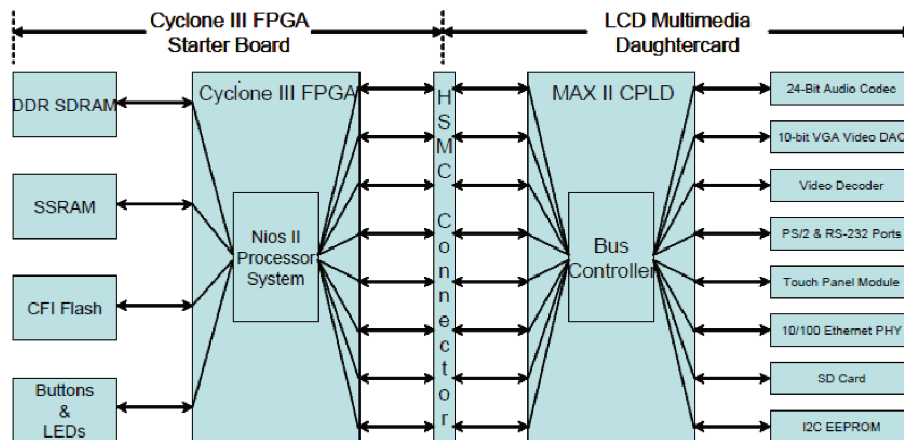


Figura 2.10 - Periféricos do kit de desenvolvimento NEEK.
Fonte: (ALTERA, 2011i)

Visto a parte do dispositivo de *hardware* utilizado, se faz necessário apresentar as ferramentas de *software* que realizam a comunicação e também o projeto de construção do processador que irá processar o algoritmo proposto:

- Quartus 10.1 SP1: utilizado no desenvolvimento dos códigos em VHDL para a descrição e síntese do *hardware*. Também foi utilizada uma extensão chamada SoPC *Builder* que realiza a montagem e a configuração do processador NIOS 2 e seus periféricos. (ALTERA, 2011l)
- NIOS 2 IDE 10.1 SP1: ambiente de programação em C/C++, utilizado para criar o *software* que será executado pelo processador NIOS 2, desenvolvido no Quartus 10.1.

2.3 Microcontrolador *Coldfire* FreeScale®

2.3.1 O microcontrolador MCF51CN128

O MCF51CN128 é um microcontrolador de 32 *bits*, núcleo *ColdFire* V1 da Freescale®, de baixo custo e consumo, barramento de até 50.33MHz, com 128 KB de memória *flash*, 24KB de memória RAM, um conversor analógico-digital (ADC), que suporta diversos tipos de comunicação,

entre eles a *ethernet*. Ele possui 70 pinos que podem ser utilizados como entrada ou saída. A Figura 2.11 apresenta uma das versões do microcontrolador. (FREESCALE, 2011b)

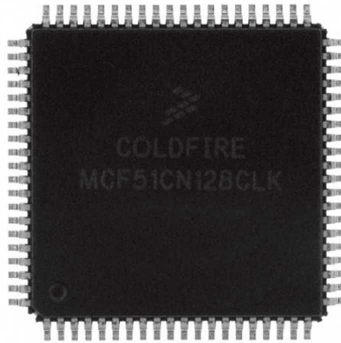


Figura 2.11 - Microcontrolador MCF51CN128.

O seu conversor analógico-digital possui 12 canais multiplexados, ou seja, apenas um pode ser lido no mesmo instante de tempo. Possui 12 *bits* de resolução e pode chegar a até 2.5 microssegundos de tempo de conversão quando em configurações mais simples. Suporta o modo de conversão contínua, onde realiza conversões em sequência, uma após outra, sem necessidade de um disparo. Também possui a funcionalidade de comparação em *hardware*, o que permite criar um nível de disparo para conversões, o que pode garantir que o primeiro ponto de um sinal seja capturado sempre com o mesmo valor inicial, como em aplicações que requerem sincronismo. A tensão de referência para as conversões é a própria alimentação do microcontrolador, que pode ir de 1.8 V a 3.6 V. O resultado da conversão é sempre proporcional ao valor aplicado, tendo a alimentação como limite. (FREESCALE, 2011b)

2.3.2 Kits de desenvolvimento

A FreeScale® oferece uma ferramenta de desenvolvimento para este microcontrolador, o módulo TOWER (TWR-MCF51CN128). Trata-se de um conjunto de placas que quando montados se assemelham a uma torre. É um kit de desenvolvimento modular, onde o usuário pode substituir as placas e montar outras configurações com os módulos existentes. A Figura 2.12 apresenta o TOWER. Existem dezenas de módulos desenvolvidos tanto pelo fabricante do microcontrolador

quanto por terceiros. Estes módulos vão de placas com diferentes microcontroladores a placas com sensores e dispositivos de comunicação. Neste projeto foram utilizados o módulo processador MCF51CN e o módulo de comunicação serial com rede *ethernet*.

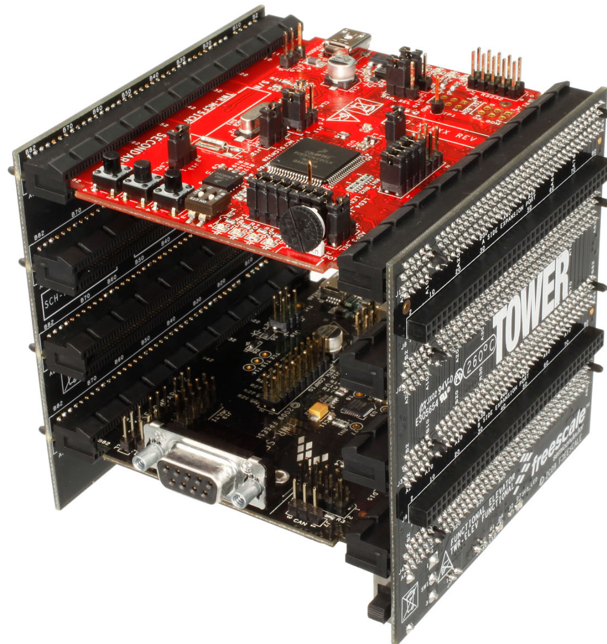


Figura 2.12 - Kit de desenvolvimento TOWER.

2.4 Sistemas operacionais de tempo real

Projetar aplicações para sistemas embarcados é sempre um desafio. Uma maneira de diminuir a complexidade da aplicação é a utilização de um projeto orientado a tarefas, dividindo o projeto em pequenas tarefas mais simples de serem gerenciadas. Cada tarefa é então responsável por uma parte da aplicação. Com isto, algumas tarefas têm mais prioridade sobre outras, ou seja, possuem requisitos de tempo real para responderem mais rapidamente e corretamente (IAR, 2012).

De acordo com Shaw (2003), um sistema operacional tempo-real (RTOS, *Real Time Operating Systems*) é um programa que agenda a realização de tarefas de maneira temporal, atribuindo prioridades a estas tarefas, gerenciando os recursos do sistema e fornecendo uma base sólida para

o desenvolvimento do código de uma aplicação. Essas aplicações vão de simples brinquedos até sistemas complexos quanto a navegação de uma aeronave.

Então, se uma aplicação utiliza um sistema operacional de tempo real, já estão presentes características que priorizam tarefas em relação a outras. Além disto, existem recursos em um sistema operacional que garantem sincronismo e comunicação entre as tarefas. Ao usar um sistema operacional de tempo real, fica garantido que tarefas críticas com relação ao tempo são executadas dentro de suas restrições (IAR, 2012).

2.4.1 Características principais

O núcleo de um RTOS é conhecido como *kernel*. Existem funções chamadas de API (*Application Programming Interface*) que acessam o *kernel* para criar tarefas e recursos do RTOS. O *kernel* também contém um escalonador. O escalonador é responsável por executar as tarefas de acordo com o tipo de escalonamento. A grande diferença entre os tipos de escalonamento é como eles distribuem o tempo do processador, ou seja, a execução do programa, entre as várias tarefas que o escalonador está gerenciando. O tipo mais popular de escalonamento é o preemptivo, baseado em prioridade. Um escalonador preemptivo baseado em prioridade suporta escalonamento tanto preemptivo quanto não preemptivo. Em uma situação preemptiva, uma tarefa de maior prioridade imediatamente interrompe uma tarefa em execução com prioridade menor (IAR, 2012).

A maior parte dos *kernels* utiliza um sistema de temporização por interrupção de geralmente 10ms, a qual é chamada de *tick*. Sem um sistema de *ticks* em um RTOS, o escalonamento simples ainda é possível, porém os serviços baseados no tempo não estão disponíveis. O sistema de *ticks* pode ser implementado com um dos temporizadores em *hardware* implementados no microcontrolador em questão (IAR, 2012).

Uma tarefa é como uma função que tem sua própria pilha (chamadas de funções, variáveis, etc) e um conjunto de dados de controle (*Task Control Block*), que armazenam informações sobre o estado da tarefa (em execução, em espera, etc). Ao contrário da maioria das funções, uma tarefa quase sempre é um laço de repetição infinito, ou seja, uma vez criada, a execução nunca deixará o laço. Uma tarefa que está pronta para ser executada está em um estado de PRONTA (*READY*), ou,

ela pode estar suspensa, esperando algo acontecer antes de entrar no estado de PRONTA. Este estado é chamado de estado de ESPERA (*WAITING*). Cada RTOS pode dar um nome a estes estados, e podem existir outros como executando, terminado, completo (IAR, 2012).

Como dito, existem dois tipos principais de escalonadores. Em um escalonador orientado a eventos (algoritmo de escalonamento com prioridade controlada), cada tarefa possui uma prioridade e a tarefa com a maior prioridade é executada. A ordem de execução depende somente desta prioridade. A regra neste caso é: o escalonador ativa a tarefa que possui a maior prioridade da lista de tarefas prontas para serem executadas (IAR, 2012).

Outro tipo de escalonador é o de tempo compartilhado, também chamado de *Round-Robin*. Com esta regra, o escalonador tem uma lista das tarefas que compõem o sistema e usa essa lista para verificar o estado da próxima tarefa. Se esta tarefa está pronta para ser executada, a tarefa será executada. Assim, cada tarefa tem uma fatia de tempo do processador (IAR, 2012).

Também é necessário realizar a comunicação de informações entre diferentes tarefas em um RTOS. Isso pode ser feito através de eventos, semáforos, ou mensagens trocadas entre as tarefas. A forma mais comum de comunicação entre tarefas é através de eventos. Eventos são sinais solicitados pelas tarefas que, quando não estão disponível, fazem a tarefa entrar no estado de espera. Uma outra tarefa pode enviar um evento para uma tarefa que estava em estado de espera, e quando isto acontece, o RTOS retira a tarefa deste estado (IAR, 2012).

Outro método de comunicação entre tarefas é o semáforo. São utilizados geralmente para proteger recursos compartilhados, como por exemplo, duas tarefas que precisam usar a mesma porta serial de um microcontrolador. Assim a primeira tarefa solicita um recurso ao semáforo, que neste caso só possui um recurso. A segunda tarefa, ao tentar acessar a porta serial, pede o recurso ao semáforo. Como este não está disponível, a segunda tarefa entra no estado de espera até que a primeira tarefa libere este recurso. Casos onde o semáforo possui somente um recurso também são chamados de exclusão mútua (*Mutex*). Isto garante que somente uma tarefa usará o recurso a cada vez (IAR, 2012).

Mensagens permitem que dados sejam enviados de uma tarefa para uma segunda ou mais tarefas. Estas mensagens podem ter qualquer tamanho e geralmente são implementadas através de caixa de mensagens ou fila de mensagens. O comportamento destes elementos variam de acordo com o fabricante do RTOS (IAR, 2012).

A Figura 2.13 apresenta um resumo de todas as características apresentadas.

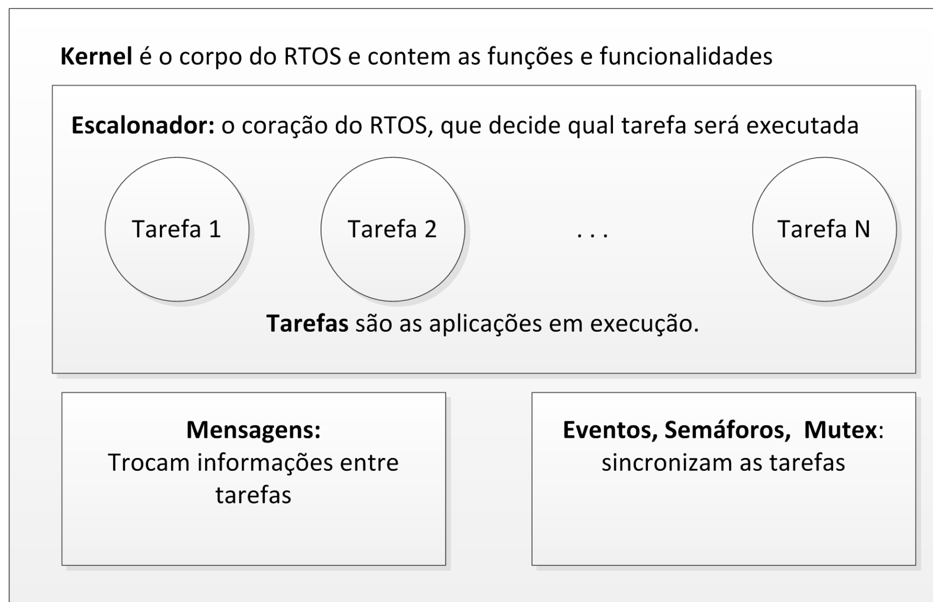


Figura 2.13 - Conceitos básicos de um sistema operacional.

2.4.2 RTOS uC/OS2

O MicroC/OS-II é um *kernel* de tempo real popular produzido pela Micrium Inc. O MicroC/OS-II é portátil, de baixo *footprint* (tamanho reduzido do código do sistema operacional), preemptivo, de tempo real e multitarefa. Lançado pela primeira vez em 1992, MicroC/OS-II ainda é usado em centenas de aplicações comerciais. Ele é implementado em mais de 40 arquiteturas de processadores diferentes, além do processador NIOS 2. O MicroC/OS-II oferece os seguintes serviços:

- Tarefas (*tasks*)
- Eventos
- Mensagens
- Semáforos
- Gestão da memória
- Gestão do tempo

No processador NIOS 2, O *kernel* MicroC/OS-II funciona acima da camada de abstração de *hardware* (HAL). Devido a esta arquitetura, MicroC/OS-II para o NIOS 2 tem as seguintes

vantagens:

- Programas são portáveis para outros sistemas de *hardware* NIOS 2.
- Programas são resistentes a mudanças no *hardware*.
- Os programas podem acessar todos os serviços de HAL.
- Interrupções são fáceis de implementar.

(LABROSSE, 2002; ALTERA, 2011g; MICRIUM, 2011)

2.4.3 RTOS MQX

O MQX é um sistema operacional de tempo real fornecido pela FreeScale Semiconductors® para as famílias de processadores de 32 *bits*. O RTOS MQX é utilizado em milhares de projetos de sistemas embarcados, por várias companhias como ABB, Agilent, ATI Technologies, General Electric, Philips, Porsche, Sony, Tyco e Xerox; em aplicações como controle industrial, redes, armazenamento e eletrônica de consumo. O MQX é composto por um núcleo de componentes e alguns outros componentes adicionais. A Figura 2.14 mostra os componentes do *kernel* (no centro) e os componentes opcionais (ao redor da parte central). (FREESCALE, 2011c; FREESCALE, 2011a)

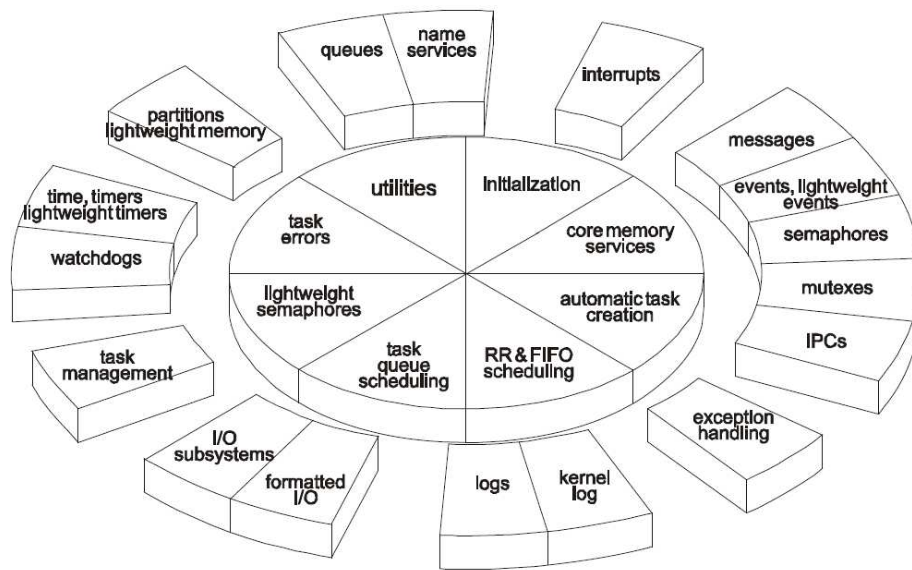


Figura 2.14 - Componentes do núcleo e componentes adicionais do MQX.
Fonte: (FREESCALE, 2011a)

O MQX oferece os serviços de escalonamento de tarefas do tipo preemptivo baseado em prioridade, do tipo *Round-Robin*, tempo compartilhado ou por escalonamento direto definido pelo usuário através de filas de tarefas. Oferece recursos de comunicação entre tarefas tais como eventos, semáforos, *mutex* e mensagens, e várias funções de gerência de tempo e chamadas diretas do *kernel*. É possível adicionar bibliotecas, como a RTCS (*Real Time Communication Suite*), que permite utilizar a comunicação em rede *ethernet*.

2.5 Representação numérica de ponto fixo

Um conjunto de *N* (sendo *N* um inteiro positivo) dígitos binários (*bits*) tem 2^N estados possíveis, pois existem duas possibilidades para cada dígito. Esses estados podem representar tudo que for contábil, como alunos em uma escola, elementos em um sistema e outros. O ponto em questão é que não há nenhum significado inerente a uma palavra binária, embora a maioria das pessoas sejam levadas a pensar nelas (à primeira vista, pelo menos) como números inteiros positivos, ou seja, a representação binária natural, onde cada *bit* tem peso igual a 1. No entanto, o significado de uma palavra binária de *N bits* depende inteiramente da sua interpretação, ou seja, do conjunto de

representação e do mapeamento que foi escolhido para ser usado (YATES, 2007).

Desta forma, consideremos que uma palavra binária representa um subconjunto de números racionais. Lembrando que os números racionais são o conjunto de elementos que podem ser expressos como a/b, onde $a, b \in Z$, e $b \neq 0$, onde $\mathbb{Z}$ é o conjunto dos inteiros. O subconjunto que nos referimos são aqueles onde $b = 2^N$. Para este exemplos, devemos restringir ainda mais o conjunto de elementos para aquele em que todos os elementos do subconjunto possuem a mesma quantidade total de *bits*, e o ponto fracionário na mesma posição, ou seja, o ponto é fixo. Essa representação é chamada de "ponto-fixo"(YATES, 2007).

2.5.1 Números racionais positivos

Uma palavra binária de N *bits*, quando interpretada como um número racional positivo, pode ter valores de um subconjunto P de racionais positivos dados por

$$P = \{p/2^b | 0 \leq p \leq 2^N - 1, p \in Z\} \tag{2.1}$$

Note que o subconjunto P possui 2^N elementos. Desta forma, podemos ter uma representação U(a, b), onde $a = N - b$ e a representa a quantidade de dígitos na parte inteira e b a quantidade de *bits* na parte fracionária. Um número inteiro pode ser representado usando esta notação, considerando que a quantidade de *bits* na parte fracionária é igual a zero (YATES, 2007).

A escala de valores representáveis pela notação U(a, b) varia de 0 a $(2^N - 1)/2^b = 2^a - 2^b$. Por exemplo, um número de ponto fixo representado como U(6, 2) tem a forma

$$b_5 b_4 b_3 b_2 b_1 b_0 . b_{-1} b_{-2} \tag{2.2}$$

sendo que este *bit* b_k tem um peso de 2^k. Neste caso, pode-se representar números de ponto fixo de 0 a $2^6 - 2^{-2} = 64 - 1/4 = 63 \quad 3/4$ (YATES, 2007).

2.5.2 Complemento de 2 em números binários

Considere uma palavra de N *bits* x interpretada como sendo uma representação de números inteiros, ou seja, U(N, 0). O complemento de um da palavra x é uma operação que inverte cada bit desta palavra. Isso pode ser feito de forma aritmética, subtraindo a palavra x de $2^N - 1$. Assim, chamando o complemento de um de $\tilde{x}$, temos que $\tilde{x} = 2^N - 1 - x$.

O complemento de dois da palavra inicial x, chamado de $\hat{x}$, é encontrado somando o valor 1 ao complemento de um de x, ou seja $\hat{x} = \tilde{x} + 1 = 2^N - x$ (YATES, 2007).

2.5.3 Números racionais sinalizados

Uma palavra de N *bits*, quando interpretada como um número racional sinalizado, pode assumir valores do subconjunto P definido por

$$P = \{p/2^b | -2^{N-1} \leq p \leq 2^(N-1) - 1, p \in Z\}. \tag{2.3}$$

Perceba que P possui 2^N elementos. Desta forma, podemos ter uma representação S(a, b), sendo $a = N - b - 1$ representando a quantidade de dígitos na parte inteira sem o sinal e b a quantidade de *bits* na parte fracionária. Nesta representação, o *bit* mais significativo representa o sinal do número, indicando se o número é positivo ou negativo.

A escala desta representação vai de $-2^{N-1-b} \leq x \leq +2^{N-1-b}$. Por exemplo, o número S(13, 2) possui $1 + 13 + 2$ *bits* e pode variar de $2^{-13} = -8192$ a $2^{13} - 2^{-2} = 8191.75$.

2.5.4 Regras básicas para aritmética em ponto fixo

Existem algumas regras de formação binária quando realizamos operações matemáticas em números de ponto fixo. Na adição, os dois números devem ser colocados na mesma escala, ou seja,

mesma quantidade de *bits* para ambas as partes, inteira e fracionária. O resultado de uma adição de dois números de N *bits* requer N+1 *bits*, em função do possível estouro da soma (YATES, 2007).

Uma multiplicação de dois números de ponto fixo positivos necessita do dobro de *bits* para armazenar o resultado ou seja:

$$U(a1, b1) \times U(a2, b2) = U(a1 + a2, b1 + b2) \tag{2.4}$$

Quando os números de pontos fixos são valores sinalizados, então temos a seguinte regra

$$S(a1, b1) \times S(a2, b2) = S(a1 + a2 + 1, b1 + b2) \tag{2.5}$$

Assim serão necessários o dobro de *bits* mais um para representar o resultado, quando os dois operandos possuem a mesma quantidade de *bits*. O acréscimo de uma unidade na parte está relacionado com a presença de um *bit* de sinal em cada operando e apenas um bit de sinal no resultado (YATES, 2007).

Existem dois tipos de deslocamento de *bits* em uma palavra, chamados aqui de literal e virtual. O deslocamento literal realmente movimenta os *bits* de uma palavra, ou para a direita ou para a esquerda, com a intenção de multiplicar por dois, dividir por dois, ou aplicar uma escala na palavra binária. Em todos os casos, há uma possível perda da precisão, em função do estouro do resultado, uma vez que a entrada e a saída possuem a mesma quantidade de dados. Quando numa multiplicação ou divisão, o resultado em termos de formação permanece constante, tem-se

$$U(a, b) \gg N = U(a, b) \tag{2.6}$$

Porém, quando ocorrer uma alteração na escala de um número, temos que

$$U(a, b) \gg N = U(a + n, b - n) \tag{2.7}$$

Um deslocamento virtual movimenta o ponto fixo sem modificar os valores. Pode ser usado como uma alternativa para realizar multiplicações e divisões por dois. É chamada de deslocamento virtual porque na verdade não realiza deslocamento algum, trata-se apenas de uma reinterpretação dos dados. Com isto não há *overflow* nem perda de resolução (YATES, 2007).

2.6 Processamento espectral de sinais

Os avanços na tecnologia de fabricação de *chips* e sua aplicação no projeto eficiente de sistemas digitais nos últimos 50 anos fizeram surgir uma nova área do conhecimento denominada processamento digital de sinais ou DSP (*Digital Signal Processing*). Através do uso de DSP, sistemas de comunicação sofisticados evoluíram, a Internet emergiu, os sinais astronômicos puderam ser transformados em informações valiosas sobre o universo, sinais sísmicos puderam ser analisados para determinar a força de um terremoto ou para prever a estabilidade de um vulcão, imagens de computador ou fotografias puderam ser melhoradas, e assim por diante (ANTONIOU, 2006).

Sinais surgem em quase todos os campos da ciência e da engenharia, por exemplo, na astronomia, acústica, biologia, comunicação, sismologia, telemetria, e economia. Sinais podem surgir naturalmente ou são criados pelo homem, através de determinados processos físico-químicos. Um sinal é alguma quantidade, propriedade ou variável que depende do tempo, por exemplo, a intensidade da luz de uma estrela ou a força do sinal sísmico. Nestes casos, temos um sinal que acontece em função do tempo, porém, podem existir sinais que possuem mais de uma variável independente, como é o caso de uma fotografia, que é um sinal de duas dimensões, ou uma imagem de TV, que possui uma terceira dimensão, o tempo (ANTONIOU, 2006).

Sinais podem ser classificados como contínuos no tempo ou discretos no tempo. Sinais contínuos no tempo estão definidos a cada instante do início ao fim. Por outro lado, sinais discretos estão definidos em momentos discretizados no tempo, por exemplo, a cada milissegundo, segundo, dia. Sinais na natureza normalmente são contínuos no tempo, enquanto que sinais produzidos pelo homem podem ser ou não contínuos no tempo, dependendo da aplicação e do sistema que o gerou, um sistema analógico ou digital. Sinais discretizados são geralmente gerados através do uso de conversores analógico-digital que a cada momento captura uma amostra do sinal contínuo no tempo.

Os sinais são mais intuitivamente representados em termos de funções no tempo, ou seja, $x(t)$ para sinais contínuos ou $x(nT)$ para sinais discretos. Em muitas situações, é útil representar os sinais em termos de funções na frequência. A representação no domínio da frequência fornece uma descrição detalhada e significativa dos componentes de frequência individual, ou seja, as suas frequências, amplitudes e ângulos de fase. A Figura 2.15 apresenta um sinal representado no domí-

nio do tempo e um sinal representado no domínio da frequência (ANTONIOU, 2006).

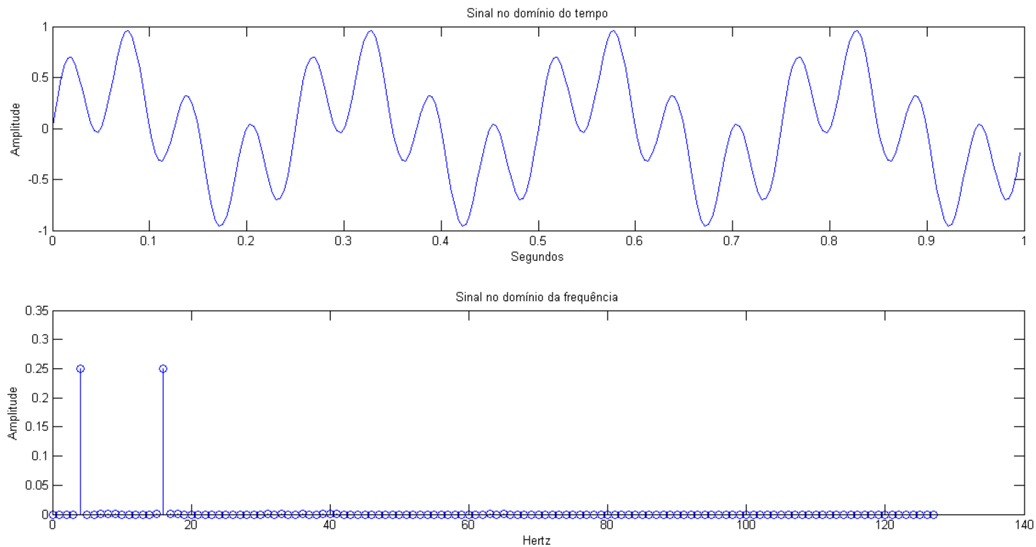


Figura 2.15 - Sinal representado no domínio do tempo e no domínio da frequência.
Fonte: Adaptado de Antoniou (2006)

Cientistas, matemáticos e engenheiros desenvolveram uma variedade de ferramentas matemáticas que podem ser usadas para a representação espectral de diferentes tipos de sinais. Dentre elas será dado destaque à Transformada Discreta de Fourier e sua versão rápida, que foram trabalhadas neste projeto.

2.6.1 Transformada Fourier Discreta

De acordo com Antoniou (2006) e Proakis e Manolakis (1996), dado um sinal de duração finita discreto no tempo, um sinal periódico discreto correspondente pode ser gerado a partir da Transformada Discreta de Fourier (DFT, *Discret Fourier Transform*), representado como um espectro de frequência discreta. Assim, pode ser obtido um sinal espectral representado por uma sequência de números. Consequentemente a DFT é indicada para ser processada por computadores e dispositivos digitais.

Para uma função contínua de uma variável $f(t)$, a transformada de Fourier $F(f)$ é definida

por:

$$F(f) = \int_{-\infty}^{\infty} f(t)e^{-j2\Pi ft}dt \tag{2.8}$$

E sua inversa por:

$$f(t) = \int_{-\infty}^{\infty} F(f)e^{j2\Pi ft}df \tag{2.9}$$

Sendo $e^{j\Theta} = \cos(\Theta) + j\sin(\Theta)$ e $j = \sqrt{-1}$

Porém, para uma série de números complexos x(k) com N amostras, a transformada X(k) é definida por

$$X(n) = \sum_{k=0}^{N-1} x(k)W_N^{kn}, \quad n = 0,1...,N-1 \tag{2.10}$$

onde $W_N = e^{-j2\pi/N}$ conhecido como fator *twiddle*.

Podemos observar que para calcular $X(k)$ para um determinado valor de k, são necessárias N multiplicações complexas e $N-1$ somas complexas. Ou seja, são necessárias N^2 multiplicações complexas e $N(N-1)$ somas complexas para se obter os N valores da série transformada. Em termos de esforço computacional, a transformada discreta tem complexidade O(N2), o que a caracteriza como lenta e ineficaz para aplicações de alto desempenho (PROAKIS; MANOLAKIS, 1996).

2.6.2 Transformada Fourier Rápida

Como mencionado, a execução direta da DFT envolve N multiplicações complexas e $N-1$ somas complexas para cada valor de $X(k)$, e como temos N valores para serem determinados, são necessárias N^2 multiplicações e $N \times (N-1)$ adições. Consequentemente, para valores altos de N, a execução direta requer uma computação considerável. Contudo, a execução direta da DFT com base na sua definição envolve uma grande quantidade de cálculos redundantes e através de algumas estratégias inteligentes pode-se alcançar enormes reduções de complexidade de computação (ANTONIOU, 2006).

Para otimizar a execução da transformada discreta foi proposto por COOLEY e TUKEY (1965) um método eficiente para esse cálculo: a transformada rápida de Fourier ou FFT (*Fast Fourier Transform*). A FFT tem como ideia dividir a transformada discreta em transformadas menores recursivamente, afim de diminuir o esforço computacional. Existem diversos algoritmos para o cálculo da FFT. Os mais comuns são os *radix-2*, onde o número de amostras é uma potência de 2. Existem duas versões principais para calcular a FFT: a dizimação no tempo ou DIT (*Decimation-In-Time*) e a dizimação na frequência ou DIF (*Decimation-In-Frequency*) (MELLO, 2011).

2.6.3 Dizimação no tempo

Considerando que a DFT é definida por:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{-kn}, \quad \text{onde } W_N = e^{j2\pi/N} \quad \text{(fator } \textit{twiddle}\text{)}. \tag{2.11}$$

e assumindo que

$$N = 2^r, \quad \text{sendo } r \text{ um inteiro.} \tag{2.12}$$

A somatória acima pode ser dividida em duas partes, onde os elementos de índice par ficam na primeira parte e os elemento de índice ímpar na segunda, da seguinte maneira

$$X(k) = \sum_{\underbrace{n=0}_{\text{par}}}^{N-1} x(n) W_N^{-kn} + \sum_{\underbrace{n=1}_{\text{n ímpar}}}^{N-1} x(n) W_N^{-kn} \tag{2.13}$$

Ou, de forma alternativa

$$X(k) = \sum_{n=0}^{(N/2)-1} x_{10}(n) W_N^{-kn} + \sum_{n=0}^{(N/2)-1} x_{11}(n) W_N^{-kn} \tag{2.14}$$

sendo

$$\begin{aligned} x_{10}(n) &= x(2n) \\ x_{11}(n) &= x(2n+1) \end{aligned} \tag{2.15}$$

para $0 \leq n \leq N/2-1$, desde que

$$W_N^{-2kn} = e^{-j4kn\pi/N} = e^{-j2kn\pi/(N/2)} = W_{N/2}^{-kn} \tag{2.16}$$

Com isto, a equação pode ser escrita da seguinte maneira

$$X(k) = \sum_{n=0}^{(N/2)-1} x_{10}(n) W_{N/2}^{-kn} + W_N^{-k} \sum_{n=0}^{(N/2)-1} x_{11}(n) W_{N/2}^{-kn} \tag{2.17}$$

De forma mais simples tem-se

$$X(k) = X_{10}(k) + W_N^{-k} X_{11}(k) \tag{2.18}$$

E desde que $X_{10}(k)$ e $X_{11}(k)$ são periódicos, cada um com período $N/2$, então tem-se

$$X(k+N/2) = X_{10}(k+N/2) + W_N^{-(k+N/2)} X_{11}(k+(N/2)) = X_{10}(k) + W_N^{-k} X_{11}(k) \tag{2.19}$$

Esta equação pode ser representa por um gráfico de fluxo de sinal, como na Figura 2.16 (ANTONIOU, 2006).

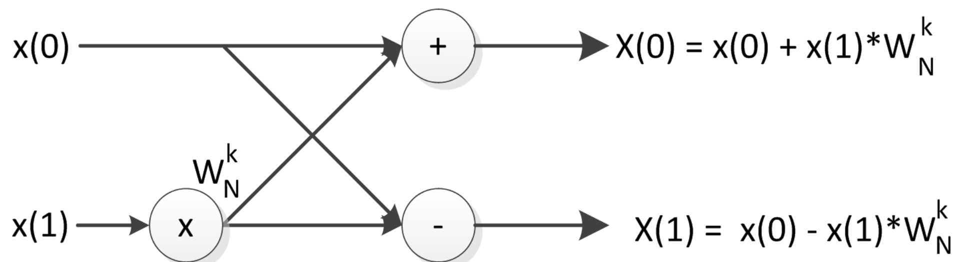


Figura 2.16 - Operação borboleta.
Fonte: Adaptado de Antoniou (2006)

Para aplicar o algoritmo de dizimação no tempo a entrada precisa ser reordenada para a ordem *bit*-invertido de índices (ver Tabela 2.2), onde os elementos são separados em dois grupos de acordo com o *bit* menos significativo na primeira separação, com o segundo *bit* menos significativo na segunda separação e assim sucessivamente. Depois de reordenada, a série passa por $\log_2 N$ etapas, onde $N/2$ borboletas são aplicadas, conforme a Figura 2.17, onde o algoritmo é aplicado a uma série de 8 amostras (ANTONIOU, 2006; PROAKIS; MANOLAKIS, 1996).

Tabela 2.2 - Tabela de ordenação *bit*-reverso.

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

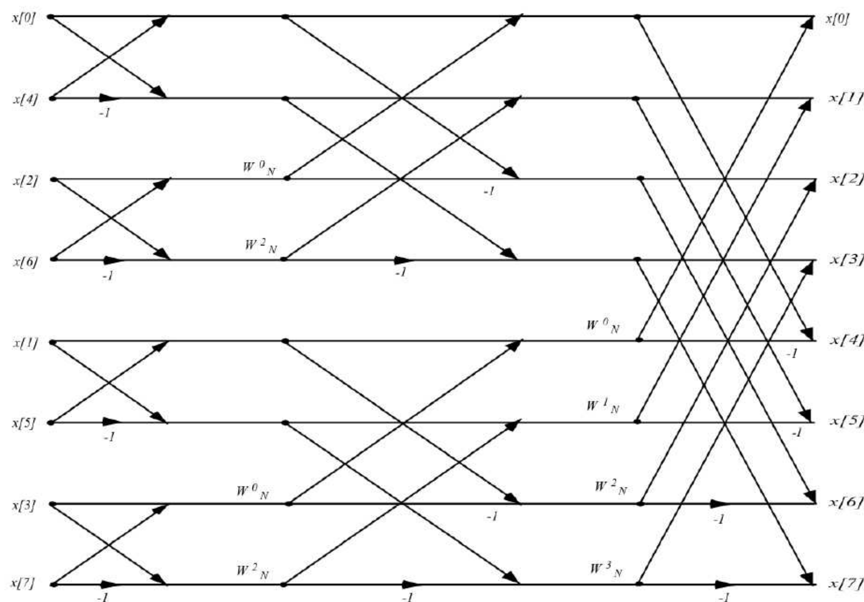


Figura 2.17 - Etapas do algoritmo FFT DIT.
Fonte: Adaptado de Antoniou (2006).

No exemplo descrito, N é um valor potência de 2. No entanto, o algoritmo pode ser aplicado a qualquer conjunto finito, apenas incluindo elementos de valor zero nos dados de entrada. Incluindo zeros aos dados de entrada, melhora-se a resolução do espectro de frequência do sinal discreto (ANTONIOU, 2006).

O algoritmo pode ser facilmente programado utilizando um único vetor de dados. Como pode ser observado na Figura 2.17, uma vez que as saídas de cada borboleta de entrada são computados, os elementos de entrada não são mais necessários para posterior processamento e pode ser substituído pelas saídas correspondentes. Quando procedemos da mesma maneira da esquerda para a direita, no final do cálculo a matriz de entrada irá conter os elementos da DFT calculada devidamente ordenada. Os elementos de entrada podem ser inseridos nos locais corretos da matriz usando uma subrotina de reordenação simples. Em geral, cada ciclo do algoritmo envolve $N/2$ borboletas e cada borboleta exige uma multiplicação de números complexos. Uma vez que existem r ciclos de

computação e $r = \log_2 N$, o número total de multiplicações é $(N/2)\log_2 N$ em oposição à N^2 no caso da execução direta da DFT. Isso constitui uma enorme redução em computação. Por exemplo, se $N = 512$, então o número de multiplicações é reduzido a uma fração de aproximadamente um por cento (2304 multiplicações) do que é exigido pela execução direta (262144 multiplicações) (ANTONIOU, 2006).

2.6.4 Dizimação na frequência

Na dizimação no tempo (DIT), o sinal no tempo é separado em duas partes, uma com os elementos de índice par e outra com os elementos de índice impar. O mesmo procedimento é então aplicado em cada nova sequência de dados até que uma sequência de um único elemento seja alcançada.Já o algoritmo da dizimação em frequência pode ser desenvolvido separando a sequência dos dados na metade, e então repetindo este processo até que alcance uma sequência de um único elemento. Isso pode ser escrito da seguinte forma

$$X(k) = \sum_{n=0}^{(N/2)-1} x(n)W_N^{-kn} + \sum_{n=N/2}^{N-1} x(n)W_N^{-kn} \tag{2.20}$$

$$= \sum_{n=0}^{(N/2)-1} [x(n) + W_N^{-kN/2} x(n + \frac{N}{2})]W_N^{-kn} \tag{2.21}$$

E substituindo o primeiro k por 2k e outro por 2k + 1 obtém-se

$$X(2k) = \sum_{n=0}^{(N/2)-1} x_{10}(n)W_{N/2}^{-kn} = X_{10}(k) \tag{2.22}$$

$$X(2k+1) = \sum_{n=0}^{(N/2)-1} x_{11}(n)W_{N/2}^{-kn} = X_{11}(k) \tag{2.23}$$

sendo

$$X_{10}(n) = x(n) + x(n + N/2) \tag{2.24}$$

$$X_{11}(n) = [x(n) - x(n + N/2)]W_N^{-n} \tag{2.25}$$

para $0 \leq n \leq N/2 - 1$. Então os índices ímpares e pares de $X(k)$ são dados pela DFT de $x_{10}(n)$ e $x_{11}(n)$ respectivamente. Assumindo que os valores de $x(n)$ estão armazenados sequencialmente em um vetor, os valores de $x_{10}(n)$ e $x_{11}(n)$ podem ser computados como ilustrado na Figura 2.18. A Figura 2.19 apresenta o fluxo de dados deste algoritmo para um sinal de 8 pontos.

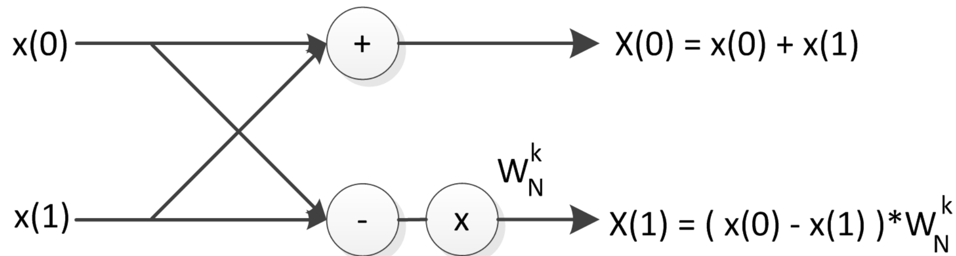


Figura 2.18 - Operação borboleta da dizimação em frequência.

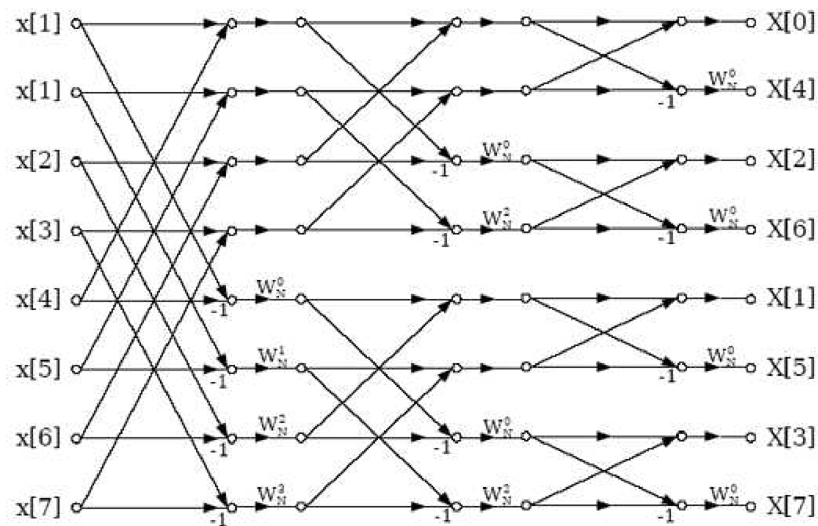


Figura 2.19 - Etapas do algoritmo FFT DIF para um sinal de 8 pontos.

Assim como no algoritmo de dizimação no tempo, o procedimento termina no último estágio, onde cada sequência possui apenas um elemento, dando o resultado da DFT, porém ordenados segundo o algoritmo "Bit-reverse", como pode ser visto na Figura 2.19. A grande vantagem deste algoritmo é que o sinal no tempo está ordenado em sua forma natural, sequencialmente. Isso torna este algoritmo ideal para processamento serial.

2.6.5 Algoritmos seriais para a FFT

Segundo Li (2003), existem muitas maneiras de realizar a transformada rápida de Fourier. Existem processadores específicos que só realizam esta tarefa, assim como existem processadores programáveis, como é o caso de DSPs (Processadores Digitais de Sinais). A arquitetura de um processador específico para a FFT é portanto otimizado com relação a estrutura de memória, unidades de controle e elementos de processamento.

Existem principalmente três tipos de processamentos: totalmente paralelo, processadores de colunas e processadores *pipeline*. Todos os três tipos de processamentos representam um mapeamento diferente do gráfico de fluxo de sinal da FFT para as estruturas de *hardware*.

A estrutura de *hardware* em um processador de FFT totalmente paralelo é um mapeamento direto do gráfico de fluxo de sinal apresentado na Figura 2.17. Desta forma, uma implementação de uma transformada de 8 pontos totalmente paralela requer 24 somadores complexos e 5 multiplicadores complexos, quando otimizados matematicamente. Isso exige muitos elementos de *hardware* e consequentemente não há otimização alguma (LI, 2003).

Para reduzir a complexidade de *hardware*, pode-se utilizar um processamento por colunas ou por *pipeline*. Em um processamento por colunas, um conjunto de elementos (uma coluna de operadores borboleta) calculam uma etapa de cada vez. Os resultados deste calculo são retornados ao mesmo conjunto de elementos do processo para calcular a próxima fase. Para transformações de grande número de elementos, o roteamento dos resultados é complexo e difícil (LI, 2003).

Para um processamento *pipeline*, cada estágio tem seu próprio conjunto de operadores. Todos os estágios são computados assim que os dados estão disponíveis. Este tipo de processamento tem características como simplicidade, modularidade e alto rendimento em termos de números de dados calculados por segundo. Estas características são importantes para tempo real, em aplicações em que a entrada dos dados no sistema é serializada, na ordem natural (LI, 2003).

Os métodos mais comuns para o processamento em *pipeline* são:

- *Radix-2 multipath delay commutator* (R2MDC)
- *Radix-2 single-path delay feedback* (R2SDC)
- *Radix-4 multipath delay commutator* (R4MDC)
- *Radix-4 single-path delay commutator* (R4SDC)

- *Radix-4 single-path delay feedback* (R4SDF)
- *Radix*2^2 *single-path delay commutator* (R2^2SDC)

A arquitetura *Radix-2 Multipath Delay Commutator* é a abordagem mais simples para implementar o algoritmo FFT usando uma arquitetura *pipeline*. A Figura 2.20 mostra um exemplo com 8 pontos. O Apêndice A apresenta o funcionamento do algoritmo em detalhes, com todas as estruturas de *hardware* e uma simulação para 8 pontos (RABINER; GOLD, 1975; JACKSON et al., 2004).

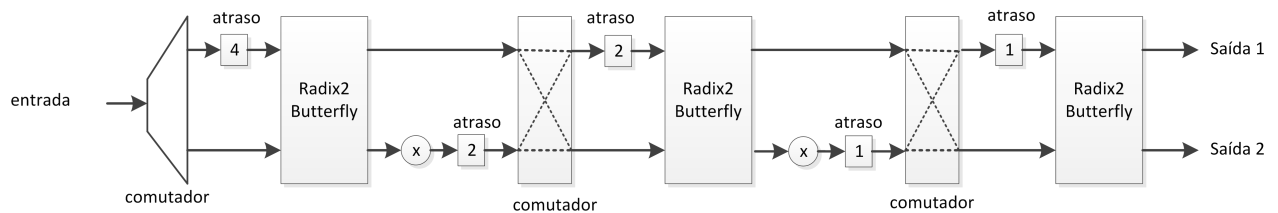


Figura 2.20 - Exemplo do algoritmo R2MDC com 8 pontos.
Fonte: Adaptado de Rabiner e Gold (1975).

3 DESENVOLVIMENTO DA ARQUITETURA

Este capítulo apresenta uma proposta de arquitetura de processamento de sinais em FPGA. Esta arquitetura permite realizar a aquisição dos dados e sua transmissão pela rede local, possibilitando realizar o processamento dos sinais em um FPGA utilizando seus recursos de *hardware* para otimizar o processo. Na primeira seção será apresentada uma visão geral da arquitetura de processamento de sinais, o sistema de aquisição com a presença dos dispositivos envolvidos e suas respectivas funções, e o sistema de processamento com o processador desenvolvido no FPGA e uma visão geral sobre o *software* desenvolvido para gerência das tarefas. Na segunda seção será apresentado o algoritmo desenvolvido em *hardware* para executar a transformada rápida de Fourier e sua integração com o processador NIOS2.

3.1 Visão geral do sistema

A proposta desta arquitetura se baseia em um sistema de aquisição de sinais através de plataformas de *hardware* espalhadas em campo, com transmissão destes dados através de uma rede local para um FPGA, que irá processá-los usando o algoritmo da FFT implementado em *hardware*, e mostrar o resultado de forma gráfica numa tela de LCD. Os dados processados podem ainda ser enviados a um computador pessoal para serem analisados. É possível monitorar diversos tipos de sinais, tanto no domínio do tempo quanto no domínio da frequência. A Figura 3.1 apresenta a sequência de funcionamento do sistema como um todo, desde a aquisição dos dados até a exibição em ambos os domínios. Nesta figura, a primeira etapa consiste da aquisição dos sinais que em seguida, são enviados pela rede local através de um roteador simples para um sistema de processamento de sinais em FPGA. Logo após, os dados são apresentados na tela de LCD em ambos os domínios.

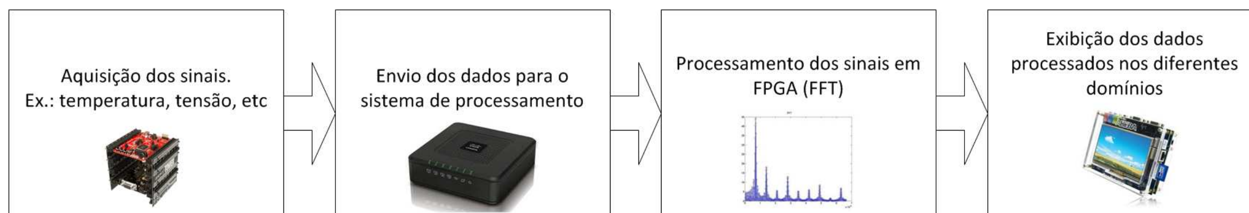


Figura 3.1 - Funcionamento do sistema: da aquisição à visualização.

A arquitetura proposta é composta de plataformas de *hardware* da FreeScale Semiconductors® (kit TWR-MCF51CN128, chamado de TOWER) e da Altera® (NIOS *Embbeded Evaluation Kit*, chamado de NEEK), um computador pessoal, e de um roteador para realizar as conexões de rede. Os TOWERs ficaram responsáveis pela aquisição dos dados e sua transmissão pela rede, enquanto que o NEEK ficou responsável por solicitar e receber estes dados, processá-los e apresentá-los no LCD. O computador pessoal é capaz de se comunicar com o NEEK e receber os dados que estão sendo apresentados na tela. A Figura 3.2 apresenta uma visão geral das conexões entre os elementos existentes.

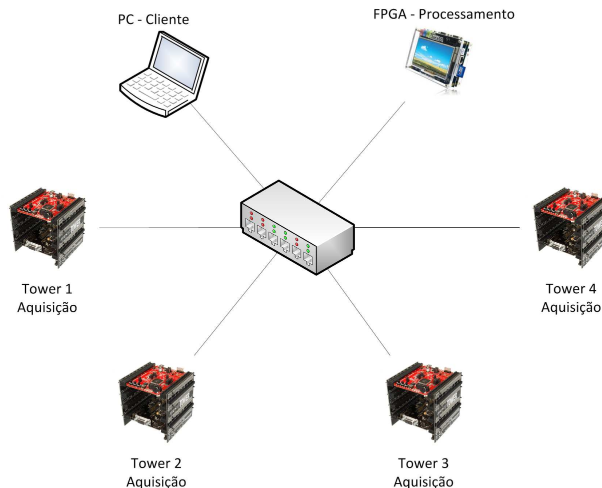


Figura 3.2 - Estrutura do sistema de aquisição em rede.

Para criar essa arquitetura foi necessário desenvolver os *softwares* que foram executados em ambas as plataformas (TOWER e NEEK), e o *hardware* que foi sintetizado no FPGA (NEEK). O TOWER foi chamado de módulo de aquisição, enquanto que o NEEK foi chamado de módulo de processamento de dados. No TOWER, o *software* deve inicializar a sua comunicação em rede e realizar consecutivas aquisições de um canal de conversão analógico-digital, armazenando os resul-

tados na memória enquanto aguarda um comando para enviar os dados. No NEEK foi desenvolvido o processador, o módulo FFT (como um periférico do processador) e um programa para controlar todo o sistema. Foi elaborado um programa de gerenciamento de telas para que vários pequenos programas fossem executados de forma concorrente, mantendo a liberdade do usuário utilizar o NEEK para cancelar ou trocar o tipo de processamento, por exemplo. Também foi implementada a comunicação com os TOWERs e os algoritmos da DFT e FFT em *software*. O sistema de gerenciamento de telas facilita a criação de objetos, como botões, gráficos, textos, caixas e linhas, e também facilita a criação de eventos ou ações associadas aos objetos. No computador pessoal foi utilizado o *software* Matlab para receber os dados no domínio do tempo e da frequência. Foi feita uma comparação dos resultados apresentados pelo processamento em FPGA com relação ao mesmo processamento com o Matlab®, com o objetivo de validar a execução dos módulos desenvolvidos.

A Figura 3.3 mostra o diagrama de fluxo de dados do sistema. O TOWER realiza a aquisição dos dados e aguarda um comando solicitando os dados. Ao receber este comando, ele envia os dados armazenados para o NEEK através da rede local. O NEEK processa e exibe os dados em sua tela de LCD. Além disto, o NEEK aguarda um comando de solicitação de dados processados. Ao receber este comando, o NEEK envia os dados processados. Foi utilizado um computador pessoal com o *software* Matlab® para solicitar os dados processados ao NEEK.

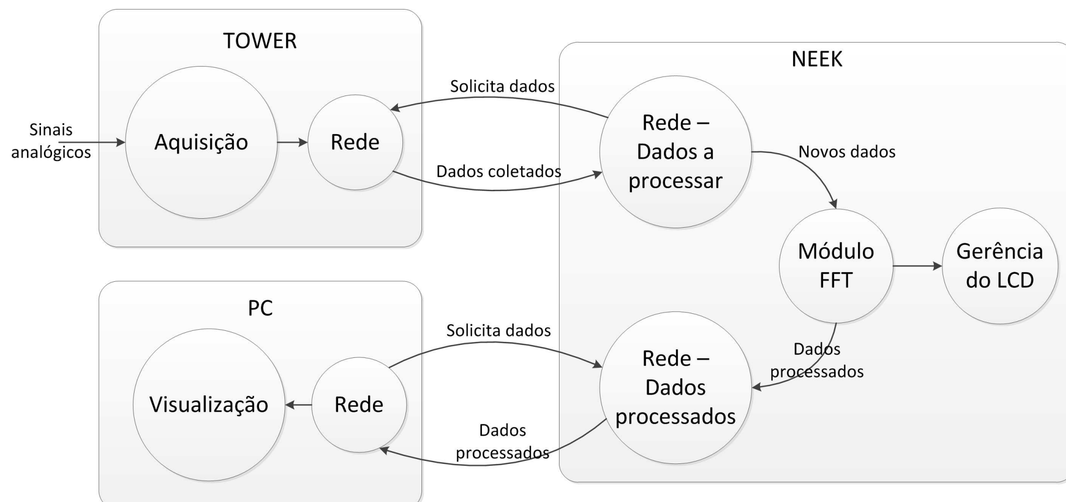


Figura 3.3 - Diagrama de fluxo de dados do sistema.

3.1.1 Aquisição de dados

A aquisição dos dados é realizada através de vários kits de desenvolvimento TOWER distribuídos em uma rede local. O TOWER tem a finalidade de realizar a aquisição de um sinal pelo conversor analógico-digital e armazenar os dados na memória, que posteriormente serão enviados pela rede local para o dispositivo que solicitar os dados, seja ele o computador pessoal presente na rede ou o FPGA. Foi especificado que o TOWER deve ser configurado pela rede TCP/IP através de comandos para configurar a taxa de aquisição, a quantidade de pontos a serem enviados, o canal de origem do sinal e o nível de tensão de disparo da aquisição.

O sistema foi modelado utilizando a metodologia SDL (*Specification and Description Language*). Nesta metodologia, a primeira etapa é criar um diagrama de partição, onde as funções do sistema são identificadas e divididas em partes mais simples, até que se chegue em um nível de complexidade suficientemente pequeno para solução de cada parte do problema. Neste caso, o módulo foi dividido em entrada, com o conversor analógico-digital, comunicação, com a rede local, aplicação, com a gerência da comunicação em rede e o controle do conversor analógico-digital, e por fim nas interrupções do temporizador e do conversor analógico-digital. Desta maneira, foi gerado o diagrama de partição da Figura 3.4.

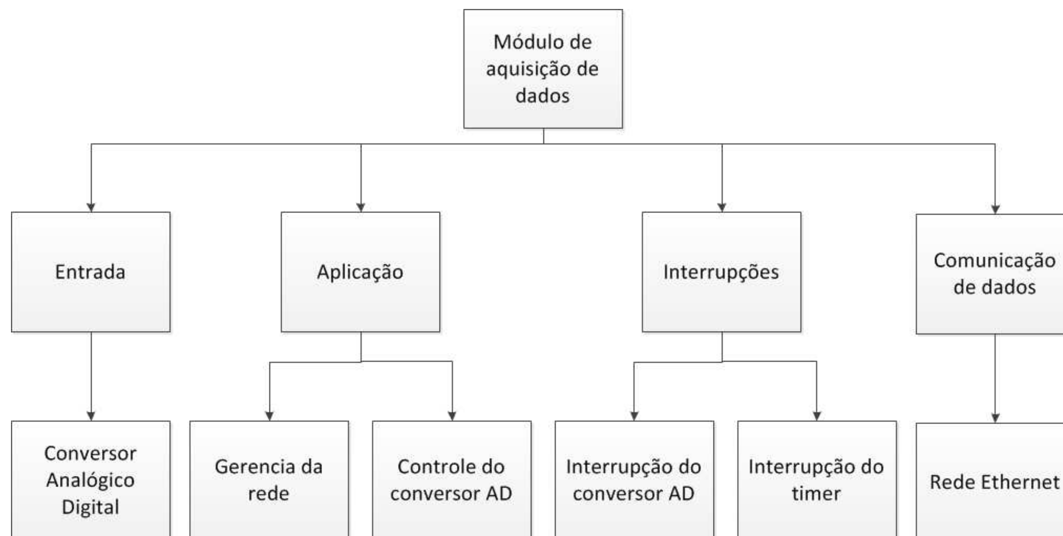


Figura 3.4 - Diagrama de partição do módulo de aquisição de dados.

A partir deste diagrama foi criado o diagrama que apresenta a interação entre as funções

do sistema. Geralmente, cada função do sistema se torna uma tarefa que deverá ser executada de forma concorrente. Neste caso, o escalonamento entre as tarefas está sendo gerenciado pelo sistema operacional de tempo real MQX, fornecido pelo fabricante do microcontrolador. O diagrama de interação da Figura 3.5 apresenta a interação entre os elementos do sistema.

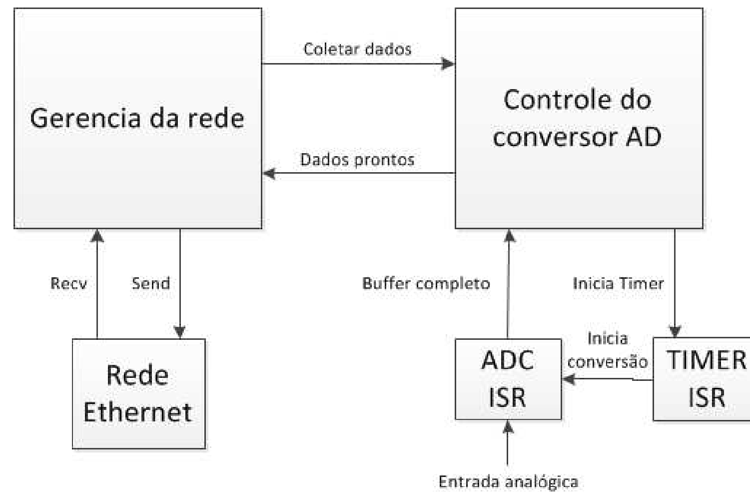


Figura 3.5 - Diagrama de interação do módulo de aquisição de dados.

Diante do exposto, para esse módulo foram desenvolvidas duas tarefas principais: uma que realiza a conversão dos dados e armazena os resultados, e outra que recebe e interpreta comandos pela rede. Estas tarefas comunicam-se através de recursos de sincronização do sistema operacional utilizado, o MQX. Ao receber um comando pela rede, a tarefa de gerência da rede identifica o comando. Caso o comando seja para retornar os dados da aquisição, esta envia um sinal para a tarefa de conversão AD. A tarefa de conversão AD, que antes estava aguardando um comando, verifica se o conjunto de dados já está completo, e caso positivo, retorna um sinal para a tarefa da rede, indicando que esta pode dar início ao envio dos dados.

A tarefa de conversão AD instala as interrupções do ADC (*Analog to Digital Converter*) e do *Timer*. Optou-se por não utilizar as rotinas do sistema operacional para a aquisição dos dados pois estavam lentas devido ao *overhead* de operações envolvidas quando utilizamos o MQX. Assim, toda vez que estas interrupções são chamadas, são tratadas ininterruptamente, não comprometendo a integridade do sinal adquirido e tampouco a sua taxa de amostragem, que deve ser exatamente a mesma entre todos os pontos adquiridos. Durante a aquisição, a interrupção do *timer* ocorre a cada 20 microssegundos, iniciando a conversão de cada ponto. O período de interrupção pode ser configurado. Ao término da conversão de cada novo ponto, uma outra interrupção é gerada e ativa

uma rotina de tratamento que armazena o dado na memória RAM. A cada pacote de *N* pontos adquirido, um sinal é enviado à função de controle da conversão analógico-digital para que ela pare as conversões e avise a tarefa da rede que um novo conjunto de dados está pronto para ser enviado. Desta maneira, essa tarefa possui 2 estados. No primeiro estado ela aguarda um evento da tarefa que monitora a rede, dando início às aquisições, indo para o próximo estado. Ao terminar as *N* conversões, esta função avisa a função da rede que o dado está pronto para ser enviado. A Figura 3.6 apresenta o diagrama de estados da tarefa de conversão AD.

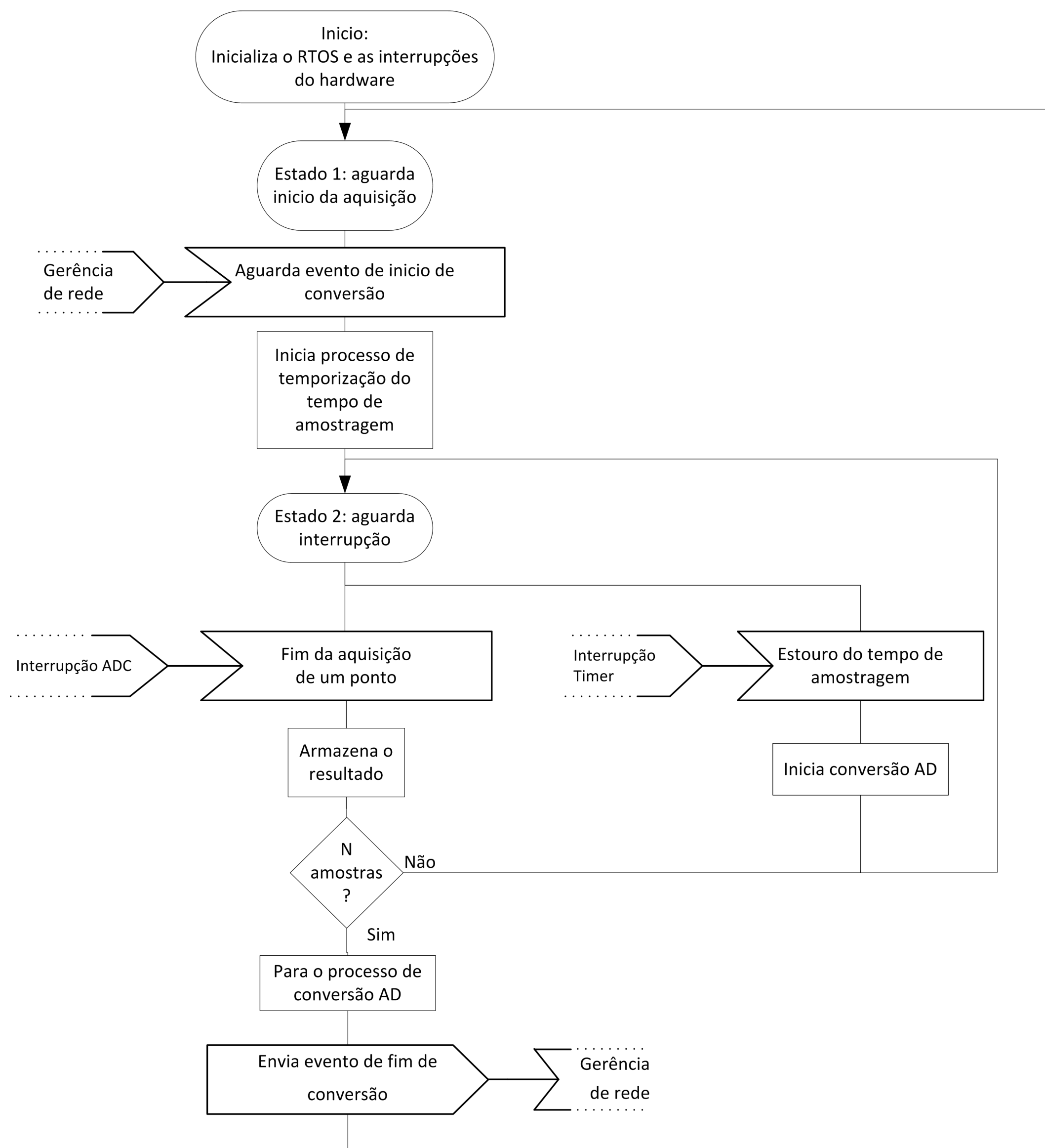


Figura 3.6 - Diagrama de estados SDL da tarefa de controle do conversor AD.

A tarefa de monitoramento da rede inicializa a pilha de comunicação via rede do sistema

operacional, o que significa inicializar todo o *hardware* e preparar as camadas mais baixas dos protocolos de rede. Estes comandos fazem parte do pacote RTCS (*Real Time Communication Suite*) fornecido com o sistema operacional MQX. Após realizar estas inicializações, o sistema dá inicio à comunicação por *sockets* utilizando o protocolo UDP. Os comandos recebidos obedecem ao formato apresentado na Figura 3.7, enquanto o pacote enviado como resposta segue o formato apresentado na Figura 3.8.

Figura 3.7 - Formato do comando enviado do NEEK para o TOWER.

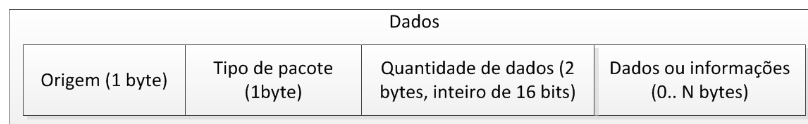


Figura 3.8 - Formato da resposta enviada do TOWER ao NEEK.

Esta tarefa possui vários estados para que a aplicação possa responder a comandos provenientes da rede. Ela começa no estado *starting*, onde inicializa o *socket*, depois vai para o estado *selecting*, onde espera até ocorrer algum recebimento de dados pela rede e, quando isto acontece ela recebe o ponteiro para o *socket* que foi aberto; em seguida passa para o estado *receiving* onde recebe o pacote de dados; e então vai para o estado *processing*, que é o interpretador de comandos. Caso o comando seja para enviar os dados da conversão AD, esta tarefa entra no estado *sending-graph*, onde envia um evento para a tarefa de conversão AD e espera por um evento indicando que os dados estão prontos. Feito isto, a tarefa envia os dados para o endereço que requisitou na próxima porta, para evitar congestionamento de pacotes, e volta para o estado *selecting*. A Figura 3.9 apresenta o diagrama de estados da tarefa que monitora a rede.

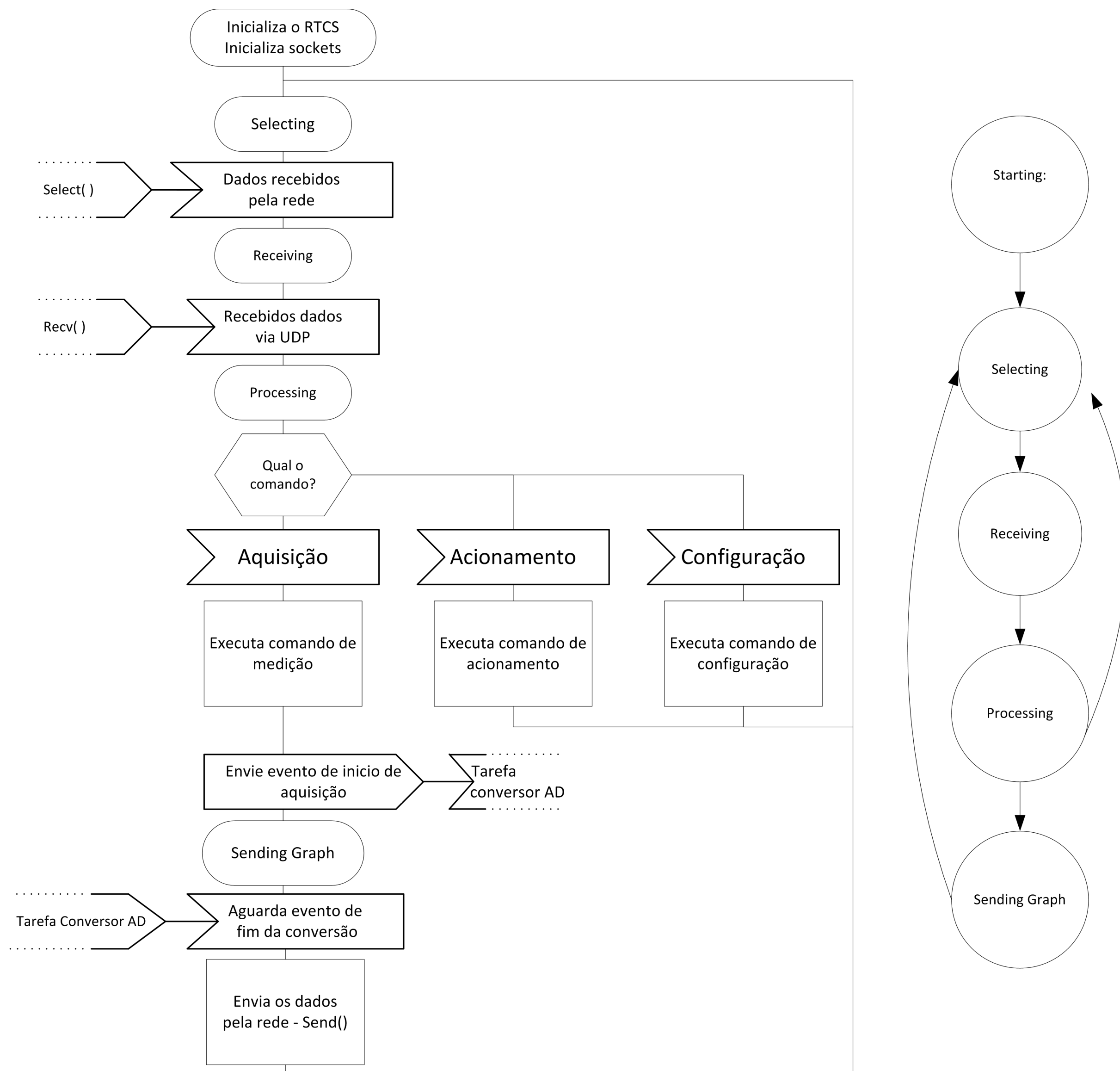


Figura 3.9 - Diagrama de estados SDL da tarefa de monitoramento da rede e diagrama de estados simplificados.

Desta maneira, pode-se criar um diagrama de sequência, apresentando o funcionamento das tarefas concorrentes de acordo com a linha do tempo. Durante todo o processo, o temporizador está ativo e gerando interrupções. A cada interrupção é iniciada uma nova conversão do sinal analógico. Ao término da conversão, uma segunda interrupção é chamada. Esta armazena o ponto em um *buffer* e avisa a tarefa de controle da conversão analógico-digital de que o *buffer* está completo. Em algum momento do tempo o NEEK envia um comando para o TOWER. Este identifica o comando e o executa. Os comandos existentes são de acionamento de portas de saída (utilizado apenas para validação da comunicação), de configuração da conversão analógico-digital, e de retornar um pa-

cote de dados. Caso o comando seja de retornar um pacote de dados, esta tarefa envia um sinal para a tarefa de controle da conversão AD. Esta aguarda que o *buffer* esteja completo e, quando isto ocorre, envia um sinal de volta para a tarefa da rede avisando que esta já pode enviar o pacote de dados de volta para o NEEK. Esta sequência de atividades pode ser vista na Figura 3.10.

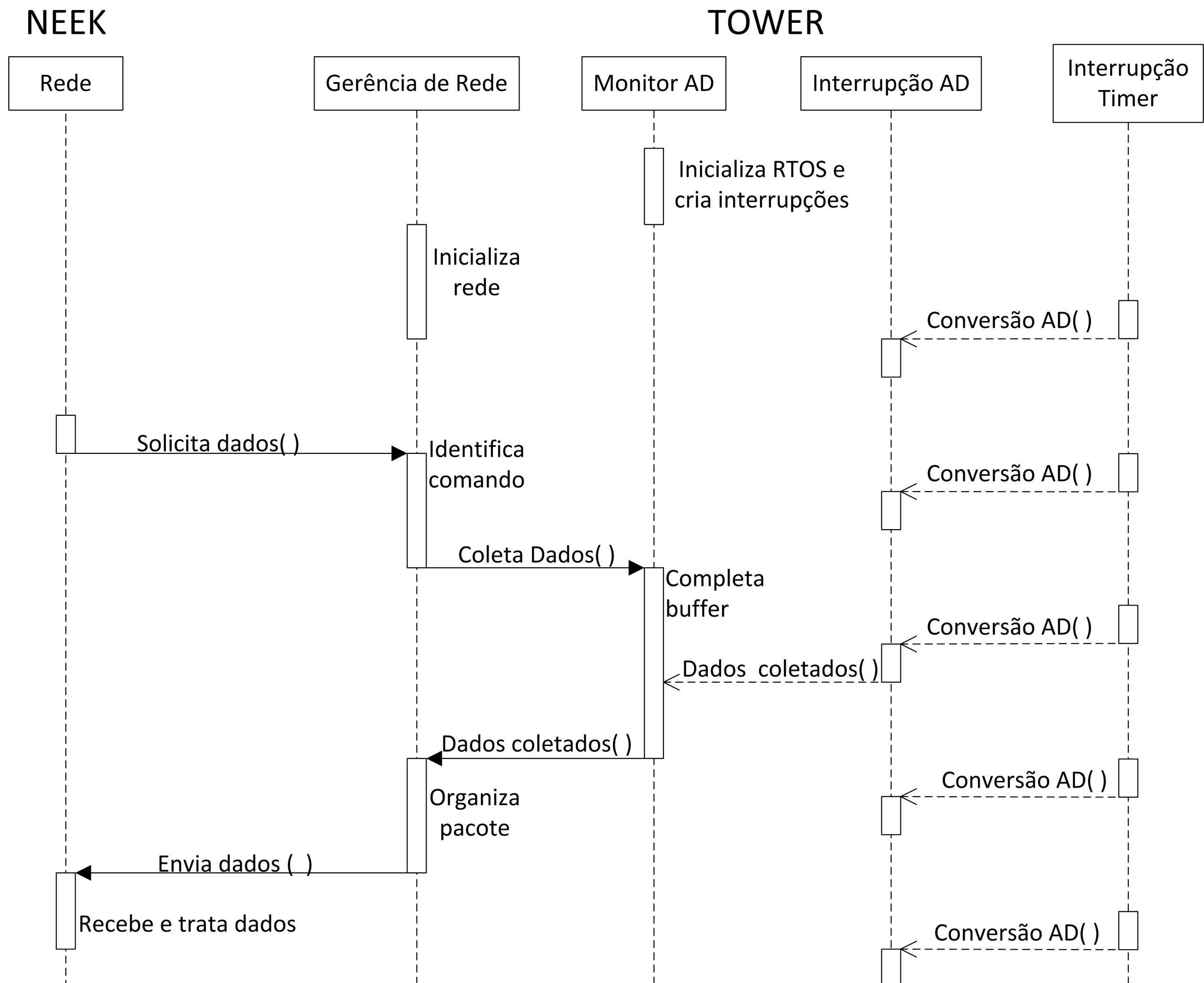


Figura 3.10 - Diagrama de sequência do módulo de aquisição de dados.

3.1.2 Processamento de sinais usando FPGA

O sistema implementado tem como requisito a capacidade de comunicar com diversos TOWERS em uma rede local via protocolo UDP, enviando comandos de configuração e de ação,

e recebendo as informações de volta como dados de uma aquisição de um sinal no tempo. Além disto, ao receber os dados da aquisição realizada pelo TOWER no domínio do tempo, o NEEK deve apresentá-los de forma gráfica em sua tela de LCD, processá-los (aplicar transformadas, filtros, convoluções) e exibir os resultados de forma gráfica. Por fim, quando solicitado, o NEEK deve enviar pela rede os dados processados para um computador pessoal através do protocolo UDP.

A Figura 3.11 apresenta o processador desenvolvido no FPGA de forma simplificada. Observa-se na figura que existe um processador chamado de CPU. Existe um controlador de rede, formado por um conjunto de módulos que controlam internamente toda a comunicação pela rede local. Existe um controlador para o LCD, também composto por um conjunto de módulos que controlam o acesso à memória e o envio dos dados de acordo com o formato exigido pelo LCD. O controlador de memória é responsável por gerar os sinais para a memória SDRAM externa. O módulo FFT foi desenvolvido para realizar a transformada rápida de Fourier em *hardware*.

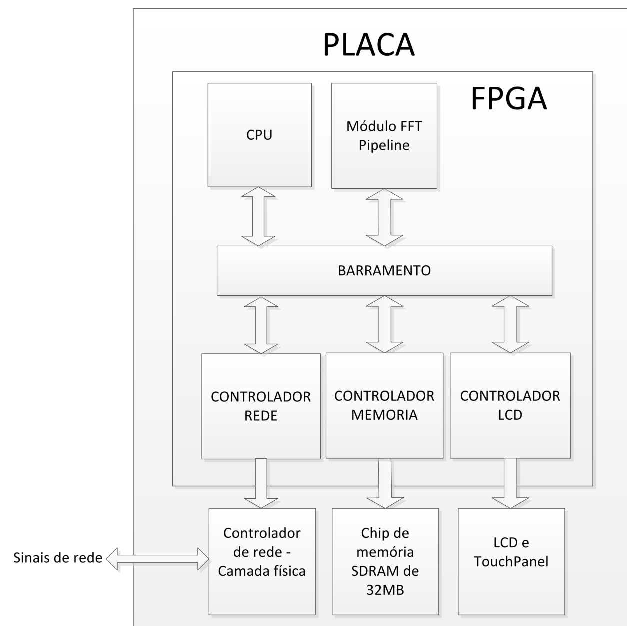


Figura 3.11 - Placa do NEEK com o processador desenvolvido e os periféricos externos ao FPGA de forma simplificada.

Depois que o *hardware* desenvolvido foi sintetizado no FPGA, partiu-se para o desenvolvimento do *software*. O *software* que foi desenvolvido para o NIOS 2 foi dividido em 2 módulos distintos, comunicação e gerenciamento de telas, que interagem entre si e concorrem pelo processador. As funções de processamento de sinais são chamadas pelo módulo de gerenciamento de telas. O *software* foi desenvolvido utilizando o sistema operacional de tempo real uC/OS2 que no ambi-

ente de programação é integrado ao *Interniche Ethernet Stack*, uma biblioteca para comunicação pela rede local. A Figura 3.12 apresenta uma visão em blocos do *software* desenvolvido no NIOS.

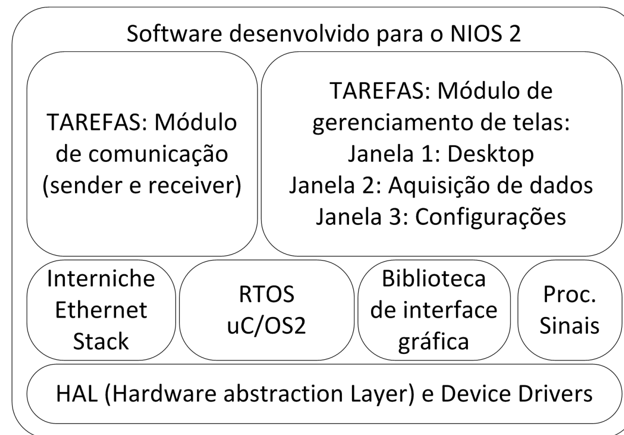


Figura 3.12 - Organização do *software* desenvolvido.

Os blocos *Interniche Ethernet Stack*, RTOS uC/OS2 e HAL são oferecidos gratuitamente pela Altera®. Já a biblioteca de interface gráfica, de processamento de sinais em *software*, o módulo de comunicação e o módulo de gerenciamento de janelas foram desenvolvidos durante o trabalho.

NEEK: Processador desenvolvido

O processador desenvolvido foi projetado para controlar todos os periféricos necessários ao funcionamento do sistema. Utilizando o SoPC *Builder*, os elementos presentes no processador foram instanciados um a um e conectados de forma coerente. A Figura 3.13 apresenta a janela de criação do processador.

Use	Connections	Module	Description	Clock	Base	End	IRQ
☑		⊟ **cpuNios**	Nios II Processor	[clk]			
		instruction_master	Avalon Memory Mapped Master	**clk100MHz**			
		data_master	Avalon Memory Mapped Master	[clk]	IRQ 0	IRQ 31	
		jtag_debug_module	Avalon Memory Mapped Slave	[clk]	0x0c004800	0x0c004fff	
☑		⊞ **fft_pipeline_avalon_0**	fft_pipeline_avalon	**clk100MHz**	0x0c000000	0x0c003fff	
☑		⊞ **sgdma_rx**	Scatter-Gather DMA Controller	**clk100MHz**	0x0c005480	0x0c0054bf	1
☑		⊞ **sgdma_tx**	Scatter-Gather DMA Controller	**clk100MHz**	0x0c0054c0	0x0c0054ff	3
☑		⊞ **tse_mac**	Triple-Speed Ethernet	***multiple***	0x0c005000	0x0c0053ff	
☑		⊞ **tse_ddr_clock_bridge**	Avalon-MM Clock Crossing Bridge	***multiple***	0x00000000	0x01ffffff	
☑		⊞ **sdram**	DDR SDRAM Controller with ALTMEMPHY	**clk50Mhz**	0x00000000	0x01ffffff	
☑		⊟ **cpu_ddr_clock_bridge**	Avalon-MM Clock Crossing Bridge	[clk_s1]			
		s1	Avalon Memory Mapped Slave	**clk100MHz**	0x00000000	0x03ffffff	
		m1	Avalon Memory Mapped Master	**sdram_sysclk**			
☑		⊟ **lcd_sgdma**	Scatter-Gather DMA Controller	[clk]			
		csr	Avalon Memory Mapped Slave	**sdram_sysclk**	0x02000000	0x0200003f	2
		descriptor_read	Avalon Memory Mapped Master	[clk]			
		descriptor_write	Avalon Memory Mapped Master	[clk]			
		m_read	Avalon Memory Mapped Master	[clk]			
		out	Avalon Streaming Source	[clk]			
☑		⊟ **lcd_ta_sgdma_to_fifo**	Avalon-ST Timing Adapter	[clk]			
		in	Avalon Streaming Sink	**sdram_sysclk**			
		out	Avalon Streaming Source	[clk]			
☑		⊟ **lcd_pixel_fifo**	On-Chip FIFO Memory	[clk_in]			
		in	Avalon Streaming Sink	**sdram_sysclk**			
		out	Avalon Streaming Source	**clk100MHz**			
☑		⊟ **lcd_ta_fifo_to_dfa**	Avalon-ST Timing Adapter	[clk]			
		in	Avalon Streaming Sink	**clk100MHz**			
		out	Avalon Streaming Source	[clk]			
☑		⊟ **lcd_64_to_32_bits_dfa**	Avalon-ST Data Format Adapter	[clk]			
		in	Avalon Streaming Sink	**clk100MHz**			
		out	Avalon Streaming Source	[clk]			
☑		⊟ **lcd_pixel_converter**	Pixel Converter (BGR0 --> BGR)	[clk]			
		in	Avalon Streaming Sink	**clk100MHz**			
		out	Avalon Streaming Source	[clk]			
☑		⊟ **lcd_24_to_8_bits_dfa**	Avalon-ST Data Format Adapter	[clk]			
		in	Avalon Streaming Sink	**clk100MHz**			
		out	Avalon Streaming Source	[clk]			
☑		⊟ **lcd_sync_generator**	Video Sync Generator	[clk]			
		in	Avalon Streaming Sink	**clk100MHz**			
☑		⊟ **lcd_i2c_scl**	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	**clk100MHz**	0x0c005500	0x0c00551f	
☑		⊟ **lcd_i2c_en**	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	**clk100MHz**	0x0c005520	0x0c00553f	
☑		⊟ **lcd_i2c_sdat**	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	**clk100MHz**	0x0c005540	0x0c00555f	
☑		⊟ **sys_clk_timer**	Interval Timer	[clk]			
		s1	Avalon Memory Mapped Slave	**clk100MHz**	0x0c005400	0x0c00543f	8
☑		⊟ **sysid**	System ID Peripheral	[clk]			
		control_slave	Avalon Memory Mapped Slave	**clk100MHz**	0x0c005570	0x0c00557f	
☑		⊟ **jtag_uart**	JTAG UART	[clk]			
		avalon_jtag_slave	Avalon Memory Mapped Slave	**clk100MHz**	0x0c005560	0x0c00556f	0
☑		⊟ **performance_counter**	Performance Counter Unit	[clk]			
		control_slave	Avalon Memory Mapped Slave	**clk100MHz**	0x0c005440	0x0c00547f	

Figura 3.13 - Janela do SoPC Builder com o processador construído.

Embora o SoPC *Builder* seja simples de utilizar, sua janela de representação do resultado não é intuitiva quanto a visão geral do processador desenvolvido. Por isto, foi criada a Figura 3.14 que representa em diagrama de blocos o processador e sua conexão com os demais periféricos.

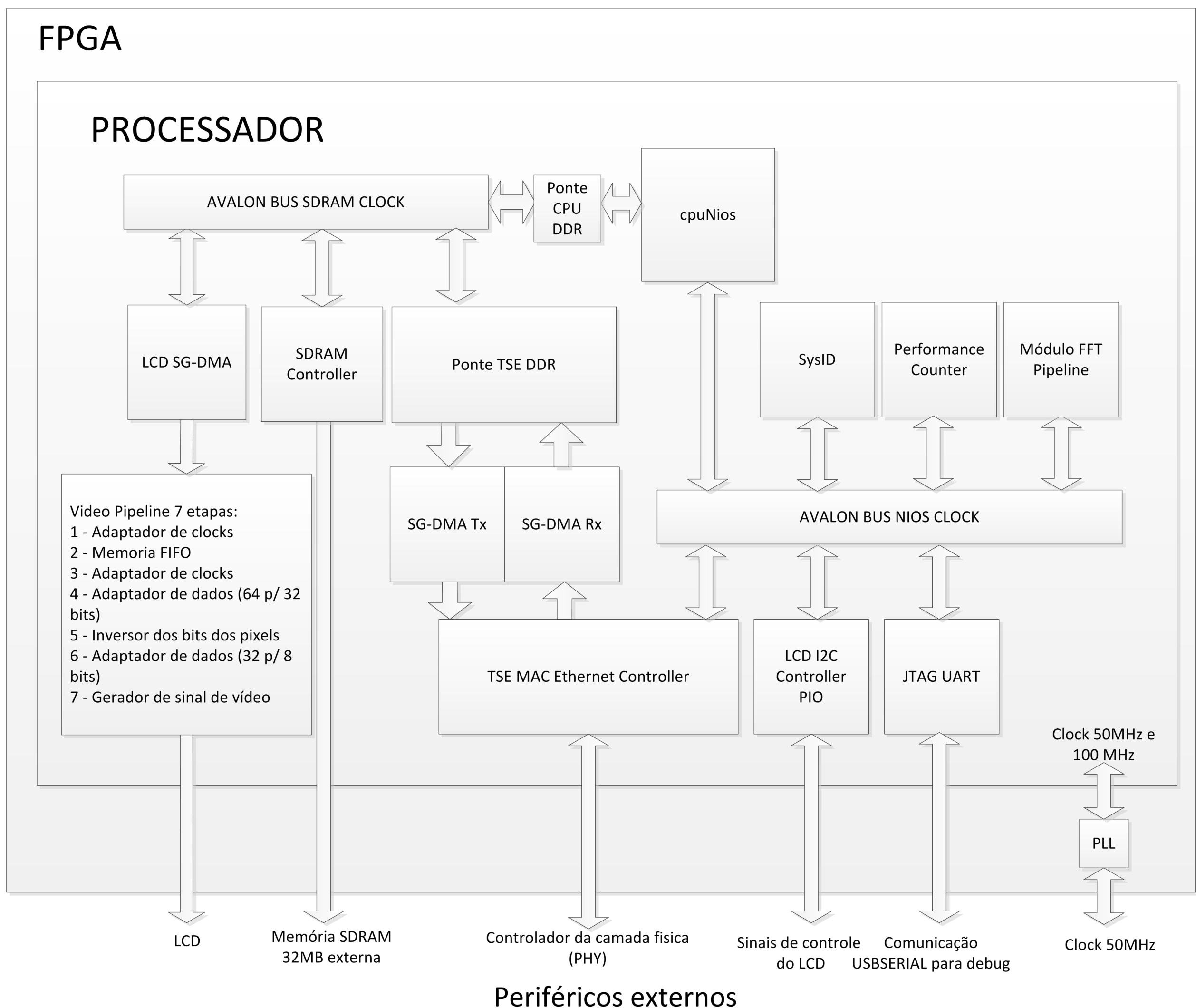


Figura 3.14 - Diagrama de blocos do processador desenvolvido.

Observa-se que foram utilizados vários periféricos. O barramento Avalon é o meio pelo qual o processador e todos os periféricos que possuem uma interface Avalon MM do tipo mestre utilizam para acessar os periféricos que possuem uma interface Avalon MM do tipo escrava. Neste projeto temos dois barramentos separados, um que está baseado na velocidade da memória SDRAM e outro na velocidade do processador. Sempre que existir uma conexão entre dois elementos de velocidades diferentes, é necessário o uso de uma ponte interligando o periférico mais rápido ao periférico mais lento, como no caso do processador com o controlador de memória SDRAM. O processador está operando a 100 MHz enquanto que o controlador da memória, devido às suas configurações, está operando a 75 MHz.

A seguir é feita uma rápida descrição de cada periférico existente no processador.

CpuNIOS é um processador NIOS 2 da Altera®, configurado na versão rápida, para melhor eficiência no desenho dos elementos gráficos na tela de LCD.

SDRAM Controller garante que os sinais sejam gerados para a memória externa de forma coerente, garantindo os requisitos de tempos. Ele possui uma interface Avalon MM escrava com endereços que abrangem os 32MB de memória.

TSE MAC Controller (*Triple Speed Ethernet Media Access Controller*) é o periférico responsável por controlar a rede *ethernet*. Possui duas interfaces Avalon ST, uma do tipo fonte de dados para enviar à memória os dados recebidos pela rede, e outra do tipo receptora de dados, para receber os dados da memória e enviá-los dados pela rede.

SGDMA Tx e Rx (*Scatter-Gather Direct Memory Access*) são periféricos que atuam em conjunto com o periférico TSE MAC. Eles têm a finalidade de ler e gravar os dados trocados pela rede na memória SDRAM sem a intervenção do processador.

LCD SGDMA é o periférico responsável por fazer a leitura dos pixels do *frame-buffer* (região de memória que armazena a imagem, em pixels, que será exibida), e enviar para o LCD. Uma vez inicializado, este módulo realiza sua função de ler dados sem necessitar do intermédio do processador. Após realizar a leitura de um pixel que está na memória, este módulo o envia a um *pipeline* de processamento através de vários periféricos. São necessárias 7 etapas antes que o pixel chegue ao LCD: primeiro adaptador de *clocks* que interliga duas interfaces Avalon ST de velocidades diferentes; memória FIFO (*Fist In First Out*) para garantir a transposição dos dados entre duas interfaces de velocidades diferentes; segundo adaptador de *clocks*; adaptador de dados que converte os dados de 64 *bits* recebidos em 2 de 32 *bits*; inversor dos *bits* que coloca os bits recebidos na ordem 0RGB (vazio, vermelho, verde, azul); adaptador de dados que converte os 32 *bits* recebidos em 4 etapas de 8 *bits*; e por fim o gerador de vídeo, que gera o sinal de vídeo para o LCD.

LCD I2C Controller é o periférico que atua nos registradores do controlador do LCD externo ao FPGA através de comunicação I2C (*Inter Intergrated Circuit*).

SysID é um periférico obrigatório que identifica o processador. Possui um simples registrador com um identificador único que pode ser gerado automaticamente ou manualmente.

Performance Counter é um periférico utilizado para mensurar dois ou mais eventos em termos de pulsos de *clocks*.

Jtag Uart é um periférico utilizado para permitir a depuração e o *download* do *software*

desenvolvido para a memória do processador.

FFT *Pipeline* é o periférico desenvolvido durante este trabalho. Ele recebe os pontos através de sua interface Avalon MM escrava, e aguarda um comando de início. Após o término do processamento, um *bit* de estado é levado a nível lógico alto, indicando o fim do processamento. Quando este módulo está sendo lido ele retorna os dados da FFT calculada.

O PLL (*Phased Locked Loop*) é um componente externo ao sistema desenvolvido no SoPC *Builder*. Ele é responsável por multiplicar o sinal de clock de 50MHz, gerando o sinal de 100 MHz que o sistema necessita para funcionar. Existem versões de PLL que podem ser utilizadas como um periférico do processador.

NEEK: Comunicação pela rede TCP/IP

O módulo de comunicação pela rede é uma parte do *software* que utiliza o protocolo UDP (*User Datagram Protocol*) para envio e recebimento de dados. Foi escolhido este protocolo pois o *overhead* do UDP é menor que do TCP (*Transmission Control Protocol*) em função da ausência de controle complexo da conexão. Contudo, por estar em rede local, a perda de pacotes é mínima, e pode-se obter maior quantidade de dados transmitidos em um mesmo intervalo de tempo.

Este módulo foi dividido em 2 tarefas pois, caso não receba dados do TOWER, o NEEK deve solicitar dados novamente. São utilizadas duas portas distintas para realizar a comunicação. Em uma delas, o módulo envia um pacote de controle requisitando os dados do TOWER e inicia um temporizador. Na próxima porta o módulo aguarda pelo recebimento dos dados e avisa à tarefa anterior que novos dados foram recebidos. Se isto não acontecer antes do tempo máximo expirar, a tarefa de envio solicita novos dados ao TOWER. A Figura 3.15 apresenta estas tarefas de comunicação, seus estados e sua sincronização através de recursos do sistema operacional.

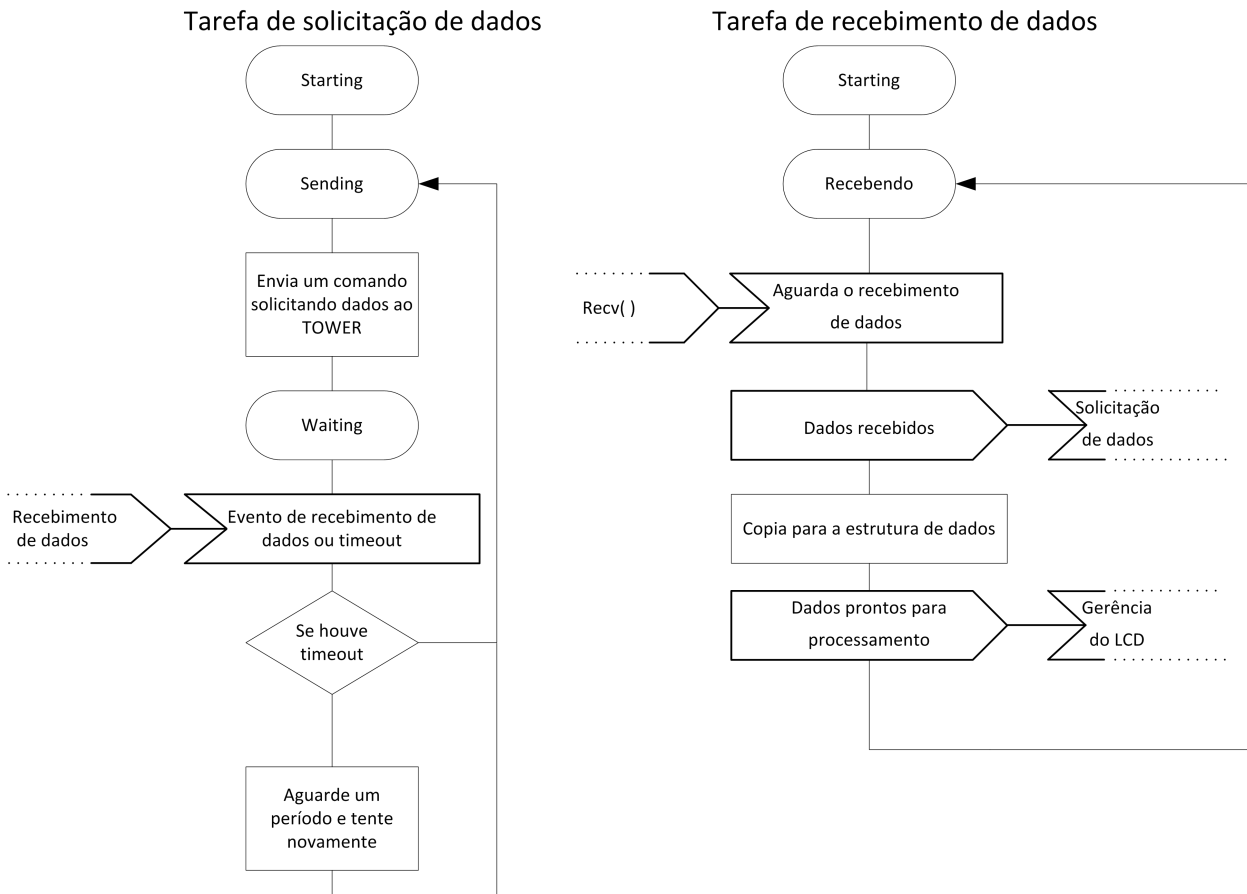


Figura 3.15 - Tarefas do módulo de comunicação

NEEK: Gerenciador de telas

Com o objetivo de facilitar a criação de interfaces, foi desenvolvida uma biblioteca de objetos e um módulo de gerenciamento de telas. Esta biblioteca faz uso de algumas funções do HAL para desenhar primitivas como retas, círculos e retângulos. A partir destas primitivas e dos atributos dos objetos, a biblioteca desenvolvida desenha no LCD. Para tanto foram criados os seguintes objetos:

- Tela
- Botão
- Texto
- Imagem
- Gráfico

- Caixa
- Linha

Cada um destes tipos de dados é composto por uma série de atributos. Estes atributos podem ser valores, ponteiros para dados, ponteiros para ponteiros, ou ainda ponteiros para funções, permitindo que existam métodos e eventos que são chamados quando uma determinada ação acontece. O *software* gerenciador de telas possui um ponteiro chamado "TelaAtual" do tipo "Tela", o qual sempre aponta para o endereço da tela que está sendo exibida no momento. Há uma tarefa responsável por inicializar o LCD e o *Touch Panel*, e monitorar o evento de toque na tela. Esta tarefa é capaz de identificar qual o tipo de evento que ocorreu (clique completo, clique *down* ou clique *up*) e processá-lo, verificando o local do clique e chamando a função referente ao evento do objeto contido na "TelaAtual", caso exista, e ainda redesenhando o objeto caso necessário.

O tipo de dado "Tela" é o mais importante para o *software* de gerenciamento de telas, pois ele armazena todos os objetos presentes em uma tela. Isto permite que as funções de desenho percorram todos os objetos e ainda execute as funções dos eventos. A Figura 3.16 apresenta esta estrutura de dados.

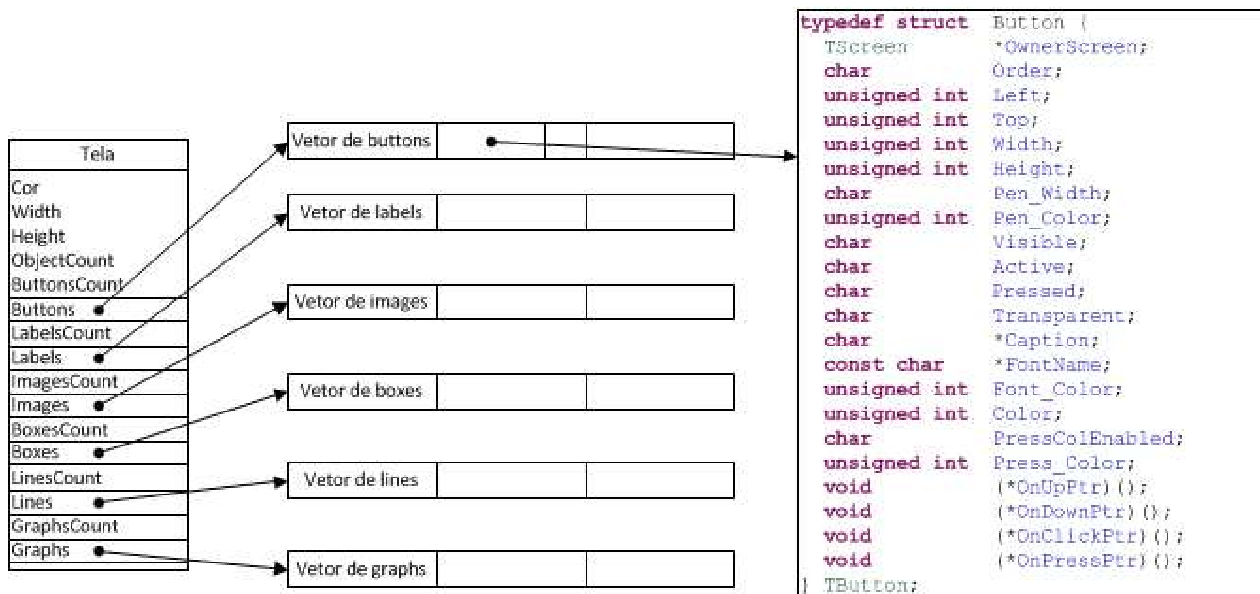


Figura 3.16 - Estrutura de dados da tela e de um botão.

Pela estrutura, observa-se que a tela possui uma cor de fundo, largura e altura definidos por campos numéricos. O campo *ObjectCount* deve conter a quantidade de objetos total da tela, para que no momento do desenho o sistema encontre a quantidade de objetos que serão desenhados.

Os campos com o nome do tipo do objeto concatenado com "*Count*"(por exemplo, *LabelsCount*) indica a quantidade de objetos daquele tipo existentes na tela. Os ponteiros para ponteiros de objetos apontam para vetores de ponteiros de objetos, que devem ser corretamente inicializados com os objetos presentes naquela tela. Assim, a função de desenho navega por estes vetores buscando os objetos e desenhando-os na tela, assim como a função de verificação do *Touch Panel* navega por estes objetos em busca dos objetos que se encontram no local do clique.

Foi implementado o algoritmo DFT, com complexidade quadrática, e a versão rápida (FFT) com complexidade N(log2(N)). A implementação foi feita para ser executada pelo processador NIOS2 em *software*, utilizando de um conjunto de instruções de ponto flutuante em *hardware*, o que acelera o processamento. Através destas rotinas criou-se uma biblioteca de processamento da FFT em *software*. Elas foram utilizadas para comparar o resultado do processamento em *hardware* com o processamento em *software*, no mesmo processador. Algumas destas rotinas serão reutilizadas para fazer a reordenação dos dados.

O sistema de geração de imagem pode ser consultado em Altera (2011o) e o conjunto de periféricos para acessar e monitorar o LCD pode ser consultado em Altera (2011d). O tutorial apresentado em Altera (2011j) detalha as rotinas em linguagem C de criação e gerenciamento do LCD e as funções de HAL para desenhar na tela. O apêndice C apresenta a estrutura completa do módulo de gerenciamento de telas em linguagem de programação C.

3.2 Algoritmo FFT em FPGA

De acordo com a proposta deste trabalho, esta arquitetura de processamento de sinais foi projetada para ser genérica, aplicável a qualquer ferramenta matemática para processamento de sinais. A Transformada Discreta de Fourier foi escolhida como ferramenta matemática de processamento. Existem diversos métodos via *hardware* (tanto paralelos quanto seriais) e via *software* para realizar tal tarefa. O método para calcular a FFT proposto neste trabalho, é utilizado em aplicações onde os dados entram de forma serial no sistema. Por isto o algoritmo utilizado foi o *Radix-2 multi-path delay commutator*, utilizando dizimação em frequência (DIF), originalmente proposto por Rabiner e Gold (1975). Esse algoritmo foi escolhido por apresentar eficiência na execução da FFT, ser de

fácil entendimento e apresentar simplicidade na criação dos componentes diretamente em VHDL sintetizável, e ainda permitir implementação de forma genérica deixando o sistema mais flexível para expansões.

Com este algoritmo os dados entram no *pipeline* em sua ordem natural. No final do processamento, os dados se apresentam armazenados em memórias separadas (um par de memórias por saída do *pipeline*, armazenando o resultado complexo), ordenados segundo o algoritmo *Bit*-reverso. Quando os dados são lidos do módulo desenvolvido, devem ser reordenados para garantir a interpretação do resultado.

O processamento foi realizado utilizando aritmética de ponto fixo com 18 *bits*, dos quais 2 *bits* referem-se à parte inteira e sinal, e 16 *bits* referem-se à parte fracionária. Desta maneira, o sistema de numeração pode ser representado por $S(1,16)$. Foram utilizados deslocamentos virtuais após as multiplicações e literais após as somas no final de cada estágio do *pipeline*.

Para validar o método, o *pipeline* foi criado para 1024 pontos, ou seja 10 estágios. O número de estágios S do pipeline é dado por

$$S = \log_2 N \quad \text{sendo N o número de pontos.} \tag{3.1}$$

Porém, para a arquitetura completa desenvolvida foram utilizados apenas 256 pontos, ou seja, 8 estágios. Isto foi necessário em função da quantidade de elementos lógicos e memória RAM interna presente no FPGA escolhido, e devido à presença do processador NIOS 2 com todos os periféricos necessários para o funcionamento da arquitetura.

3.2.1 O fluxo de dados

Inicialmente os dados estão presentes em uma memória externa ao módulo e são posteriormente movimentados para a memória interna do periférico desenvolvido. Logo após o início, passam pelo *pipeline* de processamento da FFT, e são armazenados em outra memória, para então serem movimentados novamente para a memória externa. A Figura 3.17 apresenta o diagrama do fluxo de dados.

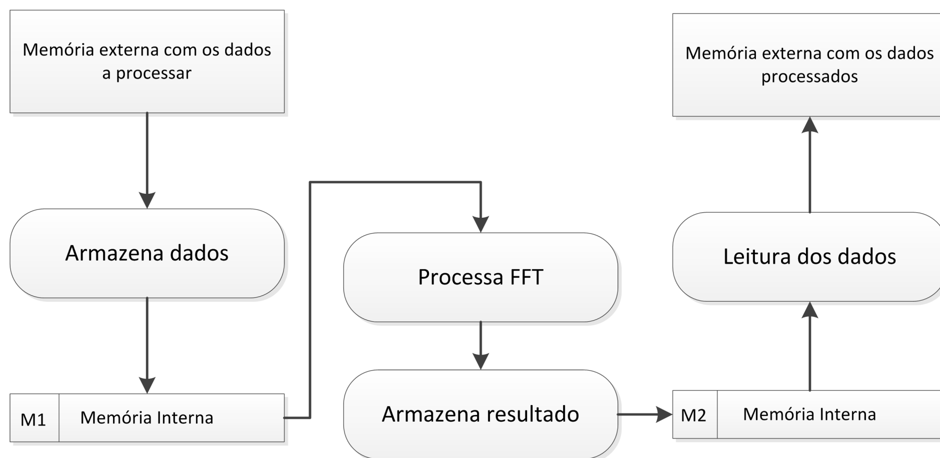


Figura 3.17 - Diagrama do fluxo de dados no módulo em desenvolvimento.

O *pipeline* desenvolvido para o processamento da FFT baseado no R2MDC foi realizado utilizando estágios. Cada estágio é uma descrição genérica de um conjunto de elementos lógicos e componentes genéricos. Assim, para acrescentar mais estágios a um *pipeline*, basta instanciá-los e passar o seu índice como parâmetro. Desta forma ficou simples a adição e remoção de estágios ao *pipeline*. A Figura 3.18 apresenta um *pipeline* construído com 8 estágios. Os dados entram no primeiro estágio à esquerda, o estágio de maior índice, e em função do comportamento serial do algoritmo, transportam-se para o estágio mais a direita, de menor índice.

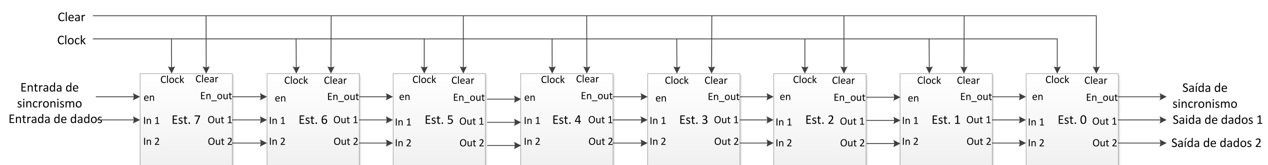


Figura 3.18 - Fluxo de dados em um pipeline de 8 estágios.

O estágio é o componente principal do *pipeline*. Um estágio é composto de:

- Uma fila de 2^S elementos, sendo S o índice do estágio;
- Duas filas de $(2^S) + 1$ elementos, sendo S o índice do estágio;
- Dois multiplexadores auto-controlados, também chamados de comutadores;
- Um registrador simples;
- Um operador borboleta com dizimação em frequência (DIF);
- Um gerador de constantes W (*twiddle factors*).

A Figura 3.19 apresenta os sinais de entrada e saída de um estágio e a Figura 3.20 apresenta um estágio detalhado, com todos os seus elementos genéricos.

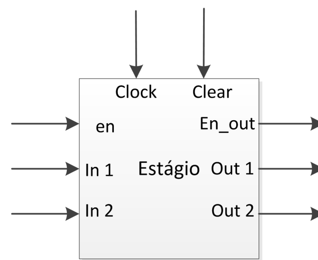


Figura 3.19 - Entradas e saídas de um estágio.

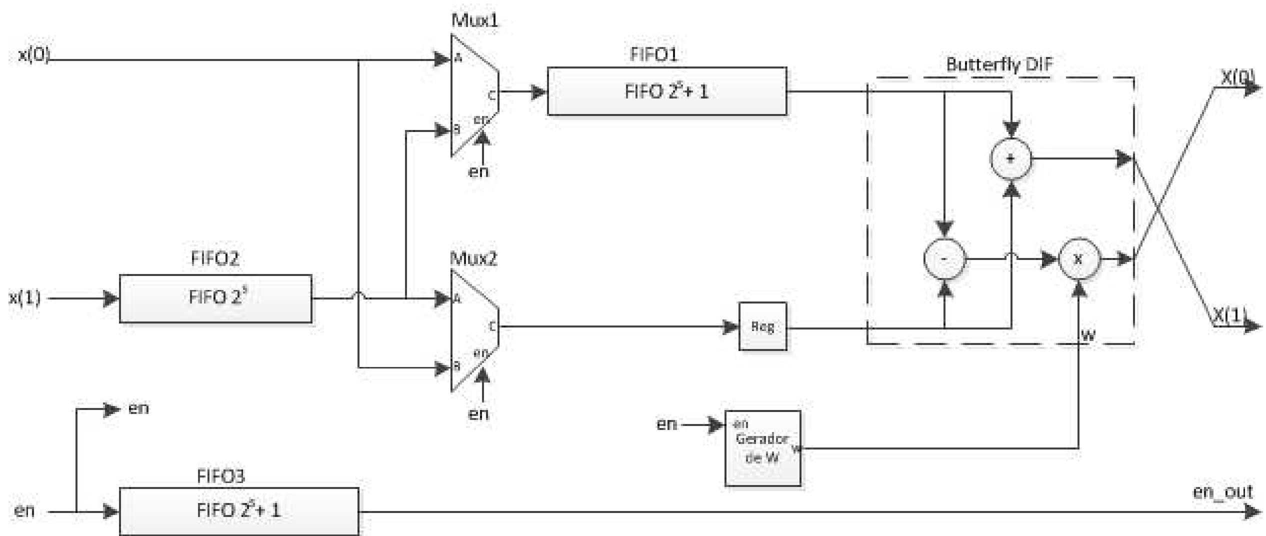


Figura 3.20 - Detalhamento da estrutura interna de um estágio.

Para sincronizar os estágios de um *pipeline*, ou seja, fazer com que cada um inicie seu processo no momento exato, foi criado um sinal de sincronismo que é passado de estágio à estágio em direção à saída do sistema. Este sinal entra no primeiro estágio junto com o primeiro ponto. Cada estágio possui uma fila de comprimento igual a 2 elevado ao índice do estágio mais 1, onde o sinal de sincronismo trafega. Assim, quando o primeiro ponto a entrar no primeiro estágio está saindo deste estágio, o sinal de sincronismo também está, habilitando o próximo estágio a funcionar. Desta maneira, quando o último estágio liberar o sinal de sincronismo, isto significa que o primeiro par de pontos está saindo do *pipeline*. A partir de então serão necessários sair os $N/2$ pares de pontos restantes. O sinal de sincronismo é visto na Figura 3.20 como "*en*"e "*en-out*", respectivamente entrada de sincronismo que habilita todos os componentes internos e saída de sincronismo para o próximo estágio.

Os multiplexadores(MUX) são os elementos comutadores. Eles têm a função de direcionar os dados para a fila ou para o registrador. Se não fosse sua característica auto-controlada, eles

seriam compostos de lógica puramente combinacional. Devido ao algoritmo, a cada 2^S elementos o multiplexador inverte sua seleção. Foi necessário desenvolver um pequeno componente para o controle do seletor do multiplexador. Este controle foi feito através de uma lógica de contagem simples baseada no *clock* e no sinal de habilitado (*enable*).

As filas têm a finalidade de provocar atrasos nos sinais de forma a alinhar a geração do fator *twiddle* com os dados de entrada. Cada elemento que entra na fila empurra o elemento da posição atual para a próxima posição. Quando a fila estiver cheia, ao entrar um novo elemento, o primeiro elemento que entrou irá sair, ou seja, o primeiro a entrar é o primeiro a sair (FIFO, *First In First Out*). A fila 1(FIFO1) recebe dados ou da entrada ou da fila 2(FIFO2), de acordo com a seleção feita pelo comutador MUX1. A fila 2(FIFO2) sempre recebe os elementos da segunda entrada. A fila 3(FIFO3) é responsável por gerar os atrasos no sinal de sincronismo. O registrador(REG) é uma fila de um único elemento. No estágio inicial a fila 2 não é utilizada, pois todos os dados entram pela entrada $x(0)$ do estágio.

O gerador de fatores *twiddle* (Gerador de W) é um componente simples que faz busca a uma lista de constantes pré-calculadas e escritas em 18bits no formato $S(1, 16)$. Tem a finalidade de colocar na sua saída (gerar) o respectivo fator referente ao cálculo que está sendo executado naquele momento. Ele sempre realiza a contagem de 0 a 2^S, e em cada estado libera o fator correspondente. Assim como os demais elementos do estágio, ele só dá início à contagem quando recebe o sinal de sincronismo.

Por fim, o componente *Butterfly* DIF realiza a operação borboleta do algoritmo de dizimação na frequência, pois os dados entram na sua ordenação natural. Ele realiza a operação segundo a seguinte fórmula:

$$X(0) = (x(0) - x(1)) \times W_N^k \tag{3.2}$$

$$X(1) = x(0) + x(1) \tag{3.3}$$

Assim o diagrama de fluxo da Figura 3.21 apresenta o diagrama de fluxo da equação. Estas operações são realizadas através de matemática de números complexos utilizando pontos fixos. O resultado da multiplicação sofre um deslocamento virtual enquanto que o resultado da adição um deslocamento literal em um *bit*.

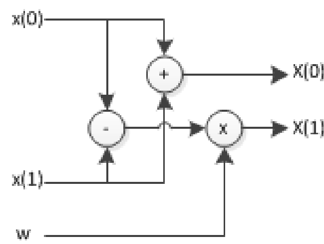


Figura 3.21 - Operação borboleta no algoritmo DIF.

Portanto, o *pipeline* foi encapsulado em um componente na linguagem VHDL e alguns sinais de entrada e de saída. O sinal *"en"* é a entrada de sincronismo do primeiro estágio do sistema, ou seja, nada acontece até que este sinal seja 1. Quando o último dado entrar no sistema este sinal deve ser levado para 0. O sinal *"en-out"* corresponde à saída de sincronismo do último estágio, ou seja, quando este sinal possuir o valor lógico 1, significa que o primeiro resultado válido está saindo do sistema. O sinal *"clock"* é o sinal que todos os registradores internos usam como referência para armazenarem o próximo dado, ou seja, é o sinal de relógio do sistema. O sinal *"clr"* faz com que todos os elementos de todos os estágios voltem à sua condição inicial para que uma nova transformada seja aplicada. Os sinais do tipo complexo são os dados de entrada e de saída do sistema. A Figura 3.22 apresenta estes sinais. O apêndice A mostra uma simulação passo a passo de uma FFT de 8 pontos.

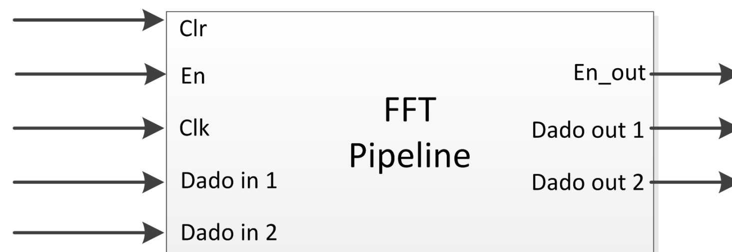


Figura 3.22 - Sinais de entrada e saída do pipeline.

Com este algoritmo, temos uma eficiência de 50% de utilização dos multiplicadores e das memórias envolvidas. Isto ocorre em função do tempo em espera que os estágios permanecem inativos, comprovado por Li (2003).

3.2.2 O controlador do fluxo de dados

O *pipeline* sozinho não realiza o processo por completo. É necessário um sistema de controle, baseado em máquinas de estados, para ler os dados da memória RAM interna, passá-los pelo *pipeline* e armazená-los em outra memória RAM interna. Feito isto, é preciso retirar estes dados das memórias para serem visualizados. Isto foi feito através da porta serial para fins de validação do resultado. Foi criado um módulo de comunicação serial e um módulo de controle, utilizados para transmissão dos dados do resultado à um computador de uso geral. Foi transmitido o sinal original no domínio do tempo e o sinal calculado pelo algoritmo FFT.

Em um computador pessoal, o *software* Matlab® foi utilizado para receber os dados pela porta serial. O sinal original no tempo foi recebido e usado como entrada para função FFT do próprio Matlab®. O sinal processado pelo FPGA foi convertido para o formato de ponto flutuante e comparado com o resultado obtido pelo Matlab®.

A Figura 3.23 apresenta o controlador completo.

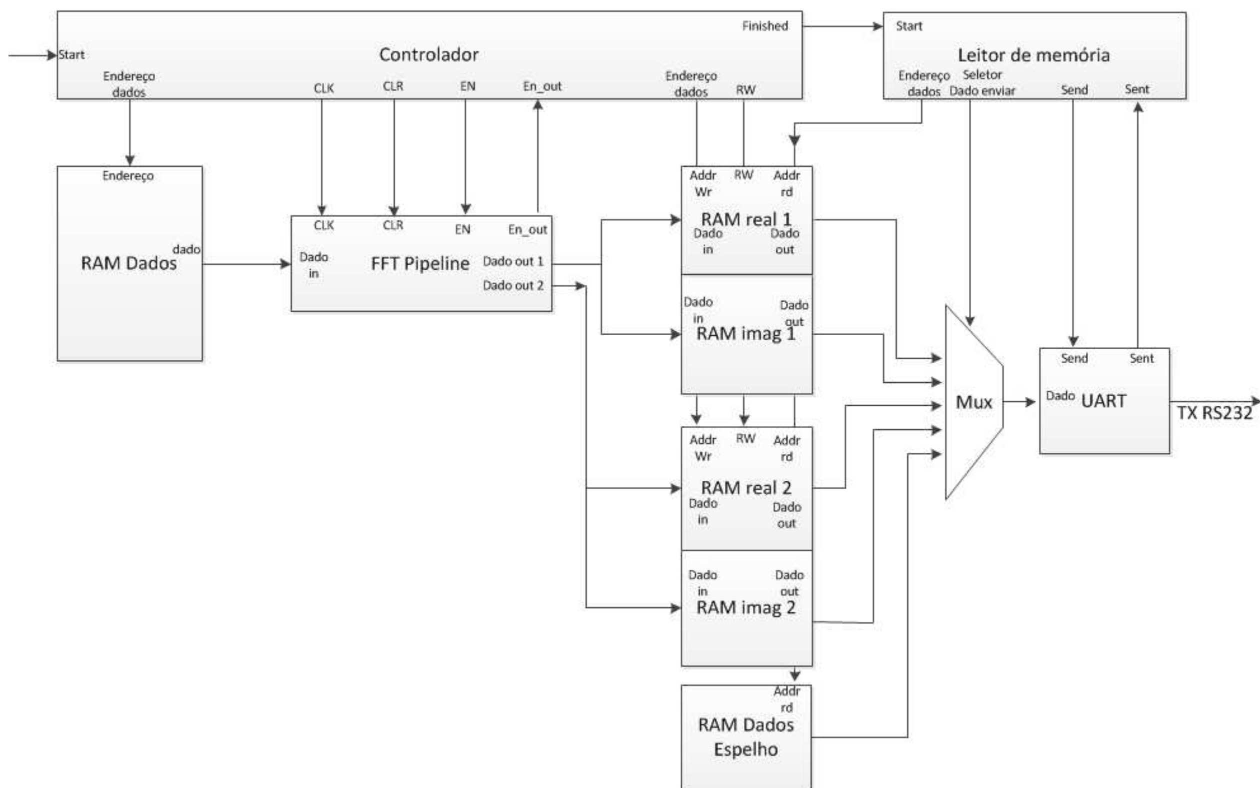


Figura 3.23 - Controlador para o pipeline FFT.

A memória RAM Dados e a memória RAM Dados Espelho possuem o mesmo conteúdo. Para o teste, o conteúdo foi pré-inicializado com sinais discretos preparados manualmente, para que na saída obtenha-se um resultado conhecido. A RAM Dados fornece dados para o *pipeline* enquanto que a RAM Dados Espelho fornece os dados originais para o transmissor de dados. Foi preciso enviar o sinal no tempo original ao Matlab®, para ser convertido para ponto flutuante sob a mesma métrica do sinal calculado pelo FPGA. Os resultados do MatLab® deveriam ser muito parecidos com os resultados obtidos com o FPGA.

O componente FFT *Pipeline* é o algoritmo apresentado na seção anterior. Para este sistema, foram utilizados 256 pontos, ou seja 8 estágios. Poderiam existir mais que 8 estágios, em função de não existir ainda um processador no sistema. Com isto, existem milhares de elementos lógicos não utilizados que poderiam ter sido explorados. O sinal *dado out 1* é um número complexo, formado por parte real e parte imaginária, cada um com 18 *bits* de resolução. Quando o sinal de sincronismo aparece na saída do *pipeline*, cada dado é armazenado em uma memória RAM diferente.

As memórias RAM (*real1, imag1, real2, imag2*) são memórias *Dual-port*, ou seja, possuem uma porta somente para escrita e outra somente para leitura dos dados. A porta de uma memória refere-se a um par endereço-dado. Todas as memórias são endereçadas para escrita simultaneamente e os dados que estão em suas entradas são armazenados simultaneamente. O endereço de leitura também está compartilhado, porém ainda acrescenta-se a memória RAM Dados Espelho, que contém os mesmos dados da RAM Dados, para que o sinal original seja enviado pela porta serial.

O Controlador é o componente que garante a sincronia dos componentes até aqui comentados. Ele garante a geração dos endereços para a leitura da memória RAM Dados, a geração de pulsos de *clock* para o componente FFT *Pipeline*, o sinal de reiniciar do FFT *Pipeline*, o sinal de inicio de sincronismo do FFT *Pipeline*. Também prepara as memórias de recebimento de dados para irem armazenando os resultados no endereço inicial, enquanto aguardam o sinal de sincronismo vindo do FFT Pipeline. Isto coloca o Controlador num estado em que os próximos dados provenientes do *pipeline* entrarão nos endereços seguintes das memórias RAM de resultado. Neste estado o Controlador incrementa o endereço a cada novo resultado. Ele permanece neste estado até que $N/2$ pulsos de *clock* tenham ocorrido, indicando que todos os pontos já foram calculados. Neste instante ele reinicia o FFT *Pipeline* e gera um sinal de fim de processamento para o componente de Leitura de memória.

O componente de Leitura de memória é responsável por enviar todos os pontos armazenados no banco de memórias para um computador através da porta serial. Este componente envia primeiro todos os pontos do sinal no tempo, e depois parte para o envio dos pontos processados. Os pontos do sinal no tempo se encontram na memória RAM Dados Espelho, enquanto o banco de memórias armazenam o resultado com números complexos do sinal processado. Este leitor de memória decide qual ponto será enviado através do endereço de leitura, comum a todas as memórias. Ele incrementa o endereço a cada ponto enviado. Após terminar de enviar os pontos da memória RAM Dados Espelho, dá início ao envio dos dados resultantes do processamento. Para tanto, ele reinicia o endereço de leitura, e para cada endereço envia quatro dados, referentes aos reais e imaginários armazenados em cada memória. Por isto ele somente incrementa o contador de endereços após o envio dos quatro dados. Para enviar qualquer dado, o Leitor de memória deve aguardar o dado sair da memória após aplicar o endereço, selecionar o dado a ser enviado através do multiplexador, e gerar um sinal "*Send*"para o controlador UART. Este sinal dá início ao processo de enviar o dado pela porta serial. No término, o controlador UART envia um sinal "*Sent*"para o Leitor de memória, e este parte para o envio do próximo dado. Este ciclo se repete para todo dado que é enviado. Um dado enviado, apesar de ter 18 *bits*, é concatenado com mais 14 *bits* para formar um conjunto de 4 *bytes*, facilitando a formatação dos dados no Matlab, onde os dados recebidos são desenhados na tela.

3.2.3 Implementação do módulo personalizado para o cálculo da FFT

Para fazer uso do algoritmo em um sistema completo, é necessário integra-lo ao processador NIOS 2. Isto foi feito através da criação de interfaces para o barramento Avalon dentro do módulo FFT. Neste ponto uma decisão importante se faz presente sobre a escolha do barramento. Pelo fato de existir memórias internas endereçáveis tanto para leitura quanto para escrita, este componente se comporta como um periférico baseado em endereços, ou seja, mapeado em memória ou *Memory Map*. Logo a interface Avalon MM é uma boa escolha para o sistema. Como os dados são lidos e escritos neste periférico, esta interface será do tipo escrava. Quando o processador fizer uso do periférico desenvolvido, os dados entrarão de forma serial na memória interna. Quando for ler os

dados processados, o processador deverá retirar os dados da memória interna do periférico, que estão ordenados segundo o algoritmo *"Bit-reverse"*.

Nesta implementação, optou-se por utilizar o processador para colocar os dados no periférico. Isso foi implementado em função da falta de elementos lógicos necessários para a criação de mais um periférico no processador. A Figura 3.14 mostra a configuração final do processador desenvolvido. Note a presença do periférico FFT *Pipeline* no barramento Avalon. Para criar este periférico, foi preciso adicionar a interface Avalon MM escrava ao componente FFT desenvolvido. A Figura 3.24 mostra a arquitetura interna detalhada do periférico desenvolvido.

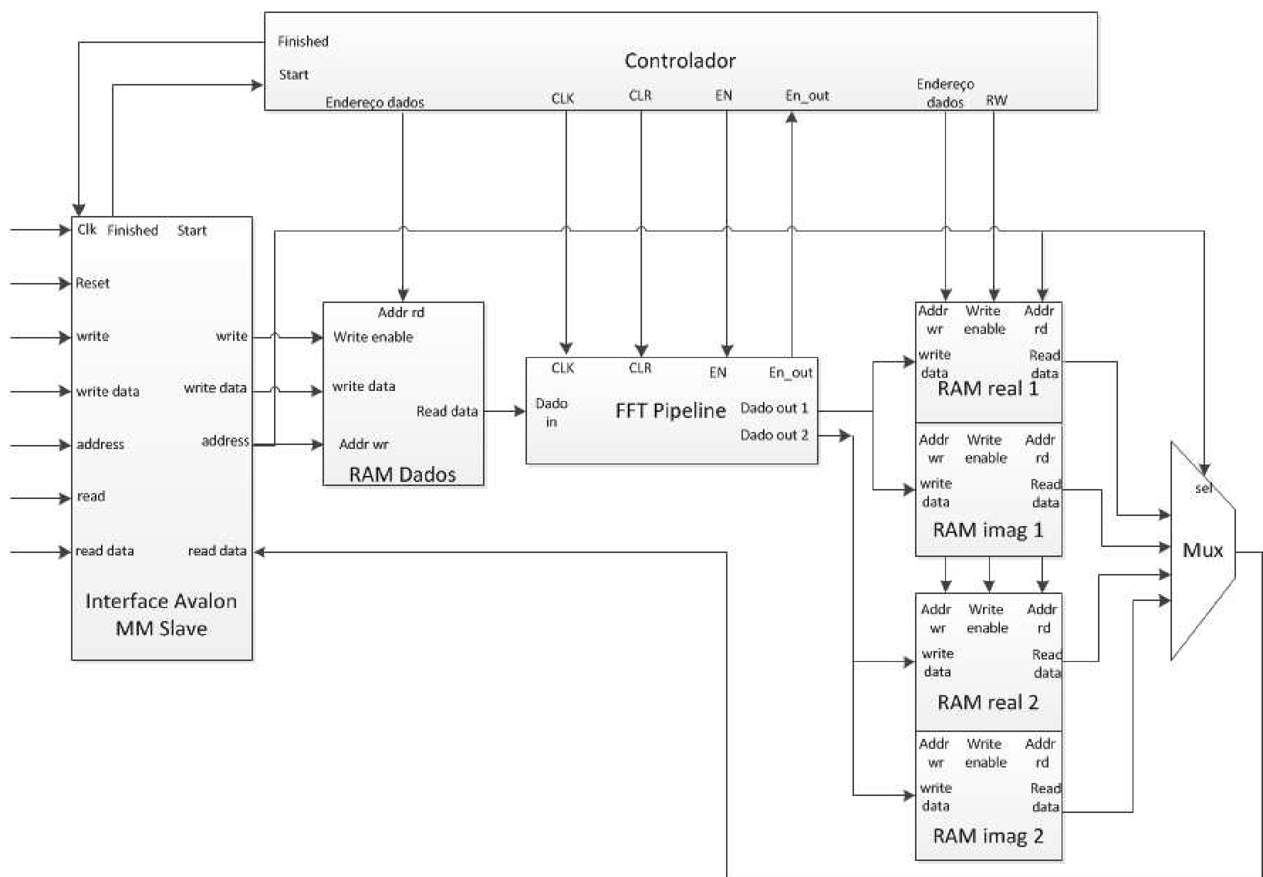


Figura 3.24 - Controlador para o pipeline FFT.

Este módulo possui quase os mesmos componentes do módulo utilizado nos testes. A diferença é que antes os dados eram transmitidos de forma serial para um computador pelo controlador UART, e agora os dados resultantes serão lidos das memórias internas pelo processador. Também aparece um novo componente, a interface Avalon MM. A interface Avalon MM é responsável por

interconectar os elementos do módulo desenvolvido ao barramento de dados do processador.

Este módulo apresenta duas memórias internas. Uma para escrita dos dados referente ao sinal no tempo, localizada à esquerda do *pipeline*, e um bloco de memórias que receberá o resultado do processamento. Estas memórias serão lidas posteriormente pela interface Avalon MM.

A interface Avalon possui dezenas de sinais diferentes. Para este módulo, foram utilizados apenas um sinal de saída de dados e seis sinais de entrada de dados. O sinal de *clock* recebido pela interface é repassado para todos os componentes do periférico, assim como o sinal de *reset*. O sinal *CS* (*Chip Select*), quando igual a 1, indica que o módulo foi selecionado. Neste caso, a combinação deste sinal com o sinal de escrita (*Write*), implica que o dado a entrar no sistema deverá ser escrito na memória interna. Para tanto, o sinal *address* recebe o endereço em que o dado deverá entrar. O sinal *writedata* recebe o dado que entrará na memória. Estes sinais são repassados diretamente para os seus receptores. O sinal *read* implica que o processador irá realizar uma leitura de dados. Assim, o sinal de endereços é agora aplicado ao conjunto de memórias onde o resultado do cálculo se encontra. Além disto, os 2 *bits* mais significativos do sinal de endereço foi utilizado como seletor do multiplexador das memórias, indicando se irá receber o dado da memória dos reais 1, imaginários 1, reais 2 ou imaginários 2. Isto foi necessário em função das memórias estarem em componentes separados devido ao paralelismo exigido pelo *pipeline* na sua saída de dados. Assim, o sinal *readdata* recebe o conteúdo da memória selecionada pelo endereçamento.

Na interface Avalon MM ainda foi implementado um registrador de controle e estado. Este registrador é lido ou escrito quando acessamos o último dos endereços possíveis. Neste registrador, existem 2 *bits* importantes. O *bit* menos significativo corresponde à ação de iniciar o processamento. Após os dados terem sido colocados na memória RAM, o *pipeline* não inicia até que este *bit* seja levado para um. Imediatamente após este *bit* ser levado para um o controlador do *pipeline* inicia o processo e o *bit* é automaticamente desativado. No final do processamento o controlador do *pipeline* gera um *bit* de fim de processamento, o sinal *finished*. Este *bit* é então escrito neste registrador de estado e controle no próximo *bit* de menor significância. Assim é possível detectar o fim do processamento através da leitura deste registrador, e por fim realizar a leitura da memória.

Os demais componentes do sistema possuem exatamente o mesmo funcionamento explicado na subseção anterior.

3.2.4 Integração do módulo implementado com o NIOS 2

Para integrar o módulo desenvolvido ao processador NIOS 2, é necessário utilizar o *software* SoPC *Builder*. A principal tarefa é realizar as conexões do barramento Avalon com as entradas e saídas do módulo em desenvolvimento. A Figura 3.25 mostra esta tela.

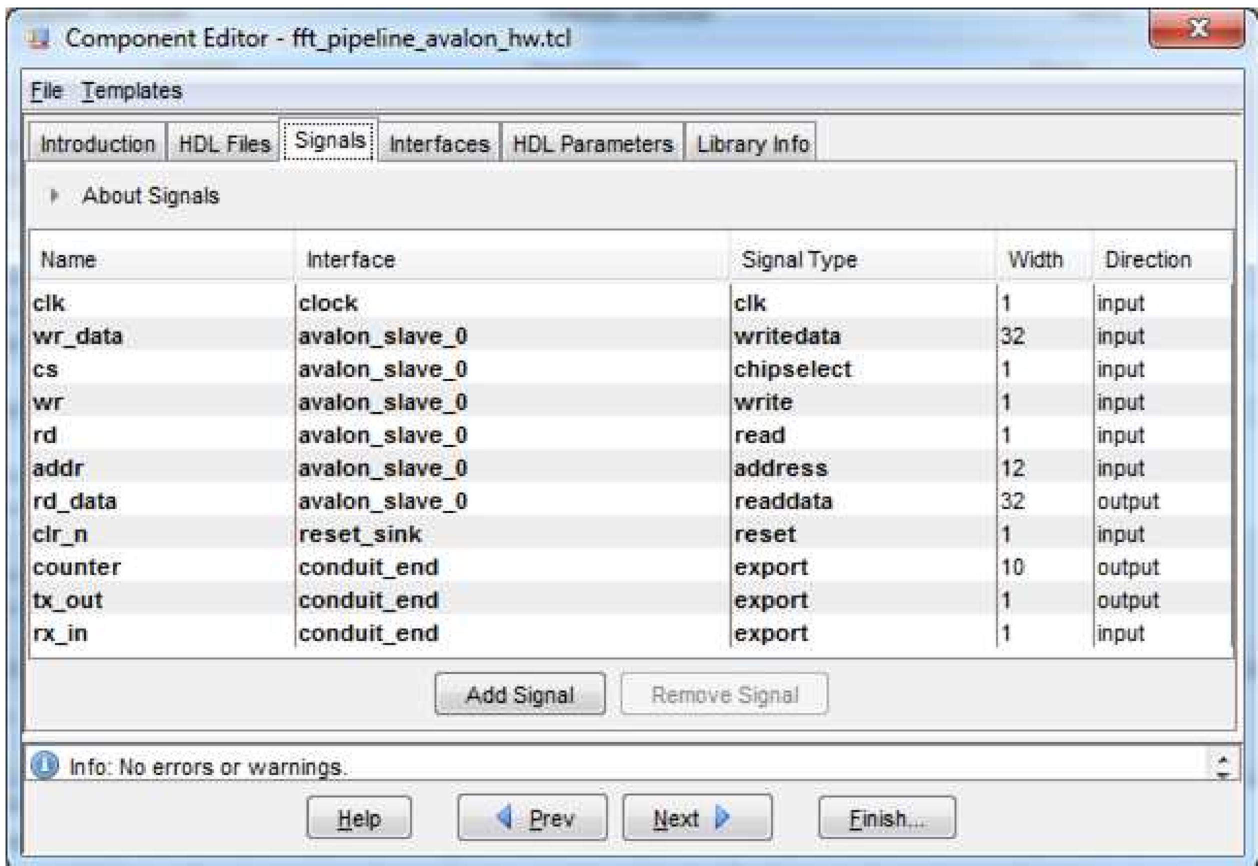


Figura 3.25 - Tela de configuração dos sinais do periférico desenvolvido.

Depois de adicionado à biblioteca de periféricos, deve-se instanciar um periférico do tipo criado e por fim realizar a sua conexão com o barramento desejado, que neste caso foi o barramento de dados do processador NIOS 2.

Após adicionado no SoPC *Builder*, gerado o sistema, compilado no Quartus e sintetizado no FPGA, a próxima etapa é criar o modelo de *software* que irá acessar este periférico do processador através de comandos diretos de escrita e leitura de dados em endereços de memória. Para este periférico foram implementados os comandos de escrever dados na memória interna, ler dados

do resultado da memória interna, iniciar o processamento e verificar o estado do processamento. Todos estes comandos foram implementados através de definições em linguagem C. A lista a seguir apresenta estes comandos.

```
#define FFT_ESCREVE_DADO(pos, data) IOWR(FFT_PIPELINE_BASE, pos, data)
#define FFT_LE_DADO(pos) IORD(FFT_PIPELINE_BASE, pos)
#define FFT_START()IOWR(FFT_PIPELINE_BASE, 0x0800, 1)
#define FFT_FINISHED() IORD(FFT_PIPELINE_BASE, 0x0800)
```

O comando FFT_ESCREVE_DADO(pos, data) escreve no endereço base somando a posição do dado a ser escrito. Visto a partir do módulo desenvolvido, o sinal *address* receberá o conteúdo do parâmetro *pos*, e o sinal *writedata* receberá o conteúdo do parâmetro *data*.

O comando FFT_LE_DADO(pos) retorna o conteúdo da memória no endereço passado pelo parâmetro *pos*.

O comando FFT_START() escreve o valor 1 no endereço do registrador de estado e controle, dando início ao processo. Neste momento o bit de fim do algoritmo recebe o valor zero. O comando FFT_FINISHED() retorna o conteúdo do registrador de estado e controle. Se o bit referente ao fim do processamento estiver em 1, significa que o processo já terminou.

4 RESULTADOS

Este capítulo apresenta os resultados obtidos com a arquitetura proposta para o processamento de sinais em 3 diferentes situações.

Na primeira situação, o módulo de processamento da FFT em *hardware* foi colocado à prova através de dados pré-armazenados na memória RAM. Estes dados foram processados pelo módulo FFT e enviados através da porta serial para um computador com o *software* Matlab® para serem comparados.

Na segunda situação, o módulo de aquisição realizou a medição do sinal analógico e enviou os dados para o FPGA, que processou os dados com o módulo de processamento da FFT e enviou o resultado para o Matlab® através da rede local.

Na terceira situação, o módulo de aquisição realizou a medição do sinal analógico e enviou os dados para o FPGA através da rede local. Os dados foram processados em *hardware* pelo FPGA e apresentados na interface homem-máquina através de gráficos.

O tempo de execução foi mensurado através do periférico de medição de desempenho presente no processador. Os dados originais foram processados novamente através de um algoritmo escrito totalmente em *software*, onde o tempo de execução foi mensurado, com a intenção de fazer uma comparação de desempenho. Por fim, foi apresentada a quantidade de elementos lógicos utilizados pelo processador e pelos seus principais periféricos.

4.1 Testes realizados e seus resultados

4.1.1 Situação 1

Os dados de uma senoide discretizada estão pré-armazenados na memória interna do periférico. Um controlador interno realiza o processamento dos dados na memória passando-os pelo componente de cálculo da FFT (*pipeline*), e armazenando os dados resultantes em um conjunto

de memórias RAM internas. O componente de leitura da memória realiza a leitura dos dados e os envia pela porta serial. Primeiro é feita a leitura e envio dos dados originais, a senoide. Depois são enviados os dados referentes ao resultado do processamento. A Figura 3.23 foi utilizada durante esta validação. A Figura 4.1 apresenta um diagrama de fluxo de dados simplificado do método utilizado neste caso de teste. No computador, os dados são recebidos pelo Matlab® através da porta serial. É realizada a FFT com o sinal original utilizando as rotinas do próprio Matlab® para realizar a comparação com o resultado obtido com o FPGA.

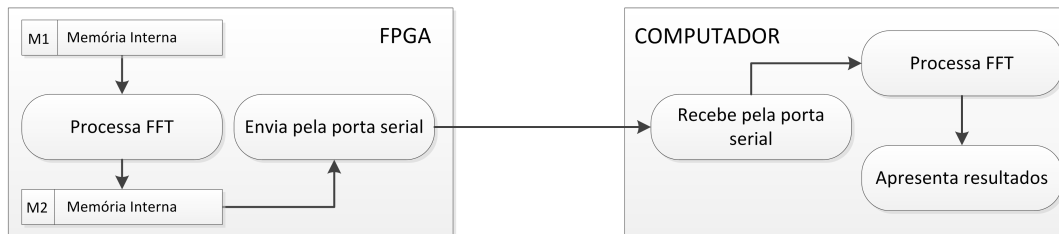


Figura 4.1 - Diagrama de fluxo de dados do teste realizado na primeira situação.

Inicialmente foi criada uma senoide com frequência de 4Hz discretizada em 256 pontos e armazenada na memória RAM interna do módulo. Considerando um tempo de aquisição de 1 segundo, a taxa de amostragem foi de 256Hz. Estas informações são relativas, pois poderíamos considerar outros valores. A Figura 4.2 mostra o resultado exibido pelo Matlab®. O primeiro gráfico apresenta os dados originais do processamento, o segundo apresenta o cálculo realizado pelo FPGA, enquanto que o terceiro apresenta o resultado calculado pelo próprio Matlab®. Nos dois gráficos de espectro de frequências os picos estão no mesmo ponto, garantindo o resultado correto do cálculo pelo FPGA. A amplitude da frequência apresentou pequenas variações, em função da resolução dos números de ponto fixo.

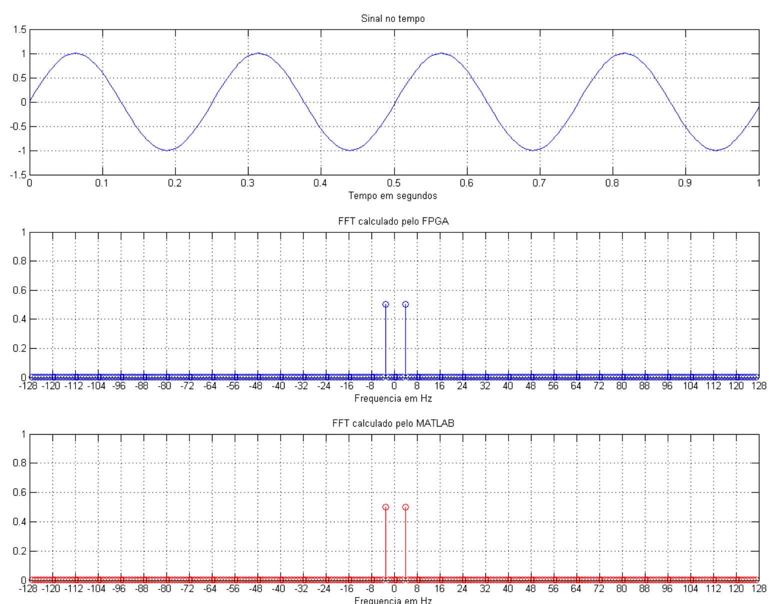


Figura 4.2 - Sinal original de 4Hz, FFT calculada pelo FPGA e FFT calculada pelo Matlab®.

Foi realizado um segundo teste com este módulo, aumentando a frequência para 32 Hz. A Figura 4.3 apresenta os resultados obtidos. Nota-se a semelhança entre a FFT calculada pelo FPGA e pelo Matlab®.

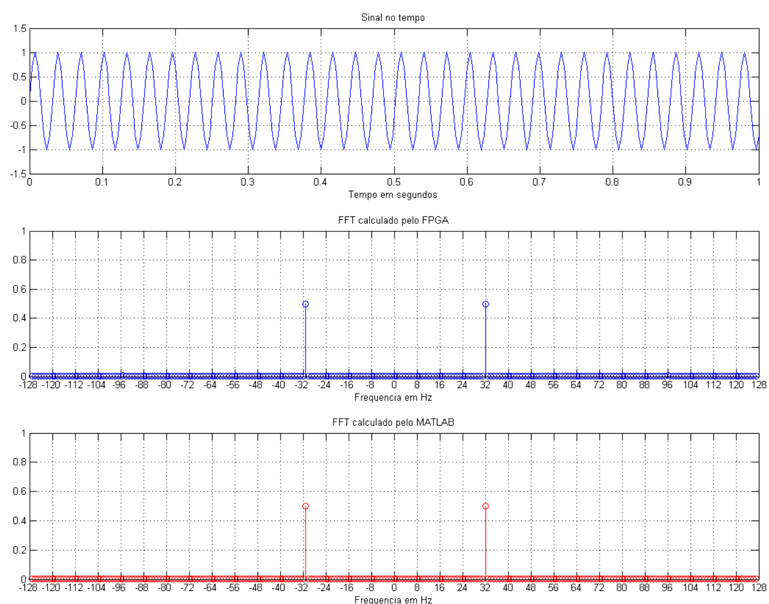


Figura 4.3 - Sinal original de 32Hz, FFT calculada pelo FPGA e FFT calculada pelo Matlab®.

Um último teste foi realizado para este módulo, utilizando uma frequência de 64Hz. A Figura 4.4 apresenta os resultados obtidos. Nota-se a semelhança entre a FFT calculada pelo FPGA e pelo Matlab®.

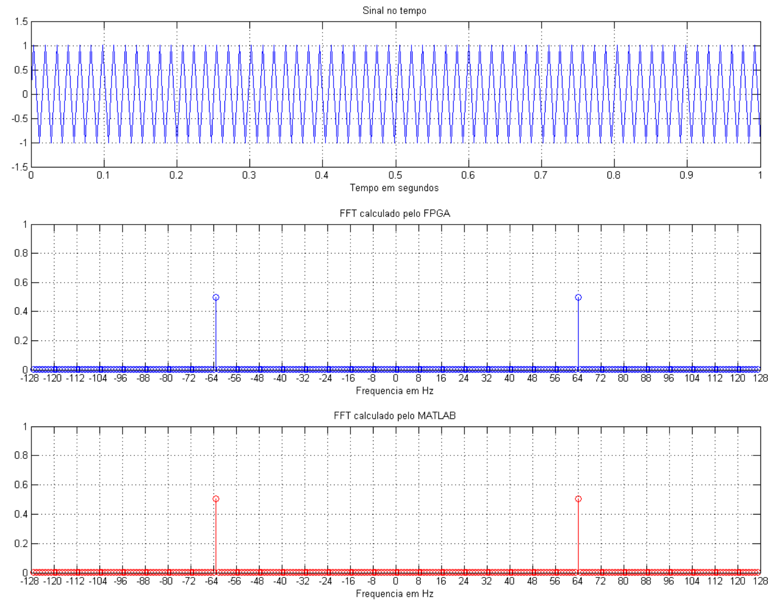


Figura 4.4 - Sinal original de 64Hz, FFT calculada pelo FPGA e FFT calculada pelo Matlab®.

4.1.2 Situação 2

Com os resultados obtidos na subseção anterior foi considerado que o módulo está apto para ser utilizado em um sistema automático de geração de sinais ou num sistema de aquisição de sinais reais. Portanto, partiu-se para esta implementação. O TOWER realiza a aquisição de 256 pontos, sob uma taxa de amostragem de 50KHz, ou seja, um período de aquisição de 5.12 milissegundos. Os dados são enviados para o FPGA através da rede local, que os processa e retransmite para um computador com o Matlab®, também utilizando a rede local. O módulo desenvolvido na Figura 3.24 foi integrado ao processador NIOS 2, como pode ser visto na Figura 3.11, e utilizado no processamento dos dados. A Figura 4.5 mostra um diagrama de fluxo de dados simplificado do método utilizado neste caso de teste.

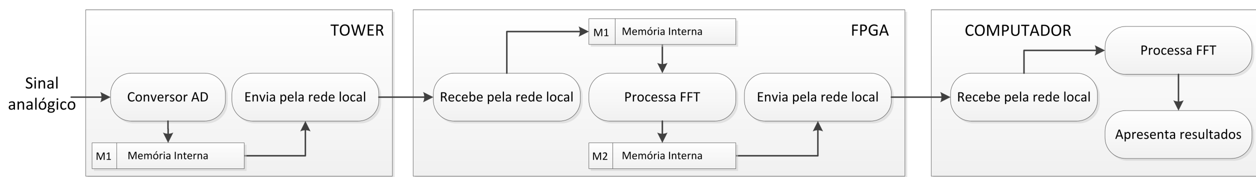


Figura 4.5 - Diagrama de fluxo de dados do teste realizado na segunda situação.

Foi utilizado um gerador de funções para aplicar os sinais que foram coletados pelo TOWER. Através deste gerador de funções foram aplicados sinais de 1 a 25KHz de diferentes formas de onda. No computador, os dados recebidos pela rede foram desenhados em gráficos. A Figura 4.6 apresenta três gráficos referentes ao sinal utilizado neste teste, uma senoide de 1KHz. No primeiro gráfico o sinal original pode ser visualizado. Nota-se que não há uma quantidade exata de períodos, dando origem ao efeito de vazamento nos gráficos dos espectros de frequências. No segundo gráfico está o cálculo do espectro de frequências realizado pelo FPGA, e no terceiro gráfico o cálculo do espectro de frequências calculado pelo Matlab®. Igualmente à situação anterior, os espectros de frequências são semelhantes no quesito amplitude e local da frequência principal do sinal.

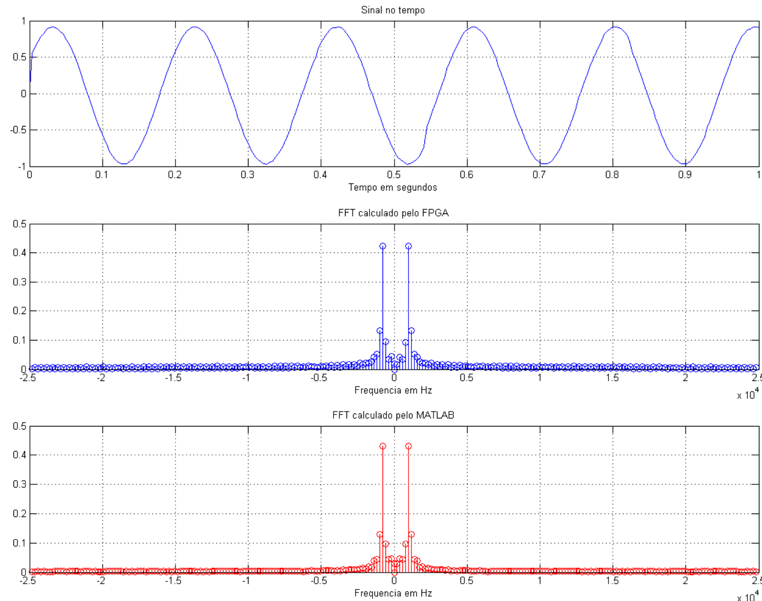


Figura 4.6 - Sinal original em forma de senoide, FFT calculada pelo FPGA e FFT calculada pelo Matlab®.

Foram realizados testes com diversas frequências e formas de ondas diferentes. A Figura 4.7 mostra os resultados obtidos em um caso de teste com uma onda quadrada de 1KHz. A Figura 4.8 mostra os resultados obtidos em um caso de teste com uma senoide de 10KHz e a Figura 4.9 mostra

os resultados obtidos em um caso de teste com uma senoide de 20KHz.

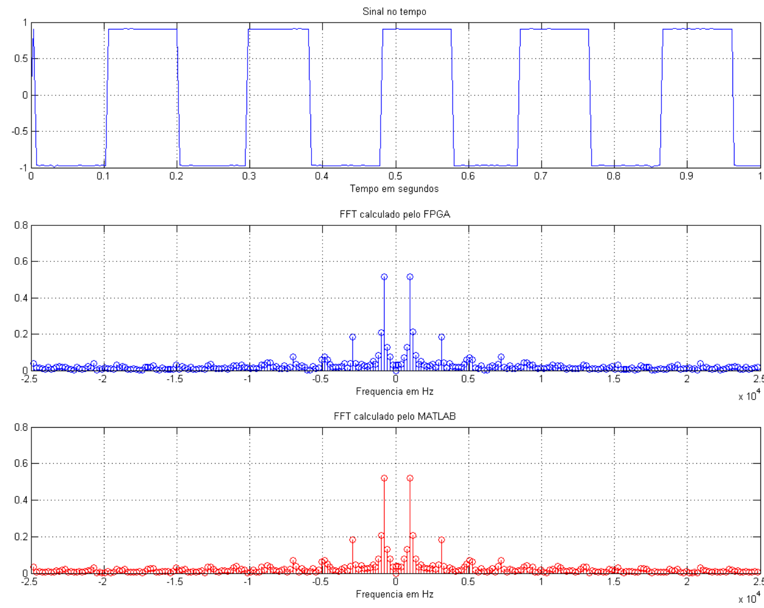


Figura 4.7 - Sinal original de onda quadrada de 10Khz, FFT calculada pelo FPGA e FFT calculada pelo Matlab®.

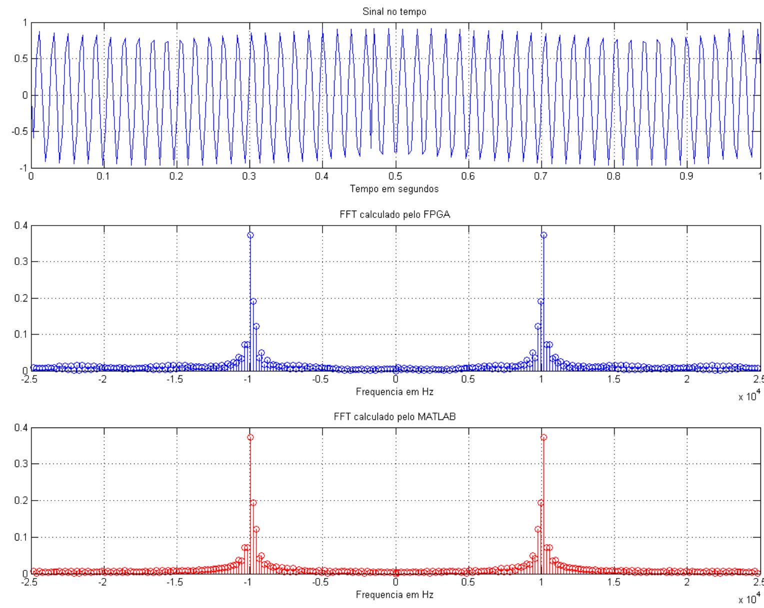


Figura 4.8 - Sinal original de senoide de 10KHz, FFT calculada pelo FPGA e FFT calculada pelo Matlab®.

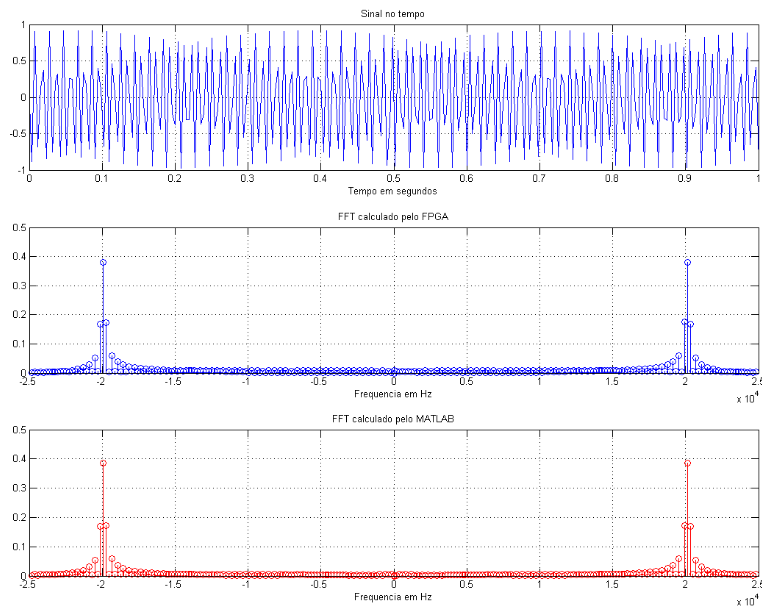


Figura 4.9 - Sinal original de senoide de 20KHz, FFT calculada pelo FPGA e FFT calculada pelo Matlab®.

4.1.3 Situação 3

Com o sistema funcionando corretamente, partiu-se para o caso de teste final, onde temos a apresentação dos dados realizada no próprio FPGA. Assim como no caso anterior, o TOWER realiza a aquisição dos dados e envia para o FPGA pela rede. Porém, o FPGA ao invés de sempre enviar o resultado pela rede, ele apenas os exibe na sua tela de LCD. O diagrama de fluxo de dados pode ser visto na Figura 4.10.

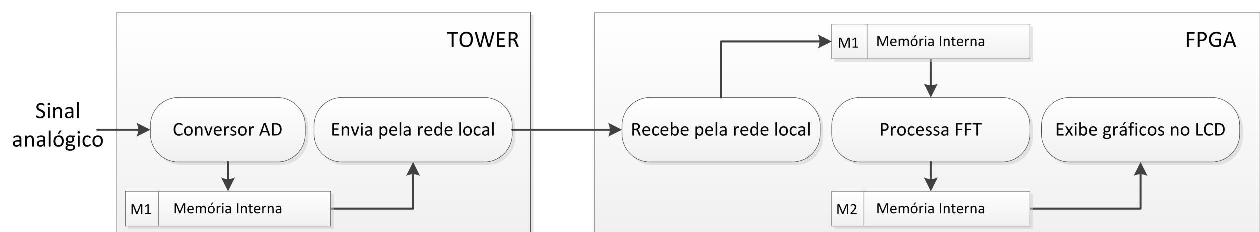
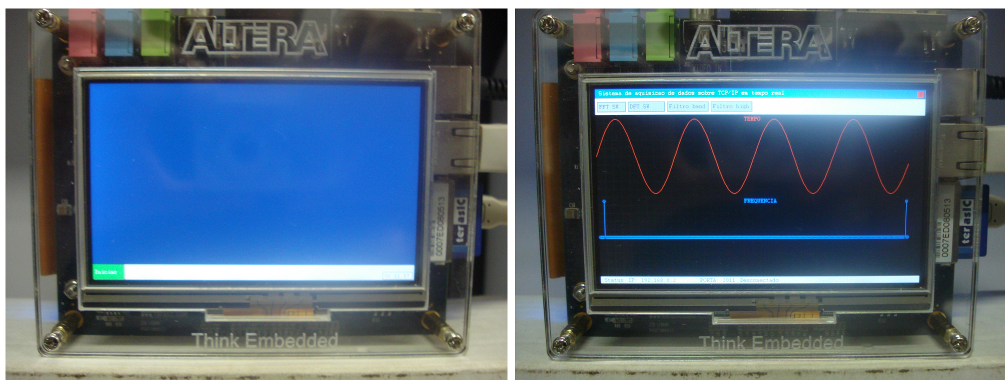


Figura 4.10 - Diagrama de fluxo de dados do teste realizado na terceira situação.

Com o FPGA e o *software* de gerenciamento de telas desenvolvido, foram criadas telas que

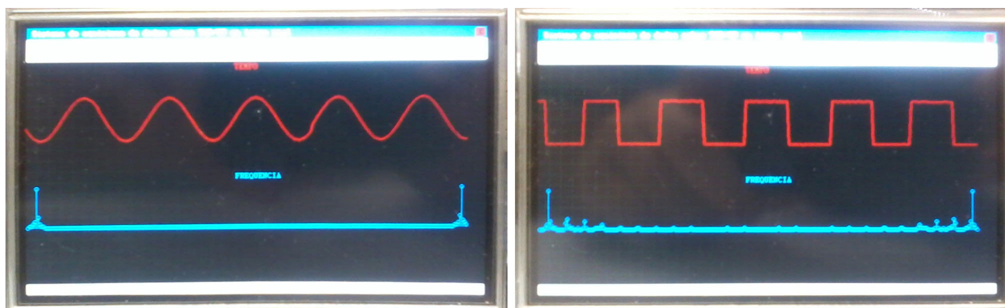
funcionam como pequenos aplicativos do sistema, mostrado na Figura 4.11.



(a) Área de trabalho (b) Aplicativo de aquisição

Figura 4.11 - À esquerda, tela da área de trabalho com o atalho para as demais telas. À direita, a tela do aplicativo de aquisição de dados.

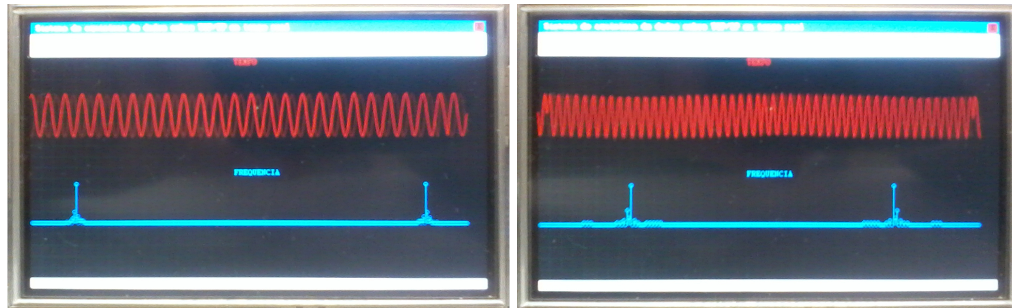
Utilizando o gerador de funções, diferentes formas de onda foram geradas. Na Figura 4.12(a), o sinal gerado foi uma senoide de 1KHz. Como não há uma quantidade exata de períodos no sinal no tempo, houve o efeito de vazamento (*leakage*) no espectro de frequência. Já na Figura 4.12(b), trata-se de uma onda quadrada de 1Khz. Nos gráficos mostrados pelo Matlab® foi realizada a organização dos vetores de frequência para que exista coerência na visualização, ou seja, a primeira metade corresponde ao espelhamento da segunda metade, gerando as frequências negativas teóricas. No FPGA esta reorganização do vetor não foi realizada, logo, a segunda metade do vetor corresponde ao espelhamento da primeira metade.



(a) Onda senoidal de 1KHz (b) Onda quadrada de 1KHz

Figura 4.12 - Diferentes espectros de frequências para diferentes formas de onda, ambas com frequência de 1KHz.

A Figura 4.13(a) apresenta os resultados obtidos com um sinal senoidal de 5KHz e a Figura 4.13(b) um sinal senoidal de 10KHz.



(a) Onda senoidal de 5KHz (b) Onda quadrada de 10KHz

Figura 4.13 - Foto do NEEK com sinal senoidal de 5KHz e de 10KHz.

4.2 Tempo de execução

O tempo de execução do processo pôde ser estimado através de lógica simples, apenas observando a regra de passagem dos pontos do vetor através do *pipeline*, onde foi estimado um tempo de processamento de 3.9 microssegundos a 100MHz. Contudo, o tempo total do processamento inclui o comando de início e a leitura do sinal de fim do processamento. O processador NIOS 2 foi utilizado na sua versão mais simples, o que aumenta o tempo de execução das instruções, aumentando também o tempo utilizado na movimentação dos dados. O tempo total foi medido através do periférico medidor de desempenho. A Tabela 4.1 apresenta o tempo gasto com o envio dos dados à memória RAM interna, com a execução do processamento e com a leitura dos dados resultantes na memória RAM interna. Foi desconsiderado em todos os casos o tempo de início e de término do medidor de desempenho, que são incluídos automaticamente por este periférico.

Tabela 4.1 - Tempos médios do processamento a 100MHz.

Etapa	Tempo (ms)	Tempo(clocks)
Envio dos dados	3.89	388900
Execução da FFT	0.004	440
Leitura dos dados	5.49	548828

Para efeito de comparação, o tempo de execução de cada etapa da versão totalmente em *software* do algoritmo também foi medida e encontra-se no Apêndice D. Na versão em *software* do algoritmo da FFT, o processador NIOS 2 foi utilizado na versão rápida, com multiplicação de

ponto flutuante habilitada em *hardware*. Em função da grande quantidade de elementos lógicos utilizados para executar tal tarefa, não foi possível colocar o periférico FFT neste processador para ser testado simultaneamente. Neste caso, o processador foi gerado sem o periférico da FFT apenas para realizar as comparações de desempenho.

4.3 Utilização de elementos lógicos

Na primeira e segunda situação de teste, onde o processador utilizado foi o NIOS 2 econômico, a quantidade de elementos lógicos utilizado pelo sistema pode ser vista na Tabela 4.2. A Tabela 4.3 apresenta maiores detalhes na quantidade total de elementos lógicos utilizados nos principais periféricos do processador.

Tabela 4.2 - Utilização dos elementos lógicos do FPGA.

Descrição	Elementos utilizados	Elementos totais	%
Elementos lógicos	20,192	24,624	82 %
Bits de memória	189,854	608,256	31 %
Multiplicadores de 9 bits	56	132	42%

Tabela 4.3 - Utilização de elementos lógicos pelos principais periféricos do processador.

Componente	LE Comb.	LE Reg.	Bits Memória	Multiplicadores 18x18
CPU	943	513	10240	0
FFT Pipeline	3027	3059	82080	28
TSE MAC	2956	3109	77312	0
SGDMA Tx	642	902	2500	0
SGDMA Rx	460	783	128	0
LCD SGDMA	625	907	0	0
SDRAM Controller	2502	1833	2880	0

5 Conclusões e Perspectivas

Neste trabalho foi desenvolvida uma arquitetura para processamento digital de sinais em FPGA, composta de vários módulos de aquisição e de um módulo de processamento e de visualização dos dados através de uma interface homem-máquina. As contribuições principais estão no desenvolvimento do processador juntamente com seu periférico de processamento da FFT, desenvolvido em VHDL. A FFT foi adotada como ferramenta matemática para o processamento digital de sinais.

Para o módulo de processamento e visualização, o processador utilizado foi o NIOS 2, uma propriedade intelectual da Altera® distribuída gratuitamente. Trata-se de um processador totalmente projetado em linguagem de descrição de *hardware* (*soft-core*). Foi desenvolvido e adicionado ao processador um periférico para cálculo da FFT de forma eficiente, que utiliza o algoritmo apresentado. Para comprovar o seu funcionamento foi realizada uma simulação passo a passo (ver Apêndice A), utilizando valores conhecidos como entrada para o *pipeline* e verificou-se que os resultados estavam corretos.

Para realizar testes de funcionamento do periférico foi utilizado, inicialmente, uma versão modificada, onde os dados do resultado foram enviados para um computador pela porta serial. No computador os dados puderam ser analisados, comprovando o correto funcionamento do periférico. Com o periférico funcionando corretamente, foi criada uma versão onde os dados resultantes do processamento ficam armazenados em uma memória interna, disponíveis através de leituras nos endereços de memória da interface Avalon MM escrava existente. A partir de então, os testes foram realizados utilizando apenas a comunicação através da rede local, onde os resultados alcançados foram os mesmos.

No FPGA onde foi desenvolvido o processamento, também foi desenvolvida a visualização dos dados processados através da interface homem-máquina, uma tela de LCD. O *software* teve a capacidade de comunicar com os demais dispositivos da rede, receber os dados, processar em *hardware* e exibir os resultados na tela. Por isto, o *software* foi dividido em dois módulos: processamento e visualização. Foi utilizado o sistema operacional de tempo real uC/OS2, que permitiu a criação de várias tarefas, onde uma delas ficou responsável por gerenciar a interface. Para tanto, foi criado um sistema de gerenciamento de telas, utilizando apenas linguagem C.

Para o módulo de aquisição de dados foi utilizado um *kit* de desenvolvimento da FreeScale Semiconductors® que possui um microcontrolador *Coldfire*. Foi utilizado o sistema operacional de tempo real MQX, para que a tarefa de comunicação em rede possa ser executada concorrentemente à tarefa de aquisição de dados.

Os resultados obtidos no Capítulo 4 mostraram que este sistema tem aplicabilidade em um problema real. O pico no espectro de frequência correspondeu em todos os casos de teste ao valor correto da frequência do sinal. Apenas a amplitude sofreu pequenas alterações em relação ao mesmo processamento com o Matlab®, em função da utilização de aritmética de ponto fixo no FPGA.

Os resultados foram obtidos através de 3 diferentes casos de testes. No primeiro, os dados originais estavam armazenados em uma RAM interna, e após processados foram enviados pela porta serial para serem analisados por um computador com o software Matlab®. No segundo caso de teste, os dados foram gerados por um gerador de funções e adquiridos pelo TOWER, que os enviou pela rede ao FPGA. O FPGA processou estes dados e os enviou para serem analisados em um computador com o software Matlab®. No terceiro caso, o sinal foi gerado num gerador de funções, adquirido pelo TOWER e enviado ao FPGA, que os processou e exibiu em sua tela de LCD. Em todos os casos de testes realizados o cálculo foi preciso no quesito pico de frequência, porém houveram pequenas variações na amplitude do sinal.

A quantidade de elementos lógicos utilizados pelo periférico FFT também foi considerada mínima em função da quantidade de pontos processados. Porém o FPGA escolhido oferece apenas 24 mil elementos lógicos e 600 mil bits de memória interna, o que impossibilitou realizar os testes com um periférico que utilize mais que 256 pontos. Isto ocorreu em função do FPGA ter que abrigar além do periférico FFT diversos outros, como por exemplo, o controlador de memória SDRAM, o controlador de rede e seus 2 SGDMAs, o processador NIOS 2, os periféricos relacionados com a geração de imagem no LCD. Além da facilidade de aumentar a quantidade de pontos no processamento, outra vantagem do algoritmo proposto é que mais de um FPGA podem ser utilizados para a execução do *pipeline*, desde que os sinais sejam interligados corretamente e se respeite o limite de velocidade do FPGA. Isto garantiria que uma quantidade muito grande de pontos fossem processados pelo *pipeline*. Outro avanço seria o paralelismo de *pipelines* em um mesmo periférico, aumentando a utilização dos multiplicadores em *hardware*.

A velocidade de processamento está diretamente relacionada aos pulsos de *clock*. Logo,

quanto maior for a sua frequência, maior a velocidade de processamento. Porém, deve ser feito uma análise da propagação dos sinais através do FPGA utilizando ferramentas oferecidas pelo ambiente de desenvolvimento para que seja encontrada a velocidade máxima permitida no sistema.

Em termos de números de *clocks*, foram necessários 390 pulsos para calcular 256 pontos. Ou seja, como o sistema está trabalhando a 100MHz, o cálculo é realizado em 3.9 microssegundos. Trata-se de um ganho de cerca de 2000 vezes quando comparado com a versão do algoritmo implementada em linguagem C, utilizando tabela de senos e cossenos pré-calculados e instruções de ponto flutuante habilitadas no núcleo do processador NIOS 2 versão rápida, que foi executado em aproximadamente 8 milissegundos. Esta relação justifica o uso de um módulo implementado em *hardware* para o cálculo da FFT. Contudo, este tempo mínimo refere-se somente ao processamento, e no Capítulo 4 foi apresentada uma comparação mais abrangente.

Assim como apresentado na introdução do trabalho, percebe-se a potencialidade de um sistema embarcado baseado em FPGA processar um sinal digital de forma ótima através de implementações de *hardware* e *software*, obtendo redução de tempo de execução, custo e consumo de energia. Logo a metodologia desenvolvida neste trabalho se mostrou eficaz para realizar o processamento digital de sinais e certamente será a base para estudos conduzidos futuramente, como uma consequência natural do que foi realizado até o momento.

5.1 Perspectivas

Esta arquitetura de processamento digital de sinais baseada em FPGA pode ser aplicada nos mais diversos campos, da área médica à área de vibrações. Além disto, mesmo sendo construído com *kits* de desenvolvimento, o sistema tem potencial para evoluir e se tornar um produto de mercado.

Uma aplicação na área médica é a monitoração de pacientes em um leitos hospitalares, onde diversos dispositivos de aquisição de dados espalhados e conectados em rede enviam as informações coletadas dos pacientes para uma central (o FPGA), que processa e toma uma decisão baseado neste resultado. Neste caso, o trabalho desenvolvido por Raizer (2010) e Tomazati (2009), respectivamente, "Análise de sinais de ECG com o uso de Wavelets e Redes Neurais em FPGA"e

"Detecção do complexo QRS em sinais cardíacos utilizando FPGA", podem ser integrados em um único FPGA para o monitoramento dos pacientes.

Um exemplo de aplicação na área de vibrações é o monitoramento de integridade de estruturas metálicas. Neste exemplo utilizam-se sensores para captar a vibração da placa para identificar possíveis alterações na integridade da estrutura. Muitos algoritmos utilizados nos métodos apresentados por Genari e Nobrega (2012), Junior (2011) podem ser convertidos para circuitos e implementados diretamente em um FPGA, criando a possibilidade de executar o monitoramento em tempo real.

Outro exemplo é o reconhecimento de voz, citado na introdução, onde é aplicada a transformada de Fourier em *hardware* e os demais processos são realizados em *software*. Este caso está muito próximo de ser atendido, visto que neste trabalho foi desenvolvida uma transformada de Fourier em *hardware*. Poderia ser facilmente aplicado em sistemas de automação que necessitem de acionamento por voz.

Muitas são as aplicações possíveis para esta arquitetura o que garante sua aplicabilidade. Outras ferramentas de processamento de sinais podem ser implementadas, como a transformada Wavelet e filtros digitais. As ferramentas matemáticas não se restringem a sinais de uma dimensão como foi realizado neste trabalho. Pagano et al. (2011) propõem a implementação de um filtro digital de imagens diretamente em *hardware* através de um periférico que pode ser adicionado ao processador NIOS2.

Para melhorar o desempenho do sistema de aquisição, é necessário utilizar uma rede local do tipo "Gigabit", que oferece altas taxas de transmissão (acima de 1 Gbps). Assim, o gargalo da comunicação de dados pode ser eliminado. Outra melhoria significativa no desempenho seria a utilização de um FPGA com mais recursos de *hardware*, como o Arria V e o Stratix V, ambos fabricado pela Altera®(ALTERA, 2011b; ALTERA, 2011m).

REFERÊNCIAS

ALTERA, C. **Altera Avalon Interface Specifications**. [S.l.], 2011. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.altera.com/literature/manual/mnl_avalon_spec.pdf>.

____. **Arria FPGA Series**. 2011. Altera Corporation Website. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.altera.com/devices/fpga/arria-fpgas/about/arr-about.html>.

____. **Embedded Peripherals IP User Guide**. [S.l.], 2011. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.altera.com/literature/ug/ug_embedded_ip.pdf>.

____. **Implementing an LCD Controller**. [S.l.], jul. 2011. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.altera.com/literature/an/an527.pdf>.

____. **Introduction to Quartus II 10.0 Software**. [S.l.], 2011. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.altera.com/literature/manual/archives/intro_to_quartus2.pdf>.

____. **Introduction to VHDL**. 2011. Altera Corporation Website Online Course. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.altera.com/education/training/courses/IHDL110?GSA_pos=1&WT.oss_r=1&WT.oss=vhdl>.

____. **MicroC/OS-II Real-Time Operating System**. [S.l.], 2011. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.altera.com/literature/hb/nios2/n2sw_nii52008.pdf>.

____. **Nios II Embedded Evaluation Kit, Cyclone III Edition**. 2011. Altera Corporation

Website. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.altera.com/products/devkits/altera/kit-cyc3-embedded.html>.

____. **Nios II Embedded Evaluation Kit, Cyclone III Edition User Guide**. [S.l.], 2011. Disponível em: <http://www.altera.com/literature/ug/niosii_eval_user_guide.pdf>.

____. **Nios II Embedded Evaluation Kit Picture Viewer Example**. [S.l.], 2011. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.alterawiki.com/wiki/NEEK_Picture_Viewer>.

____. **Nios II Processor reference Handbook**. [S.l.], 2011. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf>.

____. **SOPC Builder User Guide**. [S.l.], 2011. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.altera.com/literature/ug/ug_sopc_builder.pdf>.

____. **Stratix Series High-End FPGAs**. 2011. Altera Corporation Website. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.altera.com/devices/fpga/stratix-fpgas/about/stx-about.html>.

____. **VHDL Basics**. 2011. Altera Corporation Website Online Course. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.altera.com/education/training/courses/OHDL1110?GSA_pos=2&WT.oss_r=1&WT.oss=vhdl>.

____. **Video Sync Generator and Pixel Converter Cores**. [S.l.], 2011. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.altera.com/literature/hb/nios2/qts_qii55006.pdf>.

____. **What is an FPGA?** 2011. Altera Corporation Website. Acesso em 15 de dezembro de

2011. Disponível em: <http://www.altera.com/products/fpga.html>.

ANTONIOU, A. **Digital Signal Processing, signals, system and filters**. New York, United States of America: McGraw-Hill, 2006. 991 p.

BOUDABOUS, A. et al. Fpga implementation of vector directional distance filter. **International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS).**, v. 1, p. 1–4, 2005.

____. Hw/sw fpga implementation of vector median filter. **Research in Microelectronics and Electronics Conference**, v. 1, p. 101–104, 2007.

CARDOSO, F. C. M. A.; ARANTES, D. S. **Simulação, Co-simulação e Prototipagem de Sistemas de Comunicações Digitais**. 2007. Curso IE344 B. Acesso em 01 de fevereiro de 2012. Disponível em: <http://www.decom.fee.unicamp.br/~cardoso/ie344b.html>.

COOLEY, J. W.; TUKEY, J. W. An algorithm for the machine computation of complex fourier series. **Mathematics of Computation**, v. 19, p. 297–301, abr. 1965.

FREESCALE, S. **Freescale MQX Software Solutions**. 2011. FreeScale MQX Website. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.freescale.com/mqx>.

____. **MCF51CN128 Reference Manual**. [S.l.], 2011. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.freescale.com>.

____. **MQX RTOS**. 2011. Embedded Access Wikipedia page. Acesso em 15 de dezembro de 2011. Disponível em:
<http://en.wikipedia.org/wiki/User:Gaxis/Draft_MQX_RTOS_Page>.

GENARI, H. F. G.; NOBREGA, E. G. O. A damage detection technique based on ARMA models distance estimation. In: **1st International Symposium on Uncertainty Quantification and**

Stochastic Modeling. São Sebastião, Brazil: [s.n.], 2012.

IAR, S. **Basic concepts for Real Timer Operating Systems**. 2012. IAR Systems Website. Acesso em 01 de fevereiro de 2012. Disponível em: <http://marketing.iar.com/acton/ct/1011/s-019b-1201/Bct/l-sf-rpt-00O3000000545G6-0120/l-sf-rpt-00O3000000545G6-0120:13eac/ct21_0/1>.

JACKSON, P. A. et al. A systolic fft architecture for real time fpga systems. **In High Performance Embedded Computing Conference**, p. 243–263, feb 2004.

JUNIOR, N. R. I. **Estudo de Métodos de Identificação Multivariável baseados em Subespaços aplicados ao Monitoramento da Integridade de Estruturas**. 145 p. Dissertação — Universidade Estadual de Campinas,, jul. 2011.

KUNG, Y.-S.; TSUI, T.-W.; SHIEH., N.-H. Design and implementation of a motion controller for xyz table based on multiprocessor sopc. **Chinese Control and Decision Conference.**, v. 1, p. 241–246, 2009.

LABROSSE, J. J. **MicroC/OS-II the real time kernell**. Second. New York, United States of America: CMPBooks, 2002. 606 p.

LI, W. **Studies on implementation of low power FFT processors**. 150 p. Dissertação — Linköping Studies in Science and Technology, Linköping, Sweden, Sweden, jun. 2003.

LIAO, Y.-P.; ZHOU, H.-G.; FAN., G.-R. Accelerating recognition system of leaves on nios ii embedded platform. **International Symposium on Computer, Comunication, Control and Automation**, v. 1, p. 334–337, 2010.

MAXFIELD, C. **The design warrior's guide to FPGAs: devices, tools and flows**. Newnes, 2004. (Edn Series for Design Engineers, v. 1). ISBN 9780750676045. Disponível em:

<http://books.google.com.br/books?id=dnuwr2xOFpUC>.

MAYYA, M.; ZARKA, N.; ALKAD, S. Embedded system for real-time human motion detection. **Image Processing Theory Tools and Applications (IPTA), 2010 2nd International Conference on**, v. 1, p. 523 –528, 2010.

MELLO, C. A. **Processamento digital de sinais**. 2011. Apostila. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.cin.ufpe.br/~cabm/pds/PDS.pdf>.

MENTOR, G. C. An introduction to modeling in vhdl. Curso VHDL Mentor Graphics, intructor notes. nov. 1991.

MICRIUM. **MicroC OS-II Kernel**. 2011. Micrium MicroC OS II Website. Acesso em 15 de dezembro de 2011. Disponível em:
<http://www.micrium.com/page/products/rtos/os-ii>.

MOREIRA, V. R. **Plataforma em hardware reconfigurável para o ensino e pesquisa em laboratório de sistemas digitais a distância**. 103 p. Dissertação — Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, Campinas, 2009.

PAGANO, D. M. et al. Aplicação da convolução 2d para processamento de imagens utilizando fpga. **CILAMCE 2011**, nov 2011.

PEDRONI, V. A. **Circuit Design with VHDL**. [S.l.]: MIT Press, 2004.

PERRY, D. L. **VHDL Programing by example**. [S.l.]: McGraw-Hill, 2002.

PROAKIS, J. G.; MANOLAKIS, D. G. **Digital Signal Processing, Principles, algorithms and applications**. Third. New Jersey, United States of America: Prentice-HAll, 1996. 1033 p.

RABINER, L.; GOLD, B. **Theory and application of digital signal processing**. Prentice-Hall,

1975. (Prentice-Hall signal processing series). ISBN 9780139141010. Disponível em: <http://books.google.com.br/books?id=iAxTAAAAMAAJ>.

RAIZER, K. **Analise de sinais de ECG com o uso de Wavelets e Redes Neurais em FPGA**. Dissertação (Mestrado) — UNICAMP, Universidade Estadual de Campinas, Campinas, Brasil, 2010.

RAMAKRISHNA, M.; KUMAR, K.; RAMESH, A. P. Hardware acceleration of histogram equalization and image sharpening filter on nios-ii processor based soc on fpga. **International Journal of Computer Applications**, v. 31, n. 1, p. 48–54, October 2011. Published by Foundation of Computer Science, New York, USA.

SEBESTA, R. W. **Conceitos de linguagens de programação**. BOOKMAN COMPANHIA ED, 2002. ISBN 9788536301716. Disponível em: <http://books.google.com.br/books?id=b0tcn_uPLoAC>.

SHAW, A. C. **Sistemas e software de tempo real**. Washington, United States of America: Bookman, 2003. 240 p.

SKLIAROVA, I.; FERRARI, A. B. **Introdução à Computação Reconfigurável**. 2003. Revista do DETUA. Acesso em 15 de dezembro de 2011. Disponível em: <http://www.ieeta.pt/~iouliia/Papers/2003/1_SF_ETSet2003.pdf>.

TOMAZATI, A. **Detecção do complexo QRS em sinais cardíacos utilizando FPGA**. Dissertação (Mestrado) — UNICAMP, Universidade Estadual de Campinas, Campinas, Brasil, 2009.

WEI, W.; GUIDONG, Z. The design and implementation of high-speed data acquisition system based on nios ii. **Computing, Control and Industrial Engineering (CCIE), 2010 International Conference on**, v. 2, p. 334–336, june 2010.

YANG, Y. et al. Software design of sd card reader and image processor based on fpga. **International Conference on Mechatronic Science, Electric Engineering and Computer**, v. 1, p. 1864 – 1867, 2011.

YATES, R. **Fixed-Point Arithmetic: An Introduction**. ago. 2007. Digital Signal Labs Website. Acesso em 01 de fevereiro de 2012. Disponível em: <http://www.digitalsignallabs.com>.

ZHANG, G. et al. A real-time speech recognition system based on the implementation of fpga. **Cross Strait Quad-Regional Radio Science and Wireless Technology Conference**, v. 2, p. 1375 – 1378, july 2011.

APÊNDICE A - FUNCIONAMENTO DO ALGORITMO FFT

Este apêndice apresenta um passo a passo para a realização do algoritmo proposto, onde são apresentados todos os elementos de *hardware* existentes e os seus respectivos funcionamentos, incluindo todos os sinais existentes no processo.

Este método é utilizado em aplicações onde os dados entram de forma serial no sistema. Para esta proposta, o algoritmo utilizado foi o *Radix-2 multi-path delay commutator* (R2MDC), utilizando dizimação em frequência (DIF), originalmente proposto por Rabiner e Gold (1975).

O cálculo é realizado em várias etapas (chamadas de estágios) que possuem o mesmo funcionamento, baseado apenas em seu índice. Assim pode-se construir um sistema genérico, composto de vários estágios, que utilizam seu índice como parâmetro para calcular a FFT. Desta maneira, para aumentar a quantidade de pontos basta aumentar a quantidade de estágios, segundo a seguinte regra:

$$Nmerodepontos = 2^S \quad \text{sendo S o número de estágios.} \tag{A.1}$$

Assim, o sistema serial de cálculo da FFT, que será chamado de *pipeline*, é composto de vários estágios. A Figura A.1 apresenta a configuração genérica simplificada de um *pipeline*.

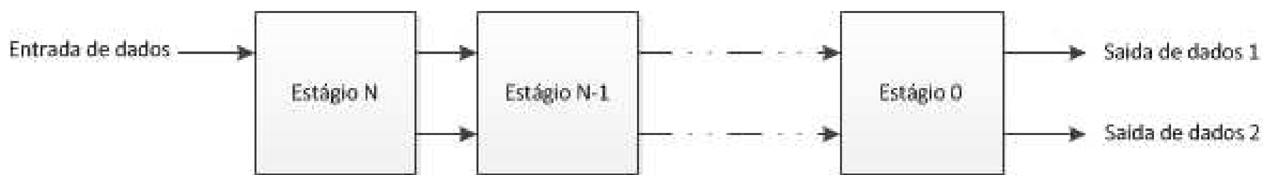


Figura A.1 - Configuração genérica do *pipeline*.

Cada estágio possui duas entradas e duas saídas. Cada entrada ou saída é um número complexo com parte real e imaginária separadas, onde cada uma é formada por um número de ponto fixo no formato S(2, 16), ou seja, dois *bits* para a parte inteira e dezesseis *bits* para a parte fracionária, totalizando em 18 *bits*. No primeiro estágio do *pipeline* (estágio N na Figura A.1), os dados entram apenas na primeira entrada, deixando a segunda entrada aberta, ou seja, recebendo dados de valor zero. As saídas da primeira entrada estão conectadas nas entradas do próximo estágio, e esta configuração se repete até o último estágio do sistema. O último estágio (estágio 0 na Figura A.1), gera no final do *pipeline* dois dados por pulso de *clock*, ordenados segundo o algoritmo

"Bit-reverse".

Cada estágio, além das entradas e saídas, possui também um sinal de entrada de *clock*, entrada de *clear*, entrada de sincronismo *en*, e libera um sinal de sincronismo *en-out*. Estes sinais são necessários para que os estágios realizem o algoritmo a cada pulso de *clock*, e mantenham um sincronismo entre si através dos sinais de sincronismo *en* e *en-out.* O sinal *clear* é utilizado para fazer os estágios voltarem à sua condição inicial. A entrada de sincronismo deve receber nível lógico alto quando o primeiro dado for entrar no sistema, enquanto a saída de sincronismo envia sinal lógico alto quando o primeiro dado processado sair do sistema. A Figura A.2 apresenta o *pipeline* com todos os sinais citados envolvidos.

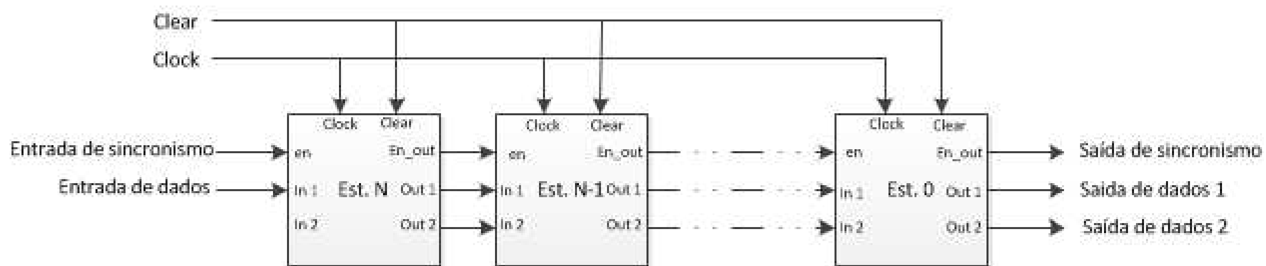


Figura A.2 - *Pipeline* com todos os sinais envolvidos.

Cada estágio apresenta um algoritmo específico para realizar o processamento. Um estágio é composto de:

•Uma fila de 2^S elementos, sendo S o índice do estágio;

•Duas fila de $2^S + 1$ elementos, sendo S o índice do estágio;

•Dois multiplexadores auto-controlados;

•Um registrador;

•Um operador borboleta DIF;

•Um gerador de constantes W ("*twiddle factors*").

Um registrador é um elemento digital que, ao receber um pulso de *clock*, copia o valor de sua entrada de dados para sua saída de dados. Uma fila de elementos é um conjunto de registradores que são conectados de forma serial. Esta conexão entre eles faz com que a cada pulso de *clock* o dado da entrada do registrador se mova para sua saída. Desta maneira, ao utilizarmos diversos registradores interligados de forma serial, estes funcionarão como uma fila, onde a cada pulso de *clock*, o dado se move para a próxima posição da fila em direção à sua saída. Portanto, devido a sua arquitetura, o primeiro elemento a entrar na fila é o primeiro a sair da fila. A Figura A.3 apresenta

um exemplo de fila de quatro elementos.

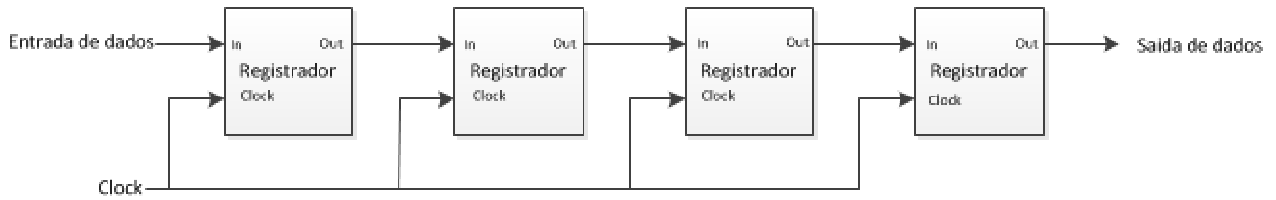


Figura A.3 - Estrutura de uma fila de dados.

Um multiplexador é um elemento digital que seleciona uma de suas entradas e a direciona para sua saída. Neste algoritmo, o multiplexador é auto-controlado através do índice do estágio e dos pulsos de *clock*. A cada pulso de *clock*, um contador interno incrementa. Quando este contador alcança a constante 2^S (onde S é o índice do estágio), o multiplexador troca a entrada selecionada.

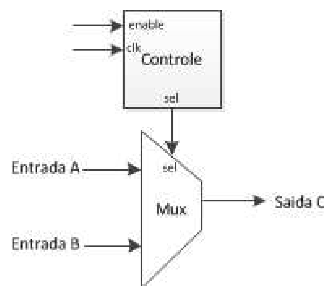


Figura A.4 - Estrutura do multiplexador auto-controlado.

O operador borboleta DIF é a redução da transformada rápida de Fourier segundo o algoritmo DIF (dizimação em frequência) para dois elementos. Esta operação utiliza operações matemáticas de soma, subtração e multiplicação de números complexos implementados em *hardware* utilizando pontos fixos de 18 *bits*.

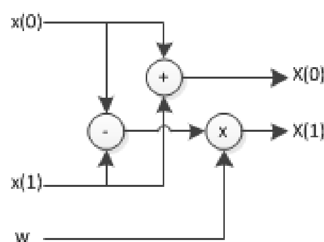


Figura A.5 - Operação borboleta DIF.

O gerador de constantes "W"é um módulo que a cada pulso de *clock* com o sinal *enable* em nível lógico alto, coloca em sua saída o próximo fator a ser utilizado na operação borboleta.

Trata-se de um circuito sequencial complexo que faz uso de um vetor de constantes e utiliza uma sequência de multiplexadores para selecionar o próximo fator a ser utilizado. Ele foi descrito em VHDL, através de comandos sequenciais. Assim como os demais módulos, este também recebe o índice do estágio para buscar o fator correto.

Figura A.6 - Entradas e saídas do gerador de fatores.

De posse de todos estes elementos, partiu-se para a criação do estágio. Este se forma através da conexão entre os elementos existentes. A Figura A.7 apresenta esta configuração de forma genérica e detalhada, onde S indica o índice do estágio.

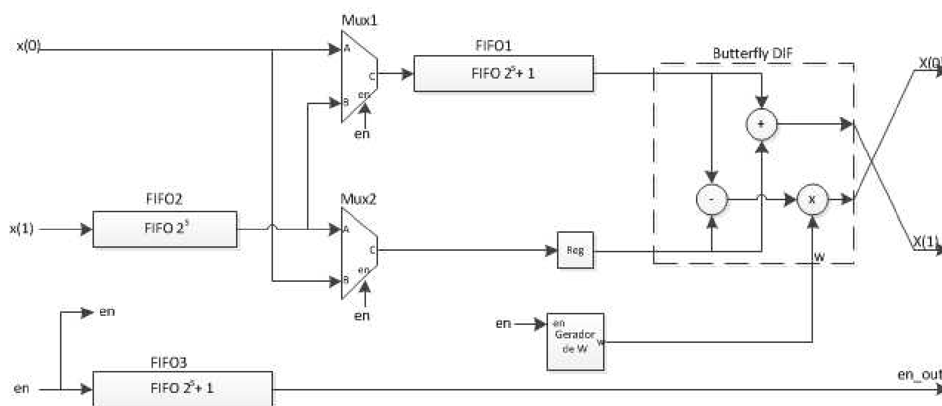


Figura A.7 - Elementos de um estágio.

No primeiro instante, antes do primeiro pulso de clock, o dado x(0) aparece na entrada A do Mux1. O Mux1 está selecionando a entrada A, logo o dado x(0) aparece na entrada da FIFO1. Já o dado x(1) está aguardando na entrada da FIFO2. O sinal de sincronismo *en* está aguardando na entrada da FIFO3, na entrada de sincronismo dos multiplexadores auto-controlados e na entrada de sincronismo do gerador de constantes "W". Ao receber o pulso de *clock*, os dados que estão na entrada das filas entram, e os multiplexadores iniciam sua contagem interna para direcionar o próximo dado para a fila correta. O gerador de W inicia sua tarefa de aplicar o fator correto. O estágio repete esta operação até que o número de pontos daquele estágio é alcançado. Quando isto acontece, os multiplexadores invertem sua seleção dando um novo destino ao dado. A partir deste

ponto, o próximo dado a entrar no estágio será selecionado pelo Mux2 e entrará no registrador Reg. O dado que sai da FIFO2 é selecionado pelo Mux1 e entrará na FIFO1. Ao receber um pulso de *clock*, um dado sairá da FIFO1 e outro do registrador Reg. Estes dados entram na borboleta DIF juntamente com o fator gerado pelo gerador de constantes "W", e o resultado, devido ao aspecto combinacional do cálculo em *hardware* sai quase que instantaneamente nas saídas X(0) e X(1). O procedimento se repete continuamente, enquanto o sinal de sincronismo estiver ativo. Este sinal tem o objetivo de garantir que os multiplexadores e o gerador de constantes W estejam sempre sincronizados com o dado que está entrando no estágio.

A.1 Simulação para oito pontos

Para melhor compreensão do funcionamento do sistema, faz-se necessário uma simulação com o intuito de entender o que está acontecendo em cada passo (pulso de *clock*) do sistema. Foram utilizados oito pontos no vetor de entrada, o que garante fácil compreensão do sistema. Para tanto são necessários três estágios, com os respectivos índices dois, um e zero. A Figura A.8 e a Figura A.9 apresentam uma visão ampla do sistema e uma visão detalhada do sistema (com omissão do *clock* e do *clear* que devem ser conectados a todos os elementos).

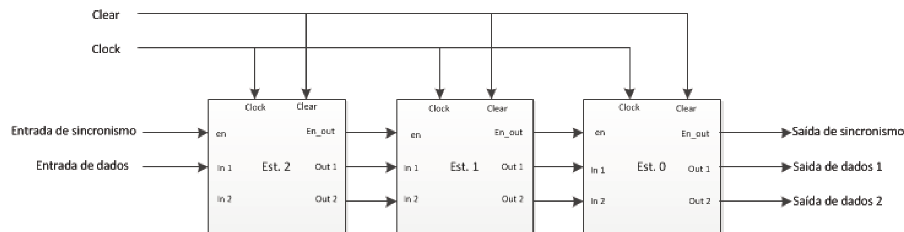


Figura A.8 - Sequência simplificada do *pipeline*.

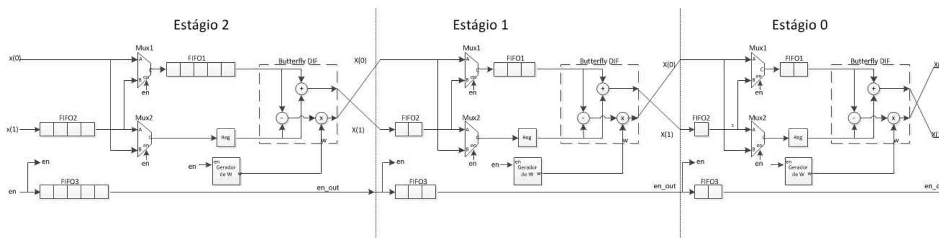


Figura A.9 - Sequência detalhada do *pipeline*.

Contudo, para realizar simulações, faz-se necessário simplificar o esquemático deixando-o mais legível e simples de simular. Neste diagrama, o multiplexador é omitido, deixando apenas os possíveis caminhos para os dados. Todo o sincronismo também é omitido, com a intensão de diminuir o número de sinais controlados manualmente. A operação borboleta DIF foi simplificada a um círculo simples. No primeiro estágio (estágio 2) a FIFO2 não é utilizada, pois os dados entram somente pela entrada 1, logo ela também foi omitida. A Figura A.10 apresenta o novo diagrama utilizado para as simulações.

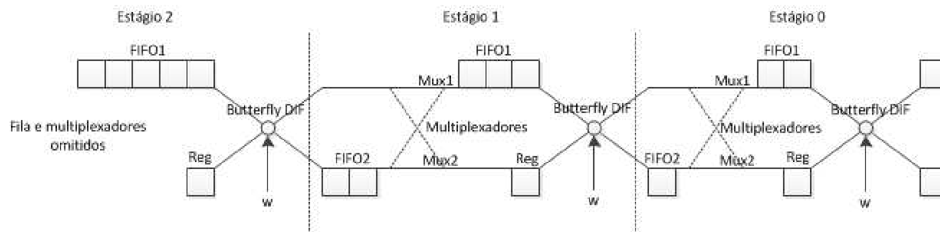


Figura A.10 - Simplificação para facilitar a simulação.

Diante deste cenário simplificado, temos na condição inicial todas as filas vazias, os multiplexadores em condição normal e o W inicial nas operações borboleta DIF. Ao receber os quatro primeiros pulsos de *clock*, com a entrada de sincronismo *en* em nível lógico alto, temos que os quatro primeiros elementos do vetor de dados entrarão na FIFO1 do primeiro estágio. A Figura A.11 apresenta estes passos.

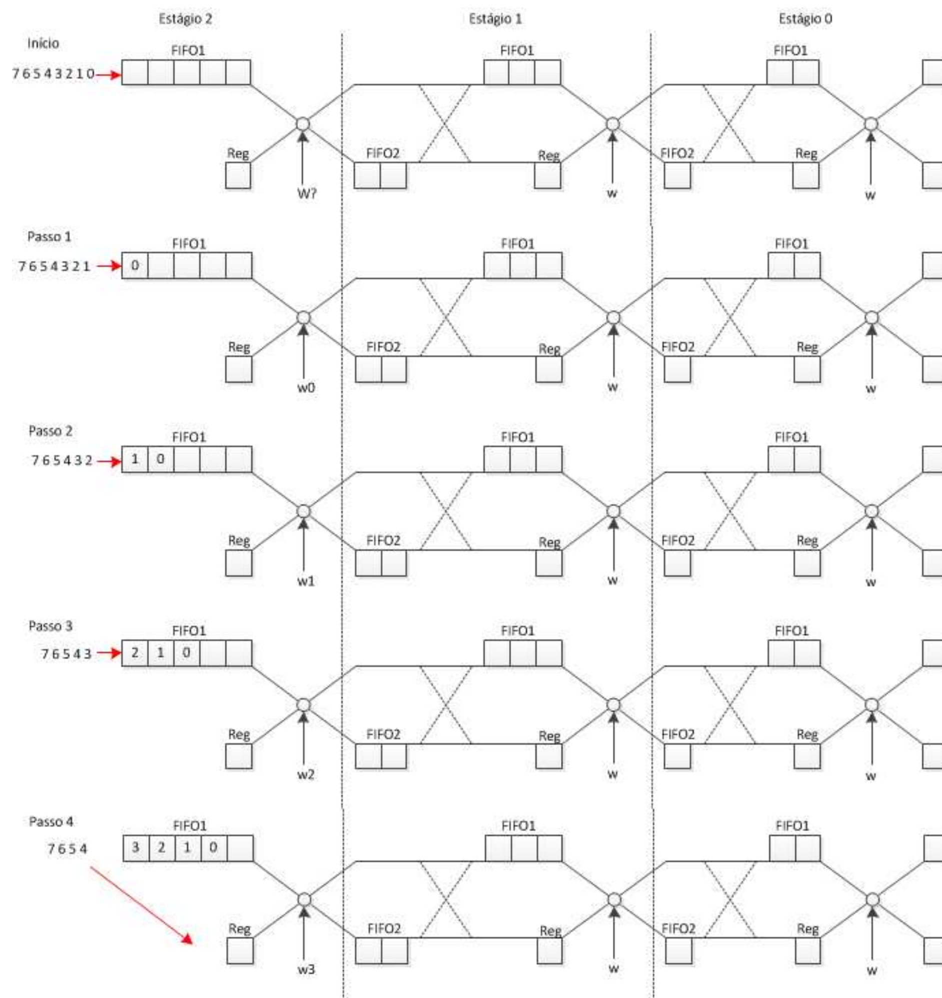


Figura A.11 - Passos 0 a 4 a simulação.

Neste instante o multiplexador 1 do estágio 2 (omitido na Figura A.11) inverte sua seleção, entrando valores 0 na FIFO1, e o multiplexador 2 do estágio 2(omitido na Figura A.11) inverte sua seleção, fazendo com que os próximos dados entrem diretamente no registrador do estágio 2. O gerador de constantes volta ao primeiro valor a gerar e a operação borboleta DIF, pela sua característica combinacional, executa o calculo necessário, e o resultado é direcionado para a saída do estágio 2. Estes dados aparecem na entrada da FIFO2 do estágio 1 e na entrada do multiplexador 1 do estágio 1. Este multiplexador está selecionando esta entrada fazendo com que este dado apareça na entrada da FIFO1 do estágio 1. O sinal de sincronização sai do estágio juntamente com o primeiro dado, logo o contador do multiplexador auto-controlado e do gerador de constantes W dá inicio à sua contagem.

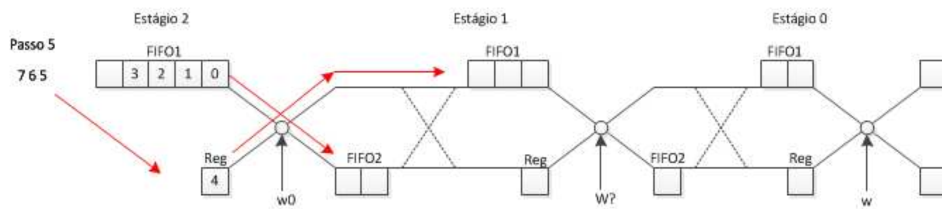


Figura A.12 - Passo 5 da simulação.

Feito isso, o próximo dado entra no registrador do estágio 2, enquanto que o resultado da operação anterior é armazenado no estágio seguinte, cada qual em uma fila diferente. Como o elemento 5 do vetor entrou no registrador do estágio 1, o cálculo da borboleta é executado entre ele e o elemento que está saindo da FIFO1 do estágio 2. Este cálculo, assim como no passo anterior é direcionado para a FIFO1 do estágio 1, empurrando o dado anterior para a próxima posição da FIFO1. Observe que a geração do W no estágio 1 continua acontecendo, enquanto que nos demais estágios ainda não houveram alterações em função do sinal de sincronismo ainda não ter chegado.

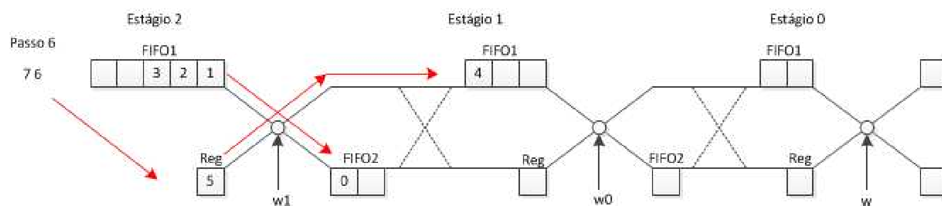


Figura A.13 - Passo 6 da simulação.

Neste próximo passo, o elemento 6 dos dados iniciais entra no sistema igualmente ao elemento 5. Ele realiza a operação borboleta com o elemento 2 da FIFO1 do estágio 2. No *clock* anterior, devido a contagem do multiplexador auto-controlado, houve a inversão dos dados selecionados nos multiplexadores do estágio 1. Logo, acontecerá uma inversão na sequência dos dados. Com isto, o elemento 6 ao invés de entrar na FIFO1 do estágio 1, entra no registrador deste estágio, enquanto que a FIFO2 está cheia e o primeiro dado que entrou, o dado 0, está sendo direcionado pelos multiplexadores para a FIFO 1. Observe a contagem do gerador de W no estágio 1 e no estágio 2.

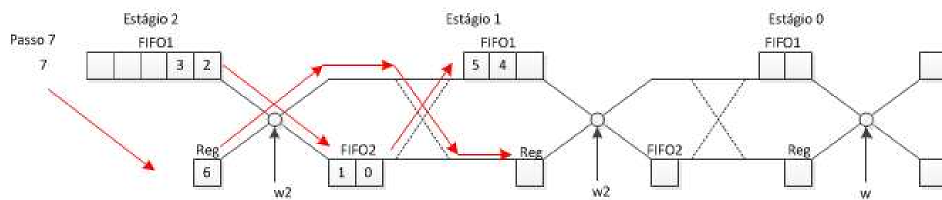


Figura A.14 - Passo 7 da simulação.

No passo 8 muitos eventos ocorrem simultaneamente dentro do *pipeline*. O elemento 7 dos dados iniciais entra no sistema e realiza a operação borboleta com o elemento 3. O elemento 6 que estava no registrador foi selecionado pelo multiplexador 2 do estágio 1 e entrou no registrador deste estágio. A FIFO1 do estágio 1 liberou o primeiro dado, o elemento 4. Com isto entram na borboleta do estágio 1 o elemento 4 e o elemento 6, e são direcionados para o próximo estágio. Juntamente com estes dados, o sinal de sincronismo, não apresentado na simulação, chega ao último estágio, fazendo com que o gerador de constantes e os multiplexadores iniciem suas contagens para a inversão dos dados no momento correto. No estágio 1, a FIFO 2 libera o elemento 1 para entrar ser selecionado pelo multiplexador 1 e este dado entrará na FIFO1 deste estágio. A partir deste passo, a entrada de sincronismo do *pipeline* é desabilitada, e quando a saída de sincronismo for habilitada, significa que o primeiro dado deverá entrar na memória para ser armazenado. A partir de então serão necessários a metade do número de pontos em quantidade de *clocks* para que todos os pontos saiam do sistema em função da saída de dois dados. Logo, a saída de sincronismo só voltará a ter valor zero após passados o número de pontos em quantidade de *clocks*.

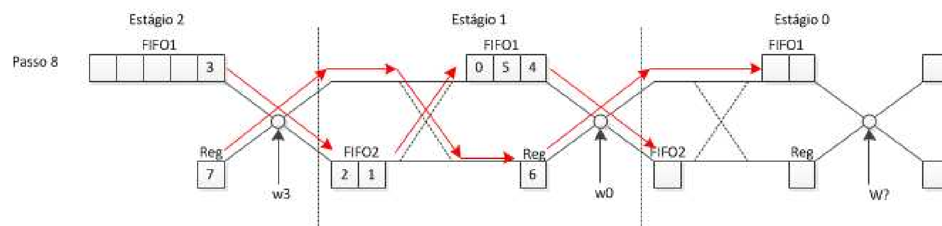


Figura A.15 - Passo 8 da simulação.

No passo seguinte, o passo 9, muitos eventos ocorrem simultaneamente no *pipeline*. No estágio 1, o dado 7 realiza a operação borboleta com o dado 5, que está saindo da FIFO1 do estágio 1. Observe a constante W gerada corretamente. Observe que o elemento 7 resultante da operação borboleta será direcionado pelos multiplexadores do estágio 0 para o registrador. Observe que o elemento 4 presente na FIFO2 do estágio 0 entrará na FIFO1 devido aos multiplexadores.

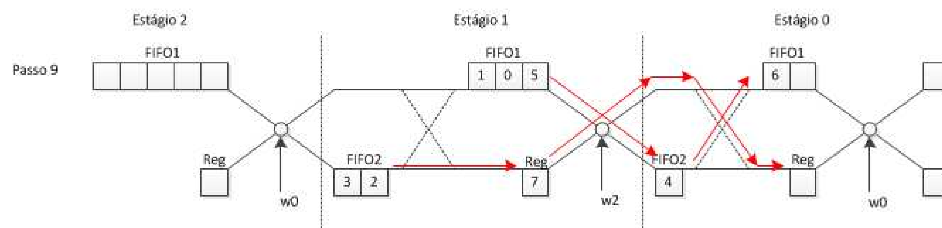


Figura A.16 - Passo 9 da simulação.

Neste passo, o passo 10, temos nossa primeira saída. Isso ocorreu pois a FIFO1 do estágio 0 colocou o elemento 6 na sua saída enquanto que o registrador deste estágio armazenou o dado que estava em sua entrada, o elemento 7. A saída de sincronismo liberou um sinal lógico alto, indicando o primeiro dado válido. Observe que todo o restante do *pipeline* funcionou como o algoritmo até aqui explicitado, deixando bem claro quais serão os próximos dados a saírem do *pipeline*.

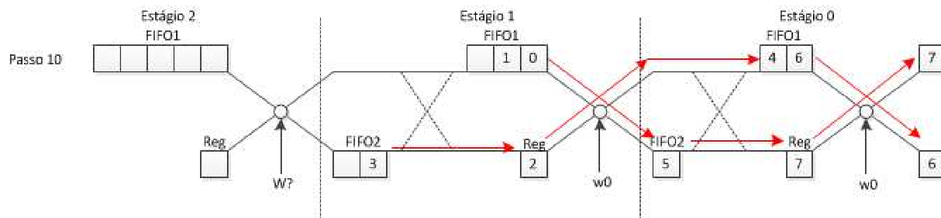


Figura A.17 - Passo 10 da simulação.

No passo 11, temos que o elemento 4 chegou ao final da FIFO1 do estágio 0, e o elemento 5 entrou no registrador deste estágio através da seleção feita pelo multiplexador. Com isto estes dados foram para a saída do *pipeline*. O elemento 3 que está na saída do estágio anterior passa pelo operador borboleta juntamente com o elemento 1, e é direcionado para o registrador do estágio 0. Os demais elementos se movimentam de acordo com o algoritmo desenvolvido. A Figura A.18 apresenta este passo.

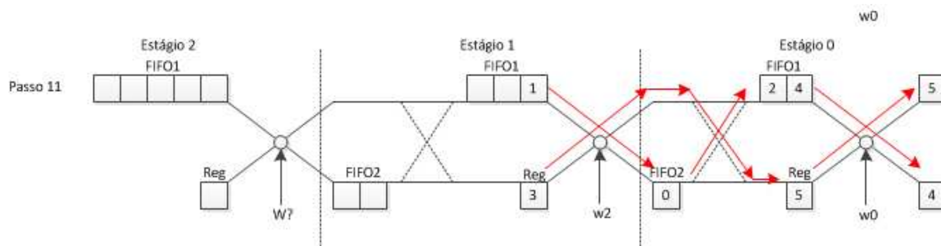


Figura A.18 - Passo 11 da simulação.

No passo 12, temos o elemento 3 realizando a operação borboleta com o elemento 2, e sendo direcionado para saída. As filas se movimentam normalmente ficando prontos para o último passo.

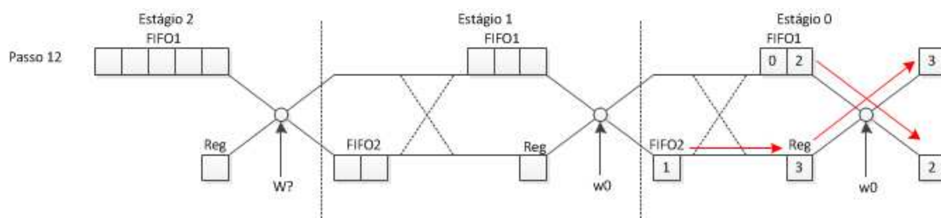


Figura A.19 - Passo 12 da simulação.

O passo 13 é o último necessário para que todos os dados saiam do *pipeline*. O elemento 0 realiza a operação borboleta com o elemento 1, e o resultado é colocado para fora do *pipeline*.

Figura A.20 - Passo 13 da simulação.

A saída de sincronização, conforme já comentado, permanecerá ativa até que o dobro da quantidade de pontos em número de *clocks* seja entrado no sistema. Isto ocorre porque no primeiro estágio o sinal de sincronização permanece ativo até que todos os dados tenham entrado no sistema, e fazendo a analise da propagação da fila existente em cada estágio chega-se a esta conclusão. A Figura A.21 a seguir apresenta uma comparação entre cada passo executado em que houve uma operação borboleta e o algoritmo de dizimação em frequência original. No final, temos as posições corretas referentes a cada elemento da saída do *pipeline*. Observa-se que em alguns passos mais de uma operação borboleta foi realizada, como nos passos 8, 10, 11, 12 e 13.

Figura A.21 - Relação dos passos com a execução do algoritmo DIF.

Apesar de estar enumerado conforme a entrada, a ordenação da saída não corresponde ao número apresentado dentro dos quadrados da última coluna da Figura A.21. Este número é apenas uma referencia para entender o funcionamento do algoritmo, visto que no decorrer do *pipeline*, muitas operações são realizadas entre eles, e com isto deixam de corresponder a exatamente ao ele-

mento. Segundo os algoritmos de dizimação na frequência, a saída encontra-se ordenada de acordo o algoritmo *Bit-Reverse*. Portanto para obter os dados na ordem correta, é necessário executar este algoritmo sobre a sequência de pontos da saída do *pipeline*.

Apêndice B - Códigos VHDL

B.1 Tipos

```
1  ----------------Package--------------------
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use IEEE.STD_LOGIC_ARITH.ALL;
5  use IEEE.STD_LOGIC_SIGNED.ALL;
6  -------------------------------------------
7  PACKAGE tipos IS
8    type complex is
9      record
10       r : std_logic_vector(17 downto 0);
11       i : std_logic_vector(17 downto 0);
12     end record;
13 end tipos;
```

B.2 Comutador

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  USE ieee.std_logic_arith.all;
4  use ieee.std_logic_signed.all;
5  use work.tipos.ALL;
6  entity staged_mux is
7  generic (stage: integer:= 2);
8  port(    clock, en, clr: in std_logic;
9         a, b: in complex;
10        data_out: out complex :=(others=>(others=>'0'))
```

```
11        );
12 end staged_mux;
13 architecture rtl of staged_mux is
14 signal seletor: std_logic:='0';
15 constant estagio: integer range 0 to 8192:= (2**stage);
16 signal i: integer range 0 to 8192:=0;
17 begin
18 data_out <= a when seletor='0' else b;
19 process (clock)
20 begin
21     if(clr = '1') then
22       seletor <= '0';
23       i<= 0;
24      elsif(rising_edge(clock) and en='1')then
25         if(i = estagio-1 )then
26           seletor <= not seletor;
27           i<= 0;
28         else
29           i<= i+1;
30         end if;
31   end if;
32 end process;
33 end rtl;
```

B.3 Registrador

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 USE ieee.std_logic_arith.all;
4 use work.tipos.ALL;
5 entity regcomplex is
6 port(    clock, clr: in std_logic;
7        data_in: in complex;
8        data_out: out complex :=(others=>(others=>'0'))
```

```
9          );
10 end regcomplex;
11 architecture rtl of regcomplex is
12 begin
13 process (clock)
14 begin
15   if (clr='1') then
16     data_out <= (others=>(others=>'0'));
17   elsif(rising_edge(clock))then
18     data_out <= data_in;
19   end if;
20 end process;
21 end rtl;
```

B.4 FIFO

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 USE ieee.std_logic_arith.all;
4 use work.tipos.ALL;
5 entity fifo_N is
6 generic (size: integer:= 4);
7 port(    clock, en, clr: in std_logic;
8        data_in: in complex;
9        data_out: out complex :=(others=>(others=>'0'))
10          );
11 end fifo_N;
12 architecture rtl of fifo_N is
13 type vetor_complex is array (size-1 downto 0) of complex;
14 signal vetor: vetor_complex:= ( (others=>(others=>(others=>'0'))));
15 begin
16 process (clock, en, clr)
17 variable i: integer range 0 to size -1:=0;
18 begin
```

```
19    if(clr = '1') then
20      vetor <=(others=>(others=>(others=>'0')));
21    elsif(rising_edge(clock) and (en='1'))then
22      vetor(size-1 downto 1) <= vetor(size-2 downto 0);
23      vetor(0) <= data_in;
24    end if;
25    data_out <= vetor(size-1);
26  end process;
27  end rtl;
```

B.5 *Butterfly* DIF

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use work.tipos.ALL;
4  USE work.componentes.all;
5  entity butterflyDIF is
6     port(
7        s1,s2 : in complex;
8        w :in complex;
9        g1,g2 :out complex
10    );
11 end butterflyDIF;
12 architecture Behavioral of butterflyDIF is
13 signal t1: COMPLEX;
14 begin
15 --butterfly equations with components
16 u1: somaCPX port map(s1, s2, g1);
17 u2: subCPX port map(s1, s2, t1);
18 u3: multCPX port map(t1, w, g2);
19 end Behavioral;
```

B.6 Estágio

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE ieee.std_logic_arith.all;
use work.tipos.ALL;
use work.componentes.all;
entity fft_pipeline_stage_w is
generic (stage: integer:= 4 );
port(   clock, en, clr: in std_logic;
        din0, din1 : in complex;
        dout0, dout1: out complex :=(others=>(others=>'0'));
          en_out: out std_logic:='0' --saida do enable
        );
end fft_pipeline_stage_w;
architecture rtl of fft_pipeline_stage_w is
signal fifo1_complex_out, fifo2_complex_in, fifo2_complex_out, reg_complex_in,
      reg_complex_out, w: complex;
signal clk, w_en, sEn_out: std_logic;
begin
clk <= en and clock;
u0_fifo1: fifo_N generic map(size => 2**stage)port map(clock, '1', clr, din1,
     fifo1_complex_out);
u1_mux1 : staged_mux generic map(stage => stage) port map(clock, en, clr,
     din0, fifo1_complex_out, fifo2_complex_in);
u2_mux2 : staged_mux generic map(stage => stage) port map(clock, en, clr,
     fifo1_complex_out, din0, reg_complex_in);
u3_fifo2: fifo_N generic map(size => (2**stage)+1 ) port map(clock, '1', clr,
     fifo2_complex_in, fifo2_complex_out);
u4_reg  : regcomplex port map(clock, clr,  reg_complex_in, reg_complex_out);
u5_bf   : butterflyDIF port map( fifo2_complex_out, reg_complex_out, w, dout1,
      dout0);
u6_wgen : w_generator generic map(stage => stage) port map(clock, en, clr, w);
u7_en   : bit_fifo_N generic map(size=>(2**stage)+1) port map(clock, '1', clr,
```

```
        en, sEn_out);
27          en_out <= sEn_out;
28 end rtl;
```

B.7 *Pipeline*

```
1 Library ieee;
2 USE ieee.std_logic_1164.all;
3 USE ieee.std_logic_arith.all;
4 USE ieee.std_logic_signed.all;
5 USE work.tipos.all;
6 USE work.componentes.all;
7 ENTITY fft_pipeline_8s IS PORT (
8      clk, en, clr: IN std_logic;
9      cpxIN0, cpxIN1: IN complex;
10     cpxOUT0, cpxOUT1: out complex;
11     enOUT: out std_logic;
12     counter: out std_logic_vector( 9 downto 0)
13     );
14 END fft_pipeline_8s;
15 ARCHITECTURE bhv OF fft_pipeline_8s IS
16 signal en0, en1, en2, en3, en4, en5, en6, en7, en8, en9, en10: std_logic;
17 signal cpx01, cpx02,cpx11, cpx12,cpx21, cpx22,cpx31, cpx32,cpx41, cpx42:
     complex;
18 signal cpx51, cpx52,cpx61, cpx62,cpx71, cpx72,cpx81, cpx82,cpx91, cpx92:
     complex;
19 signal cpx101, cpx102,cpx121, cpx122,cpx131, cpx132,cpx141, cpx142,cpx111,
     cpx112: complex;
20 BEGIN
21 u7: fft_pipeline_stage_w generic map (stage => 7)  port map (clk, en, clr,
     cpxIN0, cpxIN1, cpx71, cpx72, en7 );
22 u6: fft_pipeline_stage_w generic map (stage => 6)  port map (clk, en7, clr,
     cpx71, cpx72, cpx61, cpx62, en6 );
23 u5: fft_pipeline_stage_w generic map (stage => 5)  port map (clk, en6, clr,
```

```
    cpx61, cpx62, cpx51, cpx52, en5 );
24 u4: fft_pipeline_stage_w generic map (stage => 4)   port map (clk, en5, clr,
    cpx51, cpx52, cpx41, cpx42, en4 );
25 u3: fft_pipeline_stage_w generic map (stage => 3)   port map (clk, en4, clr,
    cpx41, cpx42, cpx31, cpx32, en3 );
26 u2: fft_pipeline_stage_w generic map (stage => 2)   port map (clk, en3, clr,
    cpx31, cpx32, cpx21, cpx22, en2 );
27 u1: fft_pipeline_stage_w generic map (stage => 1)   port map (clk, en2, clr,
    cpx21, cpx22, cpx11, cpx12, en1 );
28 u0: fft_pipeline_stage_w generic map (stage => 0)   port map (clk, en1, clr,
    cpx11, cpx12, cpxOUT0, cpxOUT1, enOut );
29 END bhv;
```

APÊNDICE C - ESTRUTURA DE DADOS DO MÓDULO DE GERENCIAMENTO DE TELAS

```
1  typedef struct  Button {
2    TScreen          *OwnerScreen;
3    char             Order;
4    unsigned int     Left;
5    unsigned int     Top;
6    unsigned int     Width;
7    unsigned int     Height;
8    char             Pen_Width;
9    unsigned int     Pen_Color;
10   char             Visible;
11   char             Active;
12   char         Pressed;
13   char             Transparent;
14   char             *Caption;
15   const char       *FontName;
16   unsigned int     Font_Color;
17   char             Gradient;
18   char             Gradient_Orientation;
19   unsigned int     Gradient_Start_Color;
20   unsigned int     Gradient_End_Color;
21   unsigned int     Color;
22   char             PressColEnabled;
23   unsigned int     Press_Color;
24   void             (*OnUpPtr)();
25   void             (*OnDownPtr)();
26   void             (*OnClickPtr)();
27   void             (*OnPressPtr)();
28 } TButton;
29
30 typedef struct  Button_Round {
31   TScreen          *OwnerScreen;
```

```
32   char            Order;
33   unsigned int    Left;
34   unsigned int    Top;
35   unsigned int    Width;
36   unsigned int    Height;
37   char            Pen_Width;
38   unsigned int    Pen_Color;
39   char            Visible;
40   char            Active;
41   char            Transparent;
42   char            *Caption;
43   const char      *FontName;
44   unsigned int    Font_Color;
45   char            Gradient;
46   char            Gradient_Orientation;
47   unsigned int    Gradient_Start_Color;
48   unsigned int    Gradient_End_Color;
49   unsigned int    Color;
50   char            PressColEnabled;
51   unsigned int    Press_Color;
52   void            (*OnUpPtr)();
53   void            (*OnDownPtr)();
54   void            (*OnClickPtr)();
55   void            (*OnPressPtr)();
56 } TButton_Round;
57
58 typedef struct  Label {
59   char            Order;
60   TScreen         *OwnerScreen;
61   unsigned int    Left;
62   unsigned int    Top;
63   unsigned int    Width;
64   unsigned int    Height;
65   char            *Caption;
66   const char      *FontName;
67   unsigned int    Font_Pos_Ver;
```

```
68    unsigned int   Font_Color;
69    char           Visible;
70    char           Active;
71    void           (*OnUpPtr)();
72    void           (*OnDownPtr)();
73    void           (*OnClickPtr)();
74    void           (*OnPressPtr)();
75  } TLabel;
76
77  typedef struct  Image {
78    TScreen        *OwnerScreen;
79    char           Order;
80    unsigned int   Left;
81    unsigned int   Top;
82    unsigned int   Width;
83    unsigned int   Height;
84    unsigned int   PictureWidth;
85    unsigned int   PictureHeight;
86    const char     *Picture_Name;
87    int       *Picture;
88    char           Visible;
89    char           Active;
90    char           AutoSize;
91    char           Picture_Type;
92    char           Picture_Ratio;
93    void           (*OnUpPtr)();
94    void           (*OnDownPtr)();
95    void           (*OnClickPtr)();
96    void           (*OnPressPtr)();
97  } TImage;
98
99  typedef struct  Graph {
100   TScreen        *OwnerScreen;
101   char           Order;
102   unsigned int   Left;
103   unsigned int   Top;
```

```
104   unsigned int   Width;
105   unsigned int   Height;
106   char           *Graph_Name;
107   short        *Data;
108   const char     *FontName;
109   int        DataSize;
110   int          DataColor;
111   int          BackColor;
112   char         ShowGrid;
113   int        GridColor;
114   char         ShowBorder;
115   int        BorderColor;
116   char         GraphType;
117   char           Visible;
118   char           Active;
119   void           (*OnUpPtr)();
120   void           (*OnDownPtr)();
121   void           (*OnClickPtr)();
122   void           (*OnPressPtr)();
123 } TGraph;
124
125 typedef struct   Box {
126   TScreen        *OwnerScreen;
127   char           Order;
128   unsigned int   Left;
129   unsigned int   Top;
130   unsigned int   Width;
131   unsigned int   Height;
132   char           Pen_Width;
133   unsigned int   Pen_Color;
134   char           Visible;
135   char           Active;
136   char           Transparent;
137   char           Gradient;
138   char           Gradient_Orientation;
139   unsigned int   Gradient_Start_Color;
```

```
140   unsigned int   Gradient_End_Color;
141   unsigned int   Color;
142   char           PressColEnabled;
143   unsigned int   Press_Color;
144   void           (*OnUpPtr)();
145   void           (*OnDownPtr)();
146   void           (*OnClickPtr)();
147   void           (*OnPressPtr)();
148 } TBox;
149
150 typedef struct  Box_Round {
151   TScreen        *OwnerScreen;
152   char           Order;
153   unsigned int   Left;
154   unsigned int   Top;
155   unsigned int   Width;
156   unsigned int   Height;
157   char           Pen_Width;
158   unsigned int   Pen_Color;
159   char           Visible;
160   char           Active;
161   char           Transparent;
162   char           Gradient;
163   char           Gradient_Orientation;
164   unsigned int   Gradient_Start_Color;
165   unsigned int   Gradient_End_Color;
166   unsigned int   Color;
167   char           PressColEnabled;
168   unsigned int   Press_Color;
169   void           (*OnUpPtr)();
170   void           (*OnDownPtr)();
171   void           (*OnClickPtr)();
172   void           (*OnPressPtr)();
173 } TBox_Round;
174
175 typedef struct  Line {
```

```
176   TScreen         *OwnerScreen;
177   char            Order;
178   unsigned int    First_Point_X;
179   unsigned int    First_Point_Y;
180   unsigned int    Second_Point_X;
181   unsigned int    Second_Point_Y;
182   char            Pen_Width;
183   char            Visible;
184   char            Active;
185   unsigned int    Color;
186 } TLine;
187 
188 struct Screen {
189     unsigned int           Color;
190     unsigned int           Width;
191     unsigned int           Height;
192     unsigned short         ObjectsCount;
193     unsigned int           ButtonsCount;
194     TButton                **Buttons;
195     unsigned int           Buttons_RoundCount;
196     TButton_Round          **Buttons_Round;
197     unsigned int           LabelsCount;
198     TLabel                 **Labels;
199     unsigned int           ImagesCount;
200     TImage                 **Images;
201     unsigned int           BoxesCount;
202     TBox                   **Boxes;
203     unsigned int           Boxes_RoundCount;
204     TBox_Round             **Boxes_Round;
205     unsigned int           LinesCount;
206     TLine                 **Lines;
207     unsigned int      GraphsCount;
208     TGraph        **Graphs;
209 };
```

APÊNDICE D - DESEMPENHO DE UMA MULTIPLICAÇÃO NO PROCESSADOR NIOS 2 E SEU EFEITO NA FFT EM SOFTWARE

O processador NIOS 2 tem a capacidade de utilizar instruções personalizadas. Dentre os grupos pré-existentes no SoPC *Builder*, destaca-se a biblioteca de operações de ponto flutuante em *hardware*, que acelera muito a execução dos cálculos.

Através do módulo medidor de desempenho (*performance counter*) é possível mensurar a quantidade de *clocks* entre dois instantes de tempo. Foi mensurado a quantidade de *clocks* para as operações de multiplicação. Nota-se pela tabela D.1, com multiplicação de ponto flutuante em *hardware*, a quantidade de *clocks* necessária para uma multiplicação diminui consideravelmente.

Tabela D.1 - Comparação dos resultados para executar 1 multiplicação, em milissegundos e por pulsos de *clock*.

Operação	Tempo (ms)	Clocks
C = A*B (SW)	0.0110	1102
C = A*B (HW)	0.0007	72

Da mesma maneira utilizou-se do medidor de desempenho para comparar a velocidade, em pulsos de *clock*, da execução de cada passo do algoritmo. Nota-se um ganho notável na utilização de instruções de ponto flutuante com relação a versão somente em *software*. A tabela D.2 apresenta os tempos em cada etapa do calculo da FFT, considerando um vetor de entrada de 256 pontos.

Tabela D.2 - Tempos de execução do algoritmo FFT para 256 pontos em ponto flutuante

Operação	Tempo (ms)	Clocks
Carrega tabela W	106.44	10645381
Reordenar vetor	0.694	69466
Aplicar Cooley Tukey (SW FP)	51.07	5106824
Aplicar Cooley Tukey (HW FP)	7.92	792680