

Dissertação de Mestrado

CONTRIBUIÇÃO PARA O ESTUDO DO EMBARQUE DE UMA REDE NEURAL ARTIFICIAL EM FIELD PROGRAMMABLE GATE ARRAY (FPGA)

Carlos Alberto de Albuquerque Silva

Natal, junho de 2010

Seção de Informação e Referência

Catálogo da Publicação na Fonte. UFRN / Biblioteca Central Zila Mamede

Silva, Carlos Alberto de Albuquerque.

Contribuição para o estudo do embarque de uma rede neural artificial em field programmable gate array (FPGA) / Carlos Alberto de Albuquerque Silva. – Natal, RN, 2010.

138 f. : il.

Orientador: Adrião Duarte Doria Neto.

Co-orientador: José Alberto Nicolau de Oliveira

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Tecnologia. Programa de Pós-Graduação em Engenharia Elétrica.

1. Computação Reconfigurável – Dissertação. 2. Redes Neurais Artificiais – Dissertação. 3. FPGA – Dissertação. 4. VHDL – Dissertação 5. *Hardware* – Dissertação. 6. Aritmética Ponto Fixo – Dissertação I. Doria Neto Adrião Duarte. II. Oliveira, José Alberto Nicolau de. III Universidade Federal do Rio Grande do Norte. IV. Título.

RN/UF/BCZM

CDU 004.032.26

CARLOS ALBERTO DE ALBUQUERQUE SILVA

**CONTRIBUIÇÃO PARA O ESTUDO DO EMBARQUE DE UMA REDE NEURAL
ARTIFICIAL EM FIELD PROGRAMMABLE GATE ARRAY (FPGA)**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da UFRN (Área de Concentração: Engenharia de Computação), como parte dos requisitos para a obtenção do título de Mestre em Ciências.

Orientador: Prof. Dr. Adrião Duarte Doria Neto

Co-orientador: Prof. Dr. José Alberto Nicolau de Oliveira

NATAL
2010

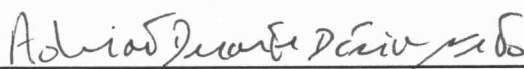
CARLOS ALBERTO DE ALBUQUERQUE SILVA

**CONTRIBUIÇÃO PARA O ESTUDO DO EMBARQUE DE UMA REDE
NEURAL ARTIFICIAL EM FIELD PROGRAMMABLE GATE ARRAY (FPGA)**

Dissertação de mestrado em Engenharia Elétrica e de Computação da UFRN, em cumprimento às exigências para obtenção do grau de Mestre em Ciências, na área de Engenharia de Computação.

Aprovado em: 30 JUNHO 2010

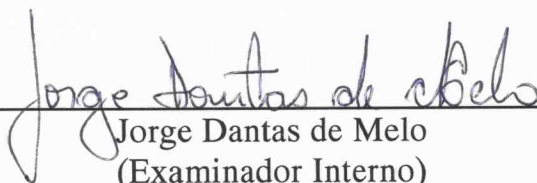
BANCA EXAMINADORA



Adrião Duarte Dória Neto
(Orientador)



José Alberto Nicolau de Oliveira
(Co-orientador)



Jorge Dantas de Melo
(Examinador Interno)



Danniell Cavalcante Lopes
(Examinador Externo)

Dedico a Deus, por Sua presença em todos os momentos de minha vida, deixando-me mais forte para superar os desafios, e aos meus pais, que me deram não somente a vida, mas, principalmente, a minha educação e condições de estudo.

AGRADECIMENTOS

Agradeço a Deus, por iluminar e guiar meu caminho durante este percurso.

Aos meus pais, Francisco e Ivanilda, por serem os principais responsáveis por esta conquista e pelo contínuo apoio em todos os momentos, ensinando-me, sobretudo, a importância da construção e coerência de meus próprios valores.

Aos meus irmãos, Alex e Fábio e a meus familiares, pelo amor, carinho, força e por acreditarem em mim.

Ao meu sobrinho Felipe, pela “quase” compreensão da ausência de “ti Carlim” nas brincadeiras.

Ao meu tio Martinho, pelo apoio, pela ajuda, paciência e por contribuir para este trabalho, ensinando e esclarecendo assuntos matemáticos em vários momentos de meus estudos.

A minha namorada, amiga e companheira, Simone Karine, por todo amor, pela ajuda, pelas sábias palavras de conselhos e pelos incentivos prestados.

Ao meu orientador, Dr. Adrião Duarte Doria Neto, e ao meu co-orientador, Dr. José Alberto Nicolau de Oliveira, pela orientação com suas idéias, implicações e críticas construtivas, e pela oportunidade de, com suas argumentações científicas e sugestões essenciais para o desenvolvimento deste trabalho, enriquecer meus conhecimentos.

Aos professores Dr. David Simonetti Barbalho e Dr. Jorge Dantas de Melo, pelos comentários e significantes sugestões que favoreceram a construção deste trabalho.

Aos amigos e colegas do Programa de Pós-Graduação em Engenharia Elétrica e de Computação - PPGEEC da UFRN, que contribuíram com sua amizade, compartilhando comigo idéias, fomentando discussões que favoreceram meu aprendizado.

Agradeço e ao Programa de Recursos Humanos - PRH14 da Agência Nacional de Petróleo - ANP, pelo auxílio financeiro.

E a todos que me ajudaram, diretamente ou indiretamente, neste percurso.

MUITO OBRIGADO!

*O essencial é ter na vida um ideal definido com
aptidão e perseverança suficiente para atingi-lo.
(autor desconhecido)*

RESUMO

Este estudo consiste na implementação e no embarque de uma Rede Neural Artificial (RNA) em *hardware*, ou seja, em um dispositivo programável do tipo *field programmable gate array* (FPGA). O presente trabalho permitiu a exploração de diferentes implementações, descritas em VHDL, de RNA do tipo *perceptrons* de múltiplas camadas. Por causa do paralelismo inerente às RNAs, ocorrem desvantagens nas implementações em *software*, devido à natureza sequencial das arquiteturas de Von Neumann. Como alternativa a este problema, surge uma implementação em *hardware* que permite explorar todo o paralelismo implícito neste modelo. Atualmente, verifica-se um aumento no uso do FPGA como plataforma para implementar as Redes Neurais Artificiais em *hardware*, explorando o alto poder de processamento, o baixo custo, a facilidade de programação e capacidade de reconfiguração do circuito, permitindo que a rede se adapte a diferentes aplicações. Diante desse contexto, objetivou-se desenvolver arranjos de redes neurais em *hardware*, em uma arquitetura flexível, nas quais fosse possível acrescentar ou retirar neurônios e, principalmente, modificar a topologia da rede, de forma a viabilizar uma rede modular em aritmética de ponto fixo, em um FPGA. Produziram-se cinco sínteses de descrições em VHDL: duas para o neurônio com uma e duas entradas, e três para diferentes arquiteturas de RNA. As descrições das arquiteturas utilizadas tornaram-se bastante modulares, possibilitando facilmente aumentar ou diminuir o número de neurônios. Em decorrência disso, algumas redes neurais completas foram implementadas em FPGA, em aritmética de ponto fixo e com alta capacidade de processamento paralelo.

Palavras-Chave: Computação Reconfigurável, Redes Neurais Artificiais, FPGA, VHDL, *Hardware*, Aritmética Ponto Fixo.

ABSTRACT

This study shows the implementation and the embedding of an Artificial Neural Network (ANN) in hardware, or in a programmable device, as a field programmable gate array (FPGA). This work allowed the exploration of different implementations, described in VHDL, of multilayer perceptrons ANN. Due to the parallelism inherent to ANNs, there are disadvantages in software implementations due to the sequential nature of the Von Neumann architectures. As an alternative to this problem, there is a hardware implementation that allows to exploit all the parallelism implicit in this model. Currently, there is an increase in use of FPGAs as a platform to implement neural networks in hardware, exploiting the high processing power, low cost, ease of programming and ability to reconfigure the circuit, allowing the network to adapt to different applications. Given this context, the aim is to develop arrays of neural networks in hardware, a flexible architecture, in which it is possible to add or remove neurons, and mainly, modify the network topology, in order to enable a modular network of fixed-point arithmetic in a FPGA. Five synthesis of VHDL descriptions were produced: two for the neuron with one or two entrances, and three different architectures of ANN. The descriptions of the used architectures became very modular, easily allowing the increase or decrease of the number of neurons. As a result, some complete neural networks were implemented in FPGA, in fixed-point arithmetic, with a high-capacity parallel processing.

Keywords: Reconfigurable Computing, Artificial Neural Network, FPGA, VHDL, Hardware, Arithmetic Fixed Point.

LISTA DE ILUSTRAÇÕES

Figura 1	- Fases envolvidas no desenvolvimento de um projeto em FPLD	18
Figura 2	- Geração das bases de dados para uma simulação em nível de pré-síntese.	20
Figura 3	- Arquitetura de um FPGA	23
Quadro 1	- Resumo das famílias e características dos FPGA da Altera®	25
Quadro 2	- Resumo das famílias e características dos FPGA da Xilinx Inc.	26
Figura 4	- Neurônio Artificial	32
Gráfico 1	- Transformação afim produzida pela presença de um <i>bias</i>	34
Figura 5	- Neurônio artificial e o <i>bias</i> como sendo peso da sinapse (w_0)	34
Gráfico 2	- Função de limiar	35
Gráfico 3	- Função de linear por partes	35
Gráfico 4	- Função sigmóide	36
Figura 6	- Rede alimentada adiante com uma única camada	38
Figura 7	- Rede alimentada adiante com uma camada oculta e uma camada de saída	39
Figura 8	- Rede recorrente	39
Figura 9	- Estrutura proposta para o neurônio em <i>hardware</i>	45
Figura 10	- Arquitetura do bloco NEURONIO	46
Figura 11	- Arquitetura do bloco NET	46
Figura 12	- Arquitetura do bloco NET com mais de uma entrada de dados	47
Figura 13	- Arquitetura do bloco FNET	48
Figura 14	- Arquitetura de um <i>perceptron</i> de múltiplas camadas com dois neurônios na camada oculta e um neurônio na camada de saída	49
Quadro 3	- Resultados das simulações com o neurônio usando uma única entrada	53
Quadro 4	- Resultados das simulações com o neurônio usando duas entradas	53
Quadro 5	- Número de elementos lógicos utilizados e área de silício ocupada no FPGA nas implementações do bloco NEURONIO com uma e duas entradas	53
Figura 15	- Arquitetura da rede neural usada para executar a função exponencial	54
Quadro 6	- Resultados obtidos para a função exponencial	54

Figura 16	- Quantidade de elementos lógicos e área de silício utilizada do FPGA	55
Gráfico 5	- Resultados da função Sinc executada no Matlab	56
Quadro 7	- Resultados comparativos da rede neural utilizada para aproximar a função Sinc	56
Figura 17	- Dados de síntese obtidos no software Quartus II da Altera	56
Figura 18	- Arquitetura da rede neural usada para executar a função XOR	57
Quadro 8	- Resultados comparativos da rede neural utilizada para a execução da função XOR	57
Figura 19	- Resultados de simulação da função XOR	58
Quadro 9	- Taxas de ocupação do FPGA na execução da função XOR	58

LISTA DE SIGLAS E ABREVIATURAS

AHDL	<i>Altera Hardware Description Language</i>
AND	<i>E</i>
ASIC	<i>Application Specific Integrated Circuit</i>
ASIP	<i>Application Specific Instruction-set Processor</i>
CAD	<i>Computer Aided Design</i>
CLB	<i>Configurable Logic Block</i>
CPLD	<i>Complex Programmable Logic Devices</i>
CR	<i>Computação Reconfigurável</i>
DARPA	<i>Departamento de Defesa dos Estados Unidos da América</i>
ECG	<i>Eletrocardiograma</i>
EEPROM	<i>Electrically-erasable Programmable Read Only Memory</i>
EPROM	<i>Erasable Programmable Read Only Memory</i>
FFT	<i>Fast Fourier transform</i>
FPGA	<i>Field Programmable Gate Array</i>
FPLD	<i>Field Programmable Logic Devices</i>
HCPLD	<i>High Capacity Programmable Logic Devices</i>
HDL	<i>Hardware Description Language</i>
HF	<i>Hardware Fixo</i>
HP	<i>Hardware Programável</i>
HP+SW	<i>Hardware software codesign</i>
IA	<i>Inteligência Artificial</i>
IEEE	<i>Institute of Electrical and Electronic Engineer</i>
IOB	<i>Input/Output Block</i>
IP	<i>Intelectual Property</i>
JTAG/BST	<i>Joint Test Action Group/boundary scan test</i>
MOS	<i>Metal Oxide Semiconductor</i>
NOT	<i>NÃO</i>
OR	<i>OU</i>
PC	<i>Personal Computer</i>
PLA	<i>Programmable Logic Arrays</i>
PLD	<i>Programmable Logic Devices</i>
PPM	<i>Prediction by Partial Matching</i>
RAM	<i>Random Access Memory</i>
REVCOM	<i>Review Committee</i>
RISP	<i>Reconfigurable Instruction Set Processors</i>
RNA	<i>Redes Neurais Artificiais</i>
RTL	<i>Nível de Transferência de Registro</i>
SPLD	<i>Simple Programmable Logic Devices</i>
SRAM	<i>Static Random Access Memory</i>
SW	<i>Software</i>
VHDL	<i>VHSIC Hardware Description Language</i>
VHSIC	<i>Very High Speed Integrated Circuits</i>
XOR	<i>OU Exclusivo</i>

SUMÁRIO

1	INTRODUÇÃO	13
2	COMPUTAÇÃO RECONFIGURÁVEL	16
2.1	DISPOSITIVOS RECONFIGURÁVEIS	17
2.1.1	Projeto baseado em FPLD	18
2.1.1.1	Entrada do projeto	19
2.1.1.2	Simulação em nível de pré-síntese	19
2.1.1.3	Compilação e síntese	20
2.1.1.4	Análise temporal	20
2.1.1.5	Simulação em nível de pós-síntese	21
2.1.1.6	Programação do FPLD	21
2.1.2	Field Programmable Gate Array (FPGA)	22
2.1.2.1	Classificação dos FPGA	23
2.1.2.2	Tecnologias de programação	24
2.1.2.3	Fabricantes	25
2.2	LINGUAGEM DE DESCRIÇÃO DE <i>HARDWARE</i>	26
2.2.1	A linguagem de descrição VHDL	27
2.2.1.1	História	27
2.2.1.2	Características da linguagem	28
2.2.1.3	Vantagens e Desvantagens	28
2.3	ALGUMAS APLICAÇÕES RECENTES COM FPGA	29
3	REDES NEURAIS ARTIFICIAIS	31
3.1	MODELOS DE UM NEURÔNIO ARTIFICIAL	32
3.1.1	Tipos de função de ativação	34
3.2	ARQUITETURAS DE REDES NEURAIS	37
3.2.1	Rede neural alimentada adiante com camada única	37
3.2.2	Rede neural com múltiplas camadas	38
3.2.3	Rede recorrente	39
3.3	PROCESSOS DE APRENDIZAGEM	40
3.3.1	Aprendizado supervisionado por retropropagação do erro	40
3.4	ALGORITMOS INTELIGENTES E REDES NEURAIS	41
3.5	REDES NEURAIS ARTIFICIAIS EM FPGA	42
4	IMPLEMENTAÇÃO DO NEURÔNIO E DE REDES NEURAIS EM FPGA	44
4.1	DESCRIÇÃO ESTRUTURAL DO NEURÔNIO EM FPGA	44
4.2	DESCRIÇÃO ESTRUTURAL DAS REDES NEURAIS IMPLEMENTADAS EM FPGA	48
4.3	DEFINIÇÃO DOS PARÂMETROS DA REDE: PESOS SINÁPTICOS E <i>BIAS</i>	50
5	TESTES, SIMULAÇÕES E RESULTADOS OBTIDOS	52
5.1	TESTES E SIMULAÇÕES FEITAS COM O BLOCO NEURÔNIO	52
5.2	TESTES E SIMULAÇÕES FEITAS COM AS REDES NEURAIS EM FPGA	54
5.2.1	Função exponencial	54
5.2.2	Função <i>Sinc</i>	55

5.2.3	XOR	57
6	CONCLUSÃO	59
	REFERÊNCIAS	62
	APÊNDICES	65

1 INTRODUÇÃO

No mundo moderno, cada vez mais as Redes Neurais Artificiais (RNAs), que constituem uma das ramificações da Inteligência Artificial (IA), estão sendo utilizadas nos mais variados campos de pesquisa, tanto para o desenvolvimento como para aplicação dessas redes.

Segundo Braga, Carvalho e Ludemir (2007), as RNAs são conhecidas como conexismo ou sistemas de processamento paralelo e distribuído que consistem em um modo de abordar soluções de problemas complexos, a exemplo de problemas de reconhecimento de padrões, classificação e aproximação de funções, sendo essencialmente proveitosas no emprego da característica de generalização.

O desenvolvimento de uma RNA pode ser realizado tanto em *software* quanto em *hardware*, havendo vantagens e desvantagens em ambos. Para o desenvolvimento em *software*, as vantagens recaem sobre a facilidade e o tempo de execução da rede; já as desvantagens estão relacionadas à lentidão do processamento dos dados, por serem eles processados sequencialmente, etc. Para a implementação em *hardware*, as vantagens estão relacionadas ao paralelismo intrínseco das RNAs, enquanto que as desvantagens ficam a cargo de se alcançar um equilíbrio razoável na precisão de *bits* e na implementação da função de ativação. Dentre as características inerentes às redes neurais artificiais, duas chamam a atenção para o desenvolvimento em *hardware*. São elas: não-linearidade, por permitir a resolução de problemas que não sejam linearmente separáveis; e processamento paralelo, que é a capacidade de receber múltiplas informações e testá-las simultaneamente (HASSAN; ELNAKIB; ABO-ELSOUD, 2008; LUDWIG; COSTA, 2007; VALENÇA, 2005).

Skliarova e Ferrari (2003) aludem à utilização *codesign hardware/software* como uma abordagem da computação reconfigurável que permite a eliminação das desvantagens utilizadas no *software* puro e no *hardware* puro. Esses autores afirmam que a computação reconfigurável se caracteriza por atingir um desempenho elevado ao promover flexibilidade para a programação em nível de porta lógica, baseando-se em dispositivos lógicos reprogramáveis. Um exemplo de dispositivo de *hardware* utilizado em computação reconfigurável é o *Field Programmable Gate Array* (FPGA).

Por volta dos anos 2000, verificou-se um aumento no uso do FPGA como plataforma para implementar as Redes Neurais Artificiais em *hardware*, explorando o alto poder de

processamento, o baixo custo, a facilidade de programação e a capacidade de reconfiguração do circuito, permitindo que a rede se adapte a diferentes aplicações. Outra característica apresentada pela maioria dos FPGA é o fato deles possuírem blocos de memória *Random Access Memory* (RAM) internas do tipo *dual port*, com canais de leitura e escrita, nos quais é possível executar simultaneamente essas duas operações com uso de endereços distintos (AMORE, 2005; VAHID, 2008).

A descrição textual de uma RNA em FPGA pode ser realizada através de uma linguagem de descrição de *hardware* como VHDL, AHDL, Verilog, entre outras. Programas descritos nessas linguagens são sintetizáveis por diversas ferramentas de prototipagem, com o Quartus[®] II da Altera[®] e o ISE[®] da Xilinx[®]. Tais ferramentas favorecem, ainda, o desenvolvimento e a validação do projeto, já que permitem a realização de simulações de pré e de pós-síntese, através das quais podem ser testados tanto o funcionamento do circuito quanto o tempo de atraso dos componentes e o seu desempenho, sem a necessidade de sua prototipagem final (MARTINS *et al.*, 2009).

O presente trabalho tem por finalidade principal desenvolver arranjos de redes neurais em *hardware*, em uma arquitetura flexível, e nas quais seja possível acrescentar ou retirar neurônios e, principalmente, modificar a topologia da rede, ou seja, viabilizar uma rede modular em aritmética de ponto fixo em um FPGA.

Em seu delineamento, esta dissertação está estruturada em seis seções, que guardam, em si, peculiaridades, e a coerência necessária às exigências da investigação, de modo a preservar sua coesão.

Na primeira seção realiza-se a apresentação formal da proposta de estudo, sua fundamentação e seus direcionamentos.

Na segunda seção são apresentados os conceitos básicos da computação reconfigurável, alguns de seus paradigmas, suas vantagens e desvantagens; são analisados alguns dispositivos reprogramáveis, em especial o FPGA, bem como são apresentadas, sucintamente, algumas características da linguagem de descrição de *hardware* que será usada no projeto dos dispositivos de *hardware* necessários ao projeto.

Na seção três apresenta-se a base teórica para a compreensão das Redes Neurais Artificiais, os paradigmas de aprendizagem, suas arquiteturas, trabalhos a elas relacionados e o seu uso em FPGA.

Já na seção quatro são apresentados os métodos usados para a implementação do neurônio e de redes neurais artificiais em FPGA.

Na seção cinco encontram-se os dados dos testes, das simulações e dos resultados das implementações de um neurônio artificial, e de três redes neurais artificiais implementadas em FPGA.

A seção seis encerra a dissertação, com a apresentação das considerações finais, das dificuldades encontradas para a implementação das RNA, das soluções adotadas e sugestões para trabalhos futuros, que poderão dar continuidade a este trabalho, tendo como base o neurônio artificial desenvolvido em *hardware*.

2 COMPUTAÇÃO RECONFIGURÁVEL

Nos últimos anos, vem sendo observado um aumento considerável do interesse pela Computação Reconfigurável (CR) por parte de diversos setores industriais e acadêmicos, tendo em vista que significativos investimentos no desenvolvimento de pesquisas vêm sendo efetuados por grandes representantes da indústria eletrônica, como a Nokia e a Sony, em união com grandes centros de pesquisas (NASCIMENTO *et al.*, 2009).

Nessa perspectiva, a utilização da Computação Reconfigurável representa uma solução intermediária entre as soluções em *hardware* e *software*, para a resolução de problemas complexos, os quais estão relacionados a fatores como desempenho, tempo de resposta, eficiência, disponibilidade, tolerância às falhas, entre outros. Assim sendo, diferentemente das soluções normalmente usadas para resolver esses problemas, a CR visa obter soluções que consumam menos recursos computacionais necessários para armazenamento, recuperação, transmissão e processamento de informações, e que sejam realizadas em baixos períodos de tempo, com possibilidade de operação em tempo real (MARTINS *et al.*, 2009).

Para Martins *et al.* (2009), as soluções para esses problemas podem ser classificadas em dois paradigmas distintos: o primeiro relacionado às soluções implementadas em *Hardware* Fixo (HF); e o segundo, às soluções implementadas em *Hardware* Programável (HP), com uso de técnicas de *hardware software codesign* (HW+SW).

O paradigma HF apresenta algumas desvantagens em sua utilização, sendo relacionado à falta de flexibilidade, já que seus componentes não podem ser alterados após a fabricação, o que torna impossível realizar adaptações futuras.

No paradigma de *hardware* programável através de *software* (HP+SW), a principal desvantagem está relacionada ao desempenho durante a execução de uma aplicação, o qual, muitas vezes, é insatisfatório. Embora permita grande flexibilidade para a execução de qualquer tipo de trabalho que envolva processamento computacional, a característica genérica desses sistemas faz com que as soluções desenvolvidas apresentem um desempenho inferior ao satisfatório.

Considerando as desvantagens apresentadas pelos dois paradigmas, a computação reconfigurável apresenta-se como uma solução intermediária que combina a velocidade do *hardware* com a flexibilidade do *software*. Segundo Skliarova e Ferrari (2003), a Computação Reconfigurável baseia-se em dispositivos lógicos reprogramáveis que podem atingir um

desempenho elevado e, ao mesmo tempo, fornecer a flexibilidade da programação em nível de portas lógicas.

Aragão, Romero e Marques (2009) explanam, em termos básicos, que a CR combina a velocidade e o desempenho do *hardware* com a flexibilidade do *software*. Essa tecnologia consiste na habilidade de se modificar a arquitetura do *hardware* em tempo real, para se adequar à aplicação que será executada.

Contudo, verifica-se que a utilização da CR em inúmeras aplicações de sistemas digitais tem permitido alcançar altas performances com baixo consumo de potência, quando comparada às aplicações praticadas em *software*. No entanto, embora apresentem qualidade inferior àquelas obtidas com implementações em Circuitos Integrados de Aplicação Específica (ASIC – *Application Specific Integrated Circuit*), em termos de área, performance e potência, as implementações em computação reconfigurável têm encontrado aplicação em sistemas que necessitam de flexibilidade semelhante à obtida por *software* e que não pode ser suprida por ASIC (NASCIMENTO *et al.*, 2009).

2.1 DISPOSITIVOS RECONFIGURÁVEIS

A rápida evolução dos recursos de concepção de circuitos integrados, tanto na área de processos quanto na área de *Computer Aided Design* (CAD), tornou possível o surgimento de dispositivos com lógica programável (*Programmable Logic Devices* - PLD) e dos Dispositivos Lógicos Programáveis no Campo (*Field Programmable Logic Devices* - FPLD).

Os dispositivos reconfiguráveis são dispositivos programáveis capazes de serem configurados para reproduzir o comportamento de um circuito lógico em *hardware* (GOMES; CHARÃO; VELHO, 2009a). Esses dispositivos apresentam-se como uma alternativa tecnológica capaz de implementar inúmeras aplicações de propósito geral, enquanto mantêm as vantagens de desempenho de aplicações em dispositivos dedicados.

A primeira utilização destes circuitos se dá, naturalmente, nos projetos de prototipagem, visto que, em sua maioria, esses circuitos podem ser reprogramados e testados nas fases preliminares do projeto, possibilitando, portanto, grande economia de tempo e dinheiro.

Os PLD são classificados em três categorias distintas: Dispositivos Lógicos Programáveis Simples (*Simple Programmable Logic Devices* - SPLD), Dispositivos Lógicos

Programáveis Complexos (*Complex Programmable Logic Devices - CPLD*) e em Arranjos de Portas Programáveis no Campo (*Field Programmable Gate Arrays - FPGA*). Atualmente, dada a tecnologia de alta integração e o considerável aumento no número de elementos lógicos que os constituem, os CPLDs e os FPGAs são referenciados como Dispositivos Lógicos Programáveis de Alta Capacidade (*High Capacity Programmable Logic Devices - HCPLD*) (TOCCI, 2007).

2.1.1 Projeto baseado em FPLD

De acordo com Navabi (2005), um projeto baseado em FPLD se inicia com a especificação e finaliza com a programação do dispositivo-destino, que pode ser um CPLD ou um FPGA. Na Figura 1, são visualizadas as fases envolvidas no desenvolvimento de um projeto em FPLD.

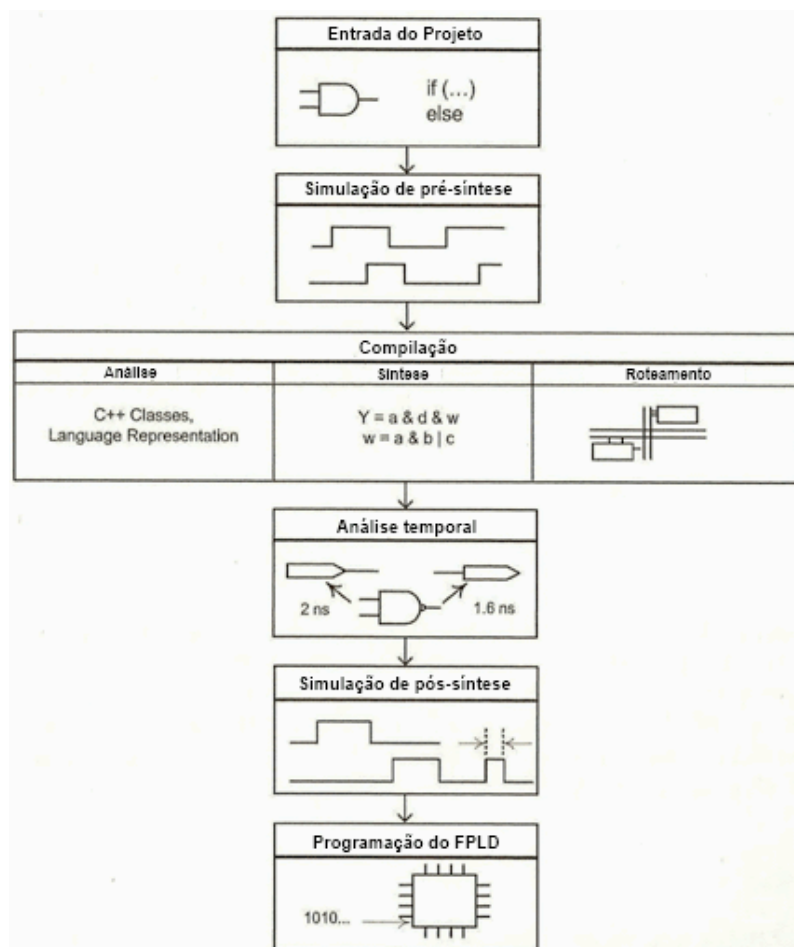


Figura 1 - Fases envolvidas no desenvolvimento de um projeto em FPLD

Fonte: (NAVABI, 2005, pág. 4)

2.1.1.1 Entrada do projeto

Nesta fase, as ferramentas de especificação de um projeto baseado em FPLD normalmente permitem que sua entrada seja realizada de forma textual ou de forma gráfica. Para a entrada textual, faz-se uso de alguma linguagem de descrição de *hardware*, como, por exemplo, a VHDL (*VHSIC Hardware Description Language*; VHSIC: *Very High Speed Integrated Circuits*) ou a Verilog HDL. Para a entrada gráfica, é feito o desenho esquemático do circuito com dispositivos gráficos pré-existentes (primitivas disponibilizadas na biblioteca da ferramenta de projeto utilizada) ou anteriormente projetados ou descritos e incorporados a uma biblioteca. Geralmente, um projeto baseado em FPLD é uma mistura das representações textual e gráfica (NAVABI, 2005).

2.1.1.2 Simulação em nível de pré-síntese

Antes de um projeto ser sintetizado em um FPLD, suas funcionalidades devem ser verificadas. Através dessa verificação, denominada de simulação em nível de pré-síntese, é possível descobrir erros na lógica do projeto, detectar problemas em sua especificação e descobrir incompatibilidade de componentes nele usados. Esse processo é também conhecido por simulação em nível de transferência de registro (*Register Transfer Level - RTL*), que é a técnica usual de modelagem e descrição de sistemas digitais mais complexos.

Todo processo de simulação precisa de uma base de dados para a realização dos testes. Na simulação em nível de pré-síntese, esta base poderá ser gerada graficamente, por meio de um editor de formas de onda, conforme se observa na Figura 2.a, ou por um arquivo *testbench* HDL, observado na Figura 2.b. Já as saídas geradas podem ser vistas nas formas de ondas ou através de relatórios gerenciais, e gravadas em arquivos de diferentes formatos, tais como arquivos representativos de formas de ondas ou em modo texto, respectivamente (NAVABI, 2005).

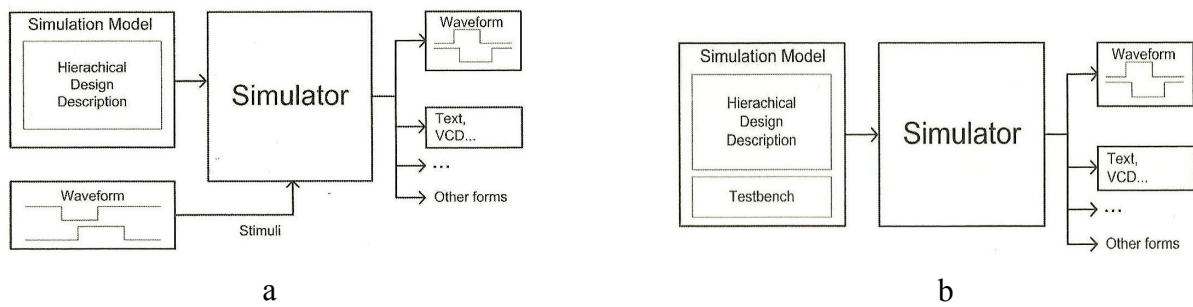


Figura 2 - Geração das bases de dados para uma simulação em nível de pré-síntese

Fonte: (NAVABI, 2005, pág. 8)

2.1.1.3 Compilação e síntese

Somente após uma simulação em nível de pré-síntese, com sucesso, um projeto deve ser compilado. O processo de compilação compreende em algumas fases, tais como:

- Análise, na qual as partes do projeto são checadas e convertidas para um formato intermediário;
- Geração de *hardware* genérico, início do processo de síntese, no qual o projeto é convertido em um conjunto de expressões booleanas e uma *netlist* de portas básicas;
- Otimização, na qual a *netlist* e as expressões booleanas são minimizadas pelo compartilhamento de componentes e pela exclusão de redundâncias;
- *Binding*, roteamento e alocação de componentes, nas quais o processo de síntese busca informações do *hardware*-alvo, decide que elementos lógicos e células do FPLD serão escolhidos e faz as devidas alocações e os respectivos roteamentos construtivos, inerentes ao projeto.

2.1.1.4 Análise temporal

Como parte do processo de compilação ou como ocorre em algumas ferramentas de projeto, após o processo de compilação, há uma fase de análise de sincronismo (*Timing Analysis*). Esta fase gera informações sobre atrasos de pior caso, velocidade de transferência,

atrasos de uma porta para outra, bem como tempos requeridos para gatilho (*setup*) e manutenção dos sinais (*hold*).

Estes resultados são gerados em forma de tabelas e gráficos, tipo formas de ondas, e são usados, pelo projetista, para decidir sobre, por exemplo, que velocidade de *clock* deve ser usada no circuito, bem como servem de entrada para o processo de simulação pós-síntese. Pelas formas de ondas geradas, é possível verificar atrasos nas mudanças de sinais e corridas críticas, as quais aparecem como *glitches* (pulsos aleatórios) (NAVABI, 2005).

2.1.1.5 Simulação em nível de pós-síntese

Durante esta fase, questões de sincronismo, frequências adequadas de *clock* e a ocorrência de corridas críticas podem ser checadas. Isto porque, só após a síntese, os detalhes de que portas são usadas na implementação, que atrasos em fios e em conexões ocorrem e quais os efeitos de alguns carregamentos, que se tornam acessíveis. Todos esses procedimentos de simulação pós-síntese são feitos em nível de *gates*, o que a torna bem mais lenta que a de pré-síntese.

Devido a atrasos, é possível que o comportamento de um projeto não se apresente como desejado e que tenha que ser reavaliado, para que todos os problemas de sincronismo sejam corrigidos (NAVABI, 2005).

2.1.1.6 Programação do FPLD

Após a fase de *Binding*, são gerados dois tipos de arquivos, específicos para os FPLD alvos, um do tipo *.sof* (*SRAM object file*) e outro do tipo *.pof* (*programming object file*), os quais deverão ser utilizados para programar dispositivos do tipo SRAM (*Static Random Access Memory*) e EEPROM (*Electrically-erasable Programmable Read Only Memory*), respectivamente.

Dentre as ferramentas oferecidas pelos fabricantes que fazem a programação dos seus dispositivos *in-circuit*, destacam-se:

- Os *stand alone*, que usam *software* e *hardware* específicos, já adaptados para a programação do dispositivo;
- Os baseados em *personal computer* (PC), que usam adaptadores apropriados para descarregar o projeto via cabo serial, ou paralelo, ou via cabos específicos de *download* associados a portas de padrões especiais do tipo *Joint Test Action Group / boundary scan test* (JTAG / BST), que oferecem eficiente capacidade de teste de componentes (padrão IEEE 1149.1), este último sendo o utilizado pelas ferramentas de *softwares* Quartus II e ISE, desenvolvidas pelos fabricantes de FPLD Altera e Xilinx, respectivamente.

2.1.2 Field Programmable Gate Array (FPGA)

Os FPGA foram disponibilizados comercialmente no início da década de 1980, pela empresa Xilinx Inc, como uma nova tecnologia para a implementação de lógica digital.

O FPGA é um circuito integrado programável, via *software*, que serve para implementar circuitos digitais, sendo constituído por um arranjo de Blocos Lógicos Configuráveis (CLB – *Configurable Logic Block*), por interconexões programáveis e por Blocos de Entrada e Saída (IOB – *Input/Output Block*). A organização dos blocos lógicos no FPGA se configura de forma bidimensional, e as interconexões estão distribuídas entre as linhas e as colunas dos blocos lógicos, na forma de trilhas horizontais e verticais, como pode ser visualizado na Figura 3.

Os FPGA representam uma evolução a partir de seus antecessores *Programmable Logic Arrays* (PLA), projetados para constituir a lógica de dois níveis (HAUCK, 1998 *apud* NASCIMENTO *et al.*, 2009). Diferentemente dos PLA convencionais, os FPGA podem suportar até milhões de portas lógicas, admitem a implementação de grandes quantidades de lógica digital multi-nível com estruturas combinacionais e sequenciais, como *dataflows*, *datapaths* e máquinas de estado finito (NASCIMENTO *et al.*, 2009)

A capacidade de ser programado em campo é uma característica importante presente nos FPGA. Com isso, a funcionalidade do dispositivo não é definida na fabricação do *chip*, e sim pelo projetista da aplicação final, por meio de *softwares* e placas (*kits*) de desenvolvimento específicas.

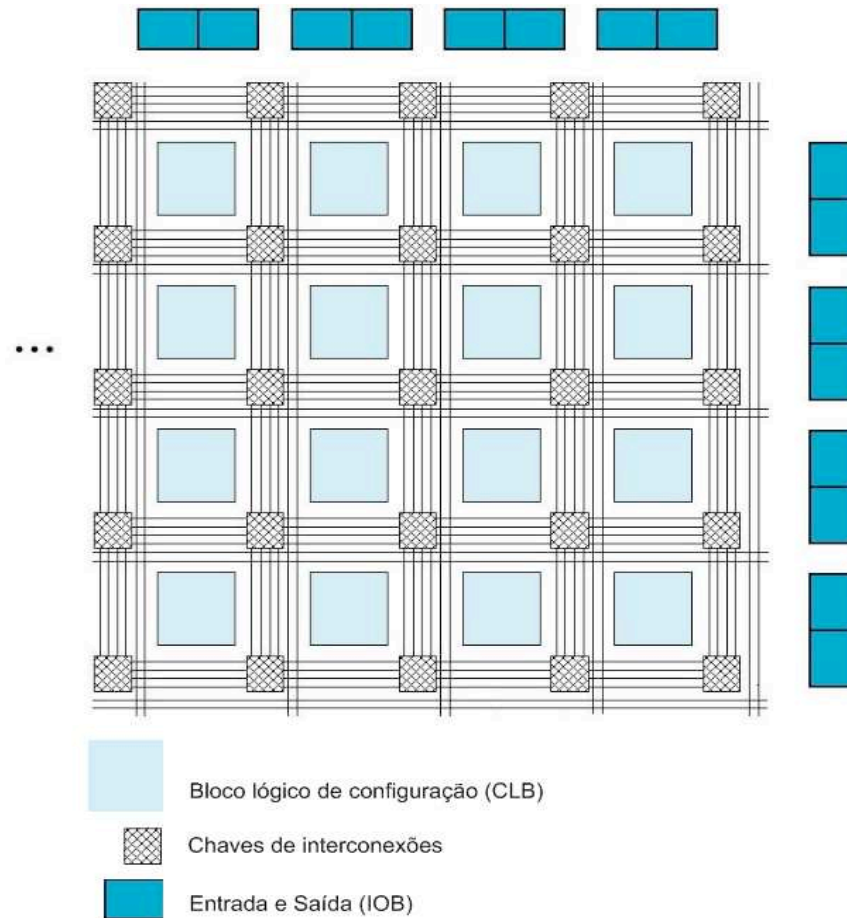


Figura 3 - Arquitetura de um FPGA

Fonte: (BRITO; ROCHA, 2009)

A funcionalidade dos CLB está relacionada à implementação das funções lógicas básicas, como o AND, o OR, o XOR e o NOT, como também das funções combinacionais mais complexas, tais como: decodificadores, codificadores, funções matemáticas, etc.

Com relação às interconexões, verificam-se que as saídas dos blocos lógicos se conectam às entradas de outros blocos. Um CLB não pode ser associado diretamente a pinos de entrada e saída. Primeiramente ele deve ser conectado a um IOB que, por sua vez, se conecta aos pinos de entrada e saída, por meio das linhas de roteamento (TOCCI, 2007).

2.1.2.1 Classificação dos FPGA

A classificação do FPGA, quanto a sua arquitetura, está relacionada com a granularidade, isto é, o tamanho e a complexidade das funções lógicas programáveis em seus

blocos lógicos, e com a estrutura de ligações programáveis (SKLIAROVA; FERRARI, 2003). De acordo com essas características, os CLB dividem-se em duas categorias distintas: as de granularidade fina e a de granularidade grossa.

Para a primeira categoria, os blocos lógicos podem implementar funções simples, servindo, deste modo, às manipulações ao nível de *bits* individuais e atingindo um elevado nível de utilização dos recursos lógicos disponíveis. A desvantagem apresentada por esses blocos lógicos está relacionada à utilização de muitos segmentos de vias de ligação e interruptores programáveis. Verifica-se, ainda, que os FPGA de granularidade fina possuem muitos pontos de configuração, necessitando de mais *bits* para serem configurados.

A segunda categoria representa os blocos lógicos de granularidade grossa que, normalmente, apresentam muitas entradas e servem bem à implementação de funções lógicas mais complexas. As operações dessas funções lógicas são executadas mais rapidamente e consomem menos área do que um conjunto de blocos lógicos elementares interligados de maneira adequada. Entretanto, se houver a necessidade de realizar operações lógicas simples, a arquitetura irá sofrer de uma baixa utilização dos recursos disponíveis (SKLIAROVA; FERRARI, 2003).

2.1.2.2 Tecnologias de programação

A programação de um FPGA é realizada utilizando-se chaves programáveis eletricamente. Estas chaves apresentam alguns atributos, tais como tamanho, impedância, capacitância e tecnologia de programação, os quais afetam a velocidade e o tempo de propagação dos sinais, e determinam as características de volatilidade e capacidade de reprogramação.

Atualmente existem essencialmente dois tipos de tecnologias de programação das chaves programáveis de roteamento, sendo que cada uma apresentará desempenho distinto dependendo do objetivo e da aplicação para a qual será empregada (CARRIÓN, 2001). Esses dois tipos de tecnologias de programação são a de programação (*Static Random Access Memory*) SRAM e a de programação por porta suspensa.

a) Tecnologia de programação (*Static Random Access Memory*) SRAM:

Implementa as conexões entre blocos lógicos através de portas de transmissão ou multiplexadores controlados por células SRAM. Os FPGA baseados nessa tecnologia são voláteis, ou seja, devem ser reprogramados e carregados quando energizados. Isto requer uma conexão a um PC ou uma memória externa permanente do tipo *FLASH* EEPROM para prover sua programação ao iniciar.

b) Tecnologia de programação por porta suspensa:

Essa tecnologia baseia-se nos transistores *Metal Oxide Semiconductor* (MOS), sobretudo construído com duas portas suspensas, semelhantes as usadas nas memórias EPROM e EEPROM. Tem a vantagem de permitir a reprogramação sem armazenamento de configuração externa, embora os transistores EPROM não possam ser reprogramados no próprio circuito, diferentemente das EEPROM, que podem ser reprogramadas eletronicamente (CARRIÓN, 2001).

2.1.2.3 Fabricantes

Dentre as principais empresas fabricantes de FPGA, destacam-se, no mercado mundial, são a Altera[®] e a Xilinx Inc. Cada uma dessas empresas possui várias famílias de dispositivos, cada uma delas direcionada a um conjunto de aplicações específicas.

Essas empresas oferecem suporte desde a fase de elaboração do projeto, por meio das ferramentas *Computer Aided Design* (CAD), até o fornecimento de núcleos de propriedade intelectual (*Intellectual Property cores* ou *IP cores*). Nos quadros 1 e 2 são referenciados alguns dispositivos FPGA, fabricados pelas empresas Altera[®] e Xilinx Inc, respectivamente, bem como são especificadas algumas de suas principais características.

Características	Arria II	Cyclone III	Stratix III
Dispositivo	EP2AGX95	EP3CLS70	EP3SL70
Elementos Lógicos	93.700	70.200	67.500
Pinos de I/O para uso geral	452	278	480
Total de RAM Mbits	5.4	2.9	2.2
Multiplicadores 18 x 18 bit	448	200	288

Quadro 1 - Resumo das famílias e características dos FPGA da Altera[®]
Fonte: (ALTERA, 2010)

Características	<u>Virtex-6</u>	<u>Virtex-5</u>	<u>Spartan-6</u>	<u>Extended Spartan-3A</u>
Elementos Lógicos	acima de 760.000	acima de 330.000	acima de 150.000	acima de 53.000
Pinos de I/O para uso geral	acima de 1200	acima de 1200	acima de 576	acima de 519
Total de RAM Bits	acima de 38 Mbits	acima de 18 Mbits	acima de 4.8 Mbits	acima de 1.8 Mbits
Multiplicadores embarcados para DSP	Sim (25x18 MAC)	Sim (25x18 MAC)	Sim (18x18 MAC)	Sim (18x18 MAC)

Quadro 2 - Resumo das famílias e características dos FPGA da Xilinx Inc

Fonte: (XILINX, 2010)

2.2 LINGUAGEM DE DESCRIÇÃO DE *HARDWARE*

A tendência mais recente, no campo dos sistemas digitais, é empregar linguagens baseadas em texto para a especificação de circuitos digitais, em substituição à entrada esquemática. Tais linguagens tornaram-se conhecidas como linguagens de descrição de *hardware* (HDL). Elas não só permitem descrever as interconexões estruturais entre os componentes, mas também o comportamento de seus componentes (VAHID, 2008).

As linguagens de descrição de *hardware* possuem semelhanças com as linguagens de programação, ou seja, devem seguir um conjunto de regras, conhecido como sintaxe da linguagem. Entretanto, não são linguagens de programação, pois visam descrever as interconexões estruturais entre os componentes, como também definir métodos que descrevam o comportamento dos componentes de *hardware*, respectivamente, chamados de descrição estrutural e de descrição comportamental.

A descrição de um circuito lógico, em um *hardware* reconfigurável, pode ser realizada por uma dentre as várias linguagens de descrição de *hardware* mais populares: VHDL, Verilog e AHDL. Tais linguagens apresentam capacidades semelhantes, diferindo, basicamente, em sua sintaxe (VAHID, 2008).

2.2.1 A linguagem de descrição VHDL

VHDL é o acrônimo de VHSIC *Hardware Description Language*, em que VHSIC é um acrônimo de *Very High Speed Integrated Circuit*.

Em uma breve definição, pode-se dizer que VHDL é uma maneira de se descrever, através de um programa, o comportamento de um circuito ou componente digital, em relação aos sinais de entrada aplicados e aos sinais de saída gerados.

2.2.1.1 História

No início da década de 1980, sob a supervisão do Departamento de Defesa dos Estados Unidos da América (DARPA), deu-se início a criação de uma ferramenta que permitiria documentar o projeto de desenvolvimento de circuitos integrados de alta velocidade, denominado VHSIC.

Devido à prioridade e à existência de várias empresas envolvidas no projeto, o Departamento de Defesa optou por definir os requisitos para o desenvolvimento de uma linguagem padrão na descrição de circuitos, para todos os envolvidos no projeto. Portanto, tratava-se de uma linguagem que permitiria às empresas descreverem e documentarem o circuito, independente do exemplar original.

Com todas as especificações e requisitos definidos pelo DARPA, inicia-se, em 1983, o desenvolvimento da linguagem básica para a descrição de *hardware*, pelas empresas IBM[®], Intermetrics[®] e Texas Instruments[®], designada *VHSIC Hardware Description Language* (VHDL).

Inicialmente, a linguagem VHDL supriu as necessidades de documentação do projeto VHSIC, porém, a necessidade maior estava em obter uma linguagem que proporcionasse uma descrição algorítmica, capaz de descrever um circuito, sem que houvesse a necessidade de especificar as ligações entre componentes.

Em 1987, ocorre o processo de padronização da linguagem pelo *Institute of Electrical and Electronic Engineer* (IEEE), a qual recebe a especificação de IEEE 1076-1987. Já em 1993, surge o IEEE 1164 (IEEE 1076-1993) acrescido de novas alterações, relacionadas, principalmente, ao tratamento de arquivos e ao acréscimo de valores que um sinal digital pode

assumir em tempo de execução. No ano de 2008 foi aprovado pelo *Review Committee* (REVCOM), o mais novo padrão IEEE 1076-2008 (ASHENDEN, 2008).

2.2.1.2 Características da linguagem

A linguagem VHDL, como qualquer outra linguagem, apresenta peculiaridades em sua composição. Uma dessas peculiaridades está relacionada ao fato de a linguagem suportar projetos com vários níveis de hierarquia, ou seja, uma descrição pode incidir na interligação de várias descrições menores (AMORE, 2005; GIACOMINI, 2009).

Outra característica que se pode relacionar é o modo concorrente para todos os comandos executados, ou seja, a seqüência dos comandos não implicará no comportamento da descrição. É permitido, além disso, especificar regiões de código sequencial, nas quais as execuções dos comandos obedecem à seqüência de apresentação.

A linguagem também aceita tanto a definição de estruturas no formato de procedimentos e funções, denominadas subprogramas, quanto emprego de caracteres maiúsculos ou minúsculos nas linhas de comando, ou seja, a VHDL não é *case sensitive*.

2.2.1.3 Vantagens e Desvantagens

As vantagens apresentadas na especificação de um sistema em VHDL são:

- Verificação, através de simulação, do desempenho de um sistema digital;
- Troca de informações sobre os projetos entre grupos de pesquisa, sem a necessidade de alteração;
- Permitem ao projetista considerar, no seu projeto, os *delay's* comuns aos circuitos digitais;
- A linguagem é independente da tecnologia utilizada;
- Facilidade de modificação dos projetos;
- Redução do custo e no tempo de na produção de um projeto;

Quanto às desvantagens, apenas uma é relevante: em VHDL não é possível gerar otimização de código.

2.3 ALGUMAS APLICAÇÕES RECENTES COM FPGA

Muitas pesquisas têm utilizado a arquitetura dos dispositivos reconfiguráveis, por exemplo, para melhorar o desempenho de suas soluções. Esse desempenho está relacionado, diretamente, com a redução de consumo de energia, o tempo de resposta e a qualidade dos resultados. A seguir serão descritos alguns trabalhos que empregaram a arquitetura reconfigurável dos FPGA em suas implementações.

Casillo (2005) apresenta o desenvolvimento de um processador *Reconfigurable Instruction Set Processors* (RISP) em FPGA, combinando as técnicas de configuração do conjunto de instruções do processador executadas em tempo de desenvolvimento, e de reconfiguração do mesmo em tempo de execução. Assim sendo, a criação e a combinação das instruções são realizadas mediante uma unidade de reconfiguração incorporada ao processador. Esta unidade permite ao usuário o envio de instruções customizadas ao processador, para que, depois, o mesmo possa utilizá-las como se fossem instruções fixas do processador.

O trabalho desenvolvido pelos autores Chelton e Benaissa (2008) detalha um novo conjunto de instruções específicas do processador (ASIP) de alta velocidade com *pipeline* para criptografia de curvas elípticas usando FPGA. Para isso, é produzido um novo algoritmo para realizar operações de duplicação de ponto e adição de ponto, com base em um conjunto específico de instruções, reduzindo o número de instruções por iteração. Nessa perspectiva, a essência desse trabalho mostra a combinação de varias técnicas para a obtenção de resultados práticos, ou seja, para a melhoria do desempenho em FPGA.

Souza (2008) propõe, em seu trabalho, um sistema completo com aquisição analógico-digital e transmissão de dados obtidos através do barramento USB para dispositivos portáteis, usando um método de compressão sem perdas, baseado em uma versão binária do algoritmo *Prediction by Partial Matching* (PPM), para sinais de ECG. Dessa forma, apresenta reduzida complexidade computacional, visando sua implementação em *hardware* por meio de modelagem com linguagem de descrição de *hardware*, utilizando o paradigma de desenvolvimento em FPGA para aplicações em equipamentos portáteis.

Gomes, Charão e Velho (2009b) descrevem, em seu trabalho, a implementação da Transformada Rápida de Fourier (FFT) em *hardware* reconfigurável (FPGA), com o propósito de aumentar o desempenho de aplicações numéricas. Essa implementação visa obter um ganho de desempenho em sistemas híbridos.

Em Lopes (2009), é apresentada uma abordagem para a implementação de arquiteturas complexas de redes neurais em FPGA, fazendo uso de vários processadores em um único *chip*. Nessa abordagem, foram usadas técnicas de processamento paralelo, através do desenvolvimento de algoritmos paralelos, para a obtenção de desempenho do sistema, em plataformas com múltiplos processadores.

Ao longo desta seção, foram introduzidos os tópicos básicos da computação reconfigurável e alguns de seus paradigmas, com suas vantagens e desvantagens. Também foram abordados alguns dispositivos reconfiguráveis, em especial o FPGA e a metodologia usada em sua programação, destacando-se o VHDL como linguagem de descrição para a programação de dispositivos de *hardware*. Na próxima seção, serão abordados os conceitos clássicos de redes neurais artificiais.

3 REDES NEURAIS ARTIFICIAIS

Baseadas no processamento em paralelo distribuído e nas conexões do sistema neural biológico, as Redes Neurais Artificiais (RNAs) constituem um paradigma computacional no campo da Inteligência Artificial (IA), para o desenvolvimento de sistemas computacionais capazes de simular tarefas intelectuais complexas, diferentemente dos paradigmas convencionais, que são constituídos por uma lógica sequencial (modelos de Von Neumann) (TAFNER, 2010).

As pesquisas em RNAs vêm sendo determinadas, desde o início, pela importância do cérebro humano no processamento das informações, as quais são completamente distintas das processadas por um computador digital convencional. O cérebro pode ser visto como um computador altamente complexo, não-linear e paralelo, o qual possui a capacidade de organizar os seus elementos estruturais, conhecidos por neurônios, de forma a executar seus processamentos de modo mais eficiente (HAYKIN, 2001).

As RNAs são sistemas de processamentos paralelos distribuídos, formados por unidades neuronais (neurônios artificiais), que, trabalhando em conjunto, são capazes de resolver problemas matemáticos de características não-lineares. Tais unidades são alocadas em uma ou mais camadas, com um grau de interconexão similar à estrutura cerebral, cuja transferência de informação entre as unidades ocorre em tempos específicos, dentro de uma margem de sincronização (NASCIMENTO; YONEYAMA, 2004; BRAGA; CARVALHO; LUDERMIR, 2007).

O poder computacional intrínseco em uma RNA é extraído, inicialmente, de sua estrutura maciçamente paralela. E, posteriormente, da sua capacidade de aprender e de generalizar a informação aprendida. Ou seja, a rede aprende a partir de um conjunto reduzido de padrões e produz respostas coesas para os dados que não estavam presentes na fase de treinamento. A auto-organização, o mapeamento de entrada-saída, a adaptabilidade e o processamento temporal são outras características presentes em uma rede neural que, aliadas às anteriores, fazem dela uma ferramenta hábil na resolução de problemas complexos (HAYKIN, 2001; BRAGA; CARVALHO; LUDERMIR, 2007).

3.1 MODELOS DE UM NEURÔNIO ARTIFICIAL

As redes neurais artificiais são compostas, em sua estrutura, por unidades computacionais denominadas neurônios artificiais. O neurônio artificial visa representar, de maneira simplificada, um neurônio biológico, por meio de uma formulação matemática. O modelo matemático do neurônio artificial desenvolvido pelos pesquisadores McCulloch e Pitts (1943) pode ser visto na Figura 4:

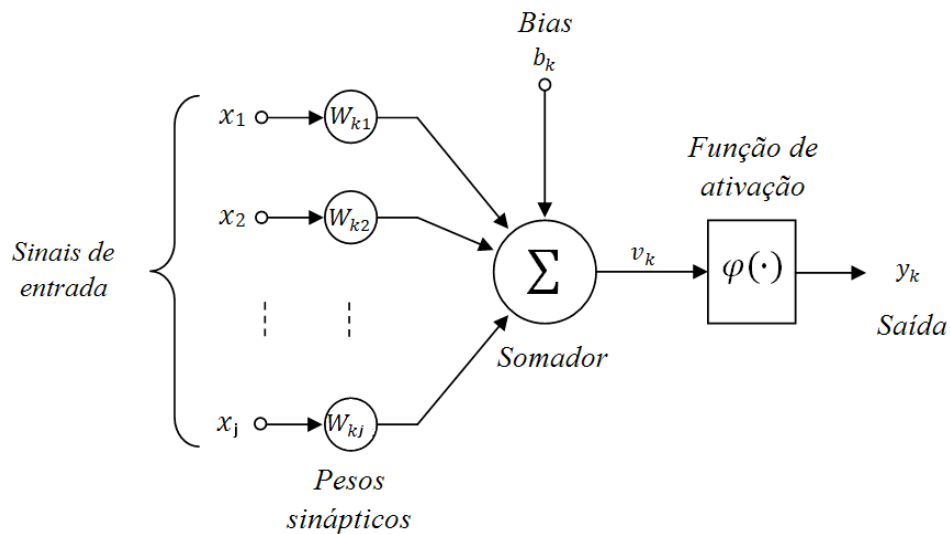


Figura 4 - Neurônio Artificial
 Fonte: (HAYKIN, 2001, pág. 36)

Com relação ao modelo proposto na Figura 4, são identificados três elementos básicos em sua composição, que são:

1. Um conjunto de *sinapses*, com cada uma sendo caracterizada por um peso. Assim, para todo sinal x_j na entrada da *sinapse* j vinculada ao neurônio k será ponderado um peso sináptico w_{kj} . Com relação ao peso sináptico w_{kj} , é importante observar que o primeiro índice refere-se ao neurônio em questão enquanto que o segundo faz alusão ao terminal de entrada da sinapse, ao qual o peso se refere;
2. Um combinador linear, empregado para executar o somatório dos sinais produzidos pelo produto entre os pesos sinápticos e as entradas fornecidas ao neurônio;
3. Uma função de ativação responsável por associar o sinal resultante do combinador linear, conhecido como potencial de ativação, a um valor de saída,

podendo aplicar não-linearidade ou restrição. Essa função define, ainda, uma restrição ao intervalo permissível de amplitude do sinal de saída a um valor finito. Geralmente essa limitação fica entre $[0, 1]$ ou $[-1, 1]$, por questão de normalização da informação (HAYKIN, 2001).

O exemplar neuronal da Figura 4 inclui um *bias* aplicado externamente, representado por b_k , o qual tem a finalidade de aumentar ou diminuir a entrada líquida da função de ativação, dependendo de seu valor positivo ou negativo. Assim, a descrição matemática de um neurônio artificial k pode ser representada pelas equações 1 e 2:

$$u_k = \sum_{j=1}^m w_{kj} x_j \quad (1)$$

e

$$y_k = \varphi(u_k + b_k) \quad (2)$$

com,

- x_1, x_2, \dots, x_m sendo os sinais de entrada;
- w_1, w_2, \dots, w_{km} sendo os pesos sinápticos do neurônio k ;
- u_k representando a saída do combinador linear;
- b_k representando o *bias*;
- $\varphi(\cdot)$ representando a função de ativação;
- y_k sendo o sinal de saída do neurônio.

A utilização do *bias* b_k tem a finalidade de empregar uma transformação afim à saída u_k do combinador linear do neurônio artificial da Figura 4, como pode ser observado na Equação 3. Ainda que o valor do *bias* b_k seja positivo ou negativo, a relação entre o potencial de ativação v_k do neurônio k e a saída do combinador linear u_k é modificada na forma mostrada na Gráfico 1. Com isso, pode-se notar como resultado da transformação afim que o gráfico de v_k em função de u_k não passa mais pela origem.

$$v_k = u_k + b_k \quad (3)$$

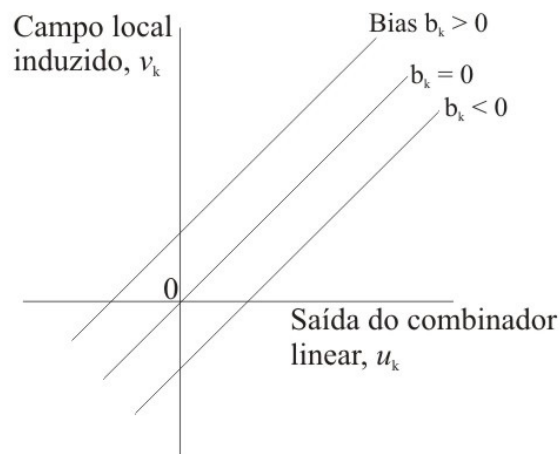


Gráfico 1 – Transformação afim produzida pela presença de um bias
 Fonte: (HAYKIN, 2001, pág. 37)

Outra forma de se representar o neurônio artificial k é considerar o *bias* como sendo o peso da sinapse (w_0), que receberá como sinal de entrada um valor fixo em “+1”, visualizado na Figura 5. Essa forma facilita a representação matemática do modelo do neurônio e o desenvolvimento dos algoritmos de aprendizado.

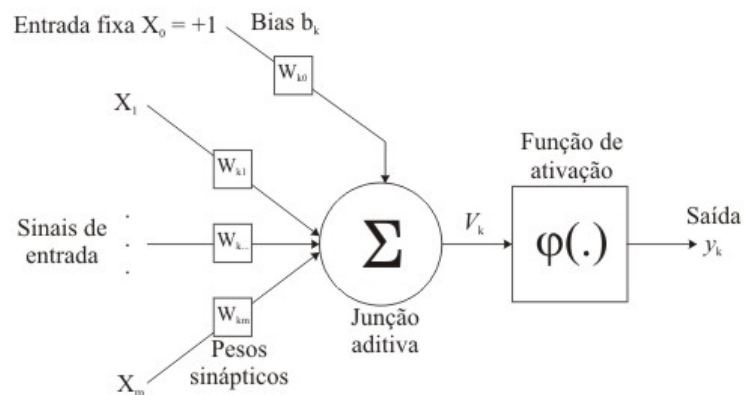


Figura 5 – Neurônio artificial e o *bias* como sendo peso da sinapse (w_0)
 Fonte: (HAYKIN, 2001, pág. 38)

3.1.1 Tipos de função de ativação

O valor de saída do neurônio, em termos de campo local induzido, é determinado por uma função de ativação, representada por $\varphi(v)$. Tal função pode assumir várias formas,

geralmente não-lineares. A seguir serão descritas três funções de ativação frequentemente utilizadas:

1. Função de Limiar ou *Heaviside* - Para esse tipo de função de ativação, ilustrado no Gráfico 2, o neurônio deve assumir o valor 1 (um) em sua saída, se o seu campo local induzido não for negativo, e 0 (zero) caso contrário, descrevendo-se deste modo a propriedade “tudo-ou-nada” do modelo de McCulloch-Pitts (HAYKIN, 2001):

$$y_k = \begin{cases} 1 & \text{se } v_k \geq 0 \\ 0 & \text{se } v_k < 0 \end{cases} \quad (4)$$

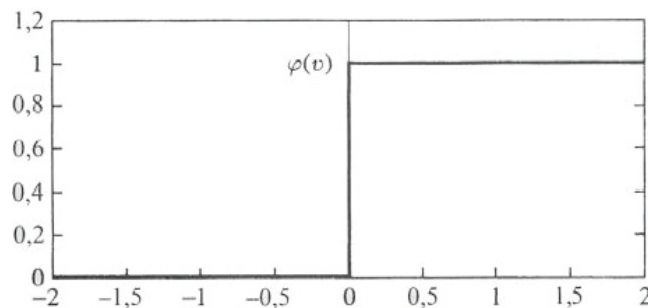


Gráfico 2 – Função de limiar
Fonte: (HAYKIN, 2001, pág. 39)

2. Função Linear por Partes - Essa função de ativação, mostrada no Gráfico 3, pode ser vista como uma aproximação de um amplificador não-linear. Podem ser observadas como formas especiais da função linear por partes quaisquer das seguintes situações:
 - Surgimento de um combinador linear caso a região linear de operação seja mantida sem entrar em saturação.
 - Se o fator de amplificação da região linear for feito infinitamente grande, a função linear por partes se reduzirá à função de limiar.

$$\varphi(v) = \begin{cases} 1, & v \geq +\frac{1}{2} \\ v, & +\frac{1}{2} > v > -\frac{1}{2} \\ 0, & v \leq -\frac{1}{2} \end{cases} \quad (5)$$

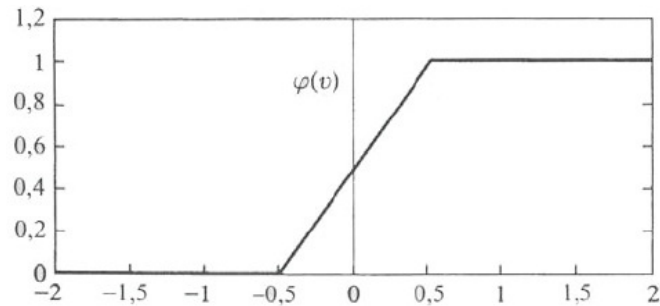


Gráfico 3 – Função de linear por partes

Fonte: (HAYKIN, 2001, pág. 39)

3. Função Sigmóide - É definida como sendo uma função crescente que exibe um equilíbrio adequado entre o comportamento linear e o não-linear. Variando-se o parâmetro de inclinação, definido por a , da função sigmoide, obtêm-se diferentes inclinações, como caracterizado no Gráfico 4. Um exemplo de função sigmóide é a função logística definida na Equação 6:

$$\varphi(v) = \frac{1}{1 + \exp(-av)} \quad (6)$$

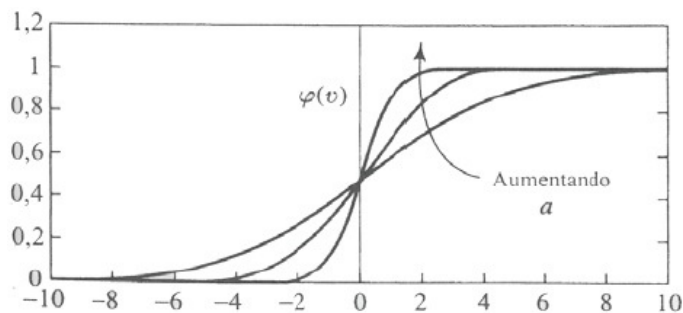


Gráfico 4 – Função sigmóide

Fonte: (HAYKIN, 2001, pág. 39)

No limite, a função sigmóide torna-se, simplesmente, uma função de limiar quando o parâmetro de inclinação aproxima-se do infinito. Enquanto os valores assumidos pela função de limiar se alternam entre 0 e 1, os da função sigmóide variam em um intervalo contínuo de valores entre 0 e 1.

Quanto às funções definidas até agora, observa-se que elas se estendem de 0 a +1. No entanto, às vezes, necessita-se que a função de ativação englobe um intervalo mais amplo, ou seja, que cubra a faixa de valores entre -1 e +1, admitindo, nesse caso, uma forma anti-

simétrica em relação à origem. Isto é, a função de ativação é uma função ímpar do campo local induzido.

$$\varphi(v) = \begin{cases} 1 & \text{se } v > 0 \\ 0 & \text{se } v = 0 \\ -1 & \text{se } v < 0 \end{cases} \quad (7)$$

Analogamente, na função sigmóide pode-se utilizar a função tangente hiperbólica, definida na Equação 8:

$$\varphi(v) = \tanh(v) \quad (8)$$

Admite-se, assim, que a função de ativação do tipo sigmóide assuma valores negativos, o que possibilita uma melhoria na convergência do algoritmo de treinamento.

3.2 ARQUITETURAS DE REDES NEURAIAS

O ponto fundamental na concepção de uma RNA, para uma dada aplicação ou resolução de problema, é a escolha de sua arquitetura. Nessa escolha enquadram-se, especialmente, as definições de parâmetros, tais como a quantidade de camadas, o número de neurônios artificiais em cada camada e o tipo de conexão a ser empregado na ligação dos neurônios (HAYKIN, 2001). Geralmente, as redes neurais se enquadram em três categorias de arquiteturas: a rede neural alimentada adiante com camada única, a rede neural com múltiplas camadas e a rede recorrente, as quais serão analisadas a seguir.

3.2.1 Rede neural alimentada adiante com camada única

Essa rede neural, ilustrada na Figura 6, caracteriza-se por apresentar uma arquitetura com uma camada de entrada de dados, seguida por uma única camada responsável pelo processamento dos sinais de entrada e pela geração dos sinais de saída. Esse tipo de rede é chamada “rede de camada única”, referindo-se, somente, à camada de saída de neurônios computacionais. A nomenclatura não considera a camada de entrada, porque não existe nela a possibilidade de computação de dados.

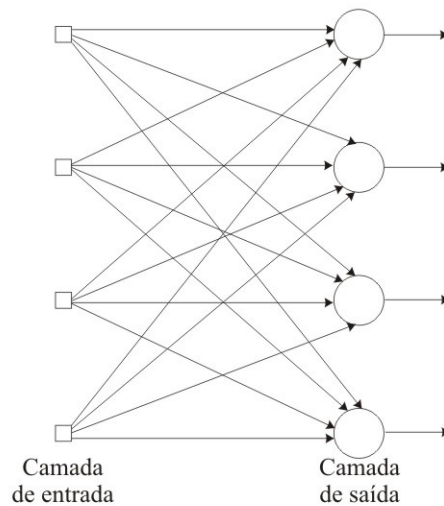


Figura 6 – Rede Neural Artificial com camada única alimentada adiante
 Fonte: (HAYKIN, 2001, pág. 36)

3.2.2 Rede neural com múltiplas camadas

É uma classe de rede neural conhecida como *perceptron*, de múltiplas camadas, que admite a presença de uma ou mais camadas ocultas e cujos nós computacionais são, correspondentemente, chamados de neurônios ocultos. Esses neurônios têm a função de intervir entre a entrada externa e a saída da rede de modo útil. Acrescentando-se uma ou mais camadas ocultas, a rede torna-se apta a extrair estatísticas de ordem elevada.

Para esse tipo de arquitetura, os sinais de entrada da rede são fornecidos pelos nós de fonte da camada de entrada aos neurônios da primeira camada oculta, que, por sua vez, processarão esses sinais, produzindo as saídas que servirão como entrada para a próxima camada oculta ou para a camada de saída. O conjunto de saídas produzido pelos neurônios da camada de saída estabelece a resposta global da rede. A Figura 7 representa o esquema de uma rede neural com a presença de uma camada oculta.

A rede neural reproduzida pela Figura 7 é dita completamente conectada se cada um dos neurônios de uma camada estiver conectado a todos os outros neurônios da camada seguinte. Caso exista pelo menos um neurônio sem ligação para outro neurônio, a rede é dita parcialmente conectada.

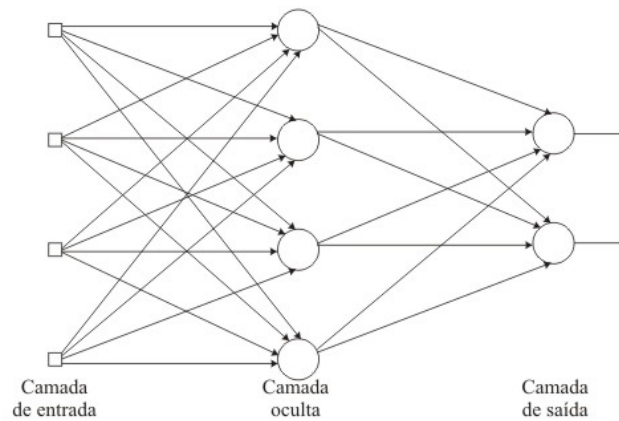


Figura 7 – Rede alimentada adiante com uma camada oculta e uma camada de saída
Fonte: (HAYKIN, 2001, pág. 48)

3.2.3 Rede recorrente

É uma arquitetura de rede neural que se distingue das demais, descritas anteriormente, por apresentar laços de realimentação e com ou sem camadas ocultas. A presença desses laços tem um impacto intenso na capacidade de aprendizagem e de desempenho da rede. Além disso, os laços de realimentação envolvem o uso de ramos particulares combinados com elementos de atraso unitário, procedendo, portanto, em um desempenho dinâmico não-linear, admitindo-se que a rede neural contenha unidades não-lineares. Essa rede é ilustrada na Figura 8.

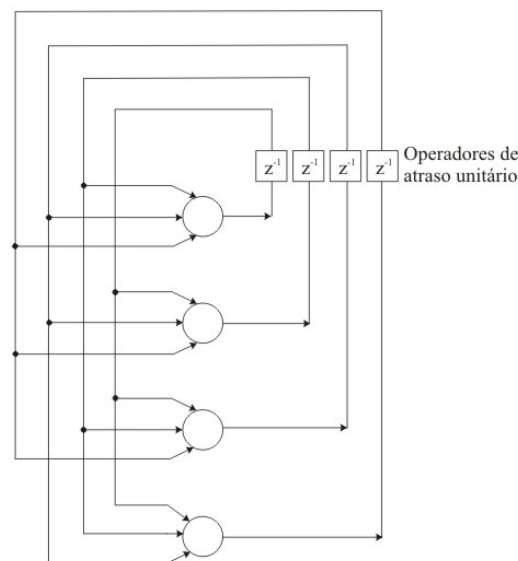


Figura 8 – Rede recorrente
Fonte: (HAYKIN, 2001, pág. 48)

3.3 PROCESSOS DE APRENDIZAGEM

Uma das características mais importantes de uma rede neural artificial é a sua capacidade de aprender a partir do ambiente, por meio de exemplos, conseguindo, com isso, melhorar seu desempenho. Isso é feito através de um processo iterativo de ajustes empregados aos seus parâmetros livres (pesos sinápticos e níveis de *bias*). O término do aprendizado ocorre quando a rede neural atinge uma solução generalizada para uma classe de problemas, segundo algum critério preestabelecido.

Quanto ao processo de aprendizagem, as redes neurais se classificam de acordo com dois paradigmas principais: aprendizado supervisionado e aprendizado não-supervisionado.

O aprendizado supervisionado caracteriza-se pela existência de um agente externo, responsável por apresentar um conjunto de padrões de entrada e de uma saída desejada, observando a saída calculada pela rede e comparando-a com a saída desejada. Caso a saída da rede não seja igual ou próxima ao valor desejado, os pesos sinápticos e níveis de *bias* devem ser ajustados iterativamente, sob a influência do sinal de erro, de forma a minimizar a diferença entre a saída gerada pela rede e a saída desejada.

Já no aprendizado não-supervisionado, não há a presença do examinador externo para acompanhar o processo de aprendizado. Nesse processo, apenas os padrões de entrada são fornecidos para a rede, ao contrário do aprendizado supervisionado, cujo conjunto de treinamento possui pares de entrada e saída. Durante esse aprendizado, verifica-se a existência de regularidades nos dados de entrada, por meio de suas características estatisticamente mais relevantes (BRAGA; CARVALHO; LUDERMIR, 2007).

O término do processo de aprendizagem se dá, essencialmente, quando a rede encontra uma solução para todas as entradas fornecidas. Contudo, outros parâmetros, também, podem ser levados em consideração, como, por exemplo, a definição de uma taxa de erro a ser alcançada ou o número máximo de ciclos.

3.3.1 Aprendizado supervisionado por retropropagação do erro

Em meio aos diversos algoritmos de aprendizado supervisionado vigentes na literatura, o algoritmo adotado pelo presente trabalho foi o da retropropagação do erro (*back-propagation*), que é o mais aplicado para treinar as redes *perceptrons* de múltiplas camadas.

O processo de desenvolvimento do algoritmo *back-propagation* envolve duas etapas. A primeira delas ocorre após as entradas serem apresentadas à rede e propagadas adiante até que um valor de saída seja computado pela última camada da rede. Esse valor deverá ser comparado com o valor da saída desejada, a fim de minimizar o erro da resposta atual em relação à saída desejada.

Na segunda etapa, o erro deverá ser propagado em direção à camada de entrada, ou seja, à camada contrária à de saída. Nessa etapa, o objetivo principal é a atualização dos parâmetros livres (pesos sinápticos), por meio da retropropagação do erro, podendo, ao término das duas etapas, ser apresentadas novas entradas à rede neural.

3.4 ALGORITMOS INTELIGENTES E REDES NEURAIIS

A Literatura da área dissemina várias pesquisas focadas na formulação de algoritmos fundamentados em técnicas de inteligência artificial, capazes de armazenar o conhecimento, aplicá-lo na resolução de problemas e adquirir novos conhecimentos por meio da experiência. Com relação a esses três aspectos, destaca-se o desenvolvimento de sistemas inteligentes baseados em redes neurais artificiais.

No entanto, os inteligentes, frequentemente, necessitam de uma rede de dispositivos distribuídos como, por exemplo, sensores, atuadores e controladores, que englobem amplas funcionalidades em termos de comunicação da rede, de integração dos seus elementos e do processamento das informações obtidas. Esses dispositivos devem empregar interfaces padronizadas e protocolos de comunicação, resultando, assim, em dispositivos reconfiguráveis e dotados de autonomia (CAGNI JÚNIOR, 2007). Um exemplo de dispositivo inteligente são os *Software Sensors*, que empregam códigos computacionais capazes de inferir valores a partir de medidas de diferentes sensores, realizando algum cálculo ou até mesmo algum tipo de controle.

Além disso, o trabalho publicado por Silva (2005) expõe o desenvolvimento de uma solução capaz de executar uma rede neural artificial no ambiente de redes, para a automação industrial *Foundation Fieldbus* (FF), baseado em blocos funcionais, para a compensação das medidas de temperatura através de *Software Sensors*.

Com relação à área de detecção de falhas, destaca-se o trabalho realizado por Fernandes *et al.* (2006), no qual foi desenvolvido um sistema inteligente firmado em redes

neurais artificiais, para a verificação e a classificação de falhas no controle de nível, em uma planta composta de dois tanques, cuja escoação gravita de um tanque para o outro. Três redes neurais foram desenvolvidas, duas delas simulando os tanques de nível, e a terceira recebendo os sinais de saída das redes iniciais. A realização da comunicação entre os componentes do sistema foi realizada por meio de um servidor *Open for Process Control* (OPC).

Além desse estudo, há, também, o trabalho publicado por Cagni Júnior (2007), que consiste na descrição de um sistema composto por uma placa processadora de sinais (DSP) e um supervisor, cuja principal finalidade é a correção dos dados aferidos por um medidor de vazão do tipo turbina. A correção é realizada por um algoritmo inteligente baseado em uma rede neural artificial.

3.5 REDES NEURAIS ARTIFICIAIS EM FPGA

Uma rede neural artificial pode ser implementada tanto por *software* quanto por *hardware*. Por causa do paralelismo intrínseco às RNAs, a sua implementação por *software* apresenta problemas inerentes à natureza sequencial da arquitetura de computador proposta por Von Neumann. Por outro lado, a implementação por *hardware* permite que se explore o paralelismo intrínseco das redes neurais artificiais. A RNA implementada em FPGA supre a falta de flexibilidade das implementações fundamentadas em circuitos integrados de aplicações específicas (ASIC).

Em geral, as redes neurais apresentam três características relevantes: paralelismo, modularidade e adaptação dinâmica. Paralelismo significa que todos os neurônios de uma mesma camada processarão os dados simultaneamente. Modularidade faz referência aos neurônios que possuem a mesma arquitetura estrutural (HAYKIN, 2001). As duas primeiras características evidenciam que os FPGA se adequam à prática das redes neurais artificiais, uma vez que esses dispositivos permitem a implementação de execuções simultâneas. Por fim, a adaptação dinâmica representa a existência de diversos algoritmos de aprendizagem que utilizam métodos para ajustar os pesos sinápticos e a topologia da rede (BRAGA, 2005).

As RNAs podem ser implementadas através da utilização de um *hardware* dedicado analógico ou digital. Com as implementações em *hardware* analógico, se obtêm circuitos mais compactos, capazes de realizar um processamento assíncrono de alta velocidade. Em contra partida, encontram-se os *hardwares* digitais que têm as vantagens de exatidão, baixa

sensibilidade ao ruído, programabilidade e adaptabilidade (MOLZ; ENGEL; MORAES 1999).

A realização de uma RNA em FPGA é justificada pela relação desempenho/custo e pela capacidade de reconfiguração do circuito, permitindo que a rede se adapte a diferentes aplicações. Como premissa, um dos desafios no projeto de redes neuronais em *hardware* é a análise de sua rapidez de resposta e sua eficiência frente a diferentes soluções, devendo-se levar em conta, também, suas restrições de área e de velocidade. (NACER; BARATTO, 2010).

Esta seção apresentou a base teórica sobre RNA, partindo da explanação do neurônio matemático, desenvolvido por McCulloch e Pitts, até a sua incorporação em uma arquitetura neural. Além disso, foram teorizados os tipos de função de ativação, as arquiteturas existentes e os processos de aprendizagem. Outros estudos abordados foram os relacionados aos algoritmos inteligentes e às redes neurais artificiais e, por último, tratou-se do uso das RNAs em FPGA. Na seção seguinte será discutido o método usado na implementação do neurônio básico em *hardware*, o qual será utilizado na construção de algumas topologias de redes neurais.

4 IMPLEMENTAÇÃO DO NEURÔNIO E DE REDES NEURAIS EM FPGA

Considerando que o objetivo maior deste trabalho é viabilizar a implementação de Redes Neurais Artificiais, totalmente em *hardware*, usando FPGA, o ponto de partida foi o desenvolvimento de sua unidade básica, o neurônio.

Assim, antes de partir para a descrição do projeto do neurônio, alguns pontos importantes que dizem respeito à notação numérica, ao paralelismo e à função de ativação devem ser esclarecidos.

Quanto à notação numérica, foi utilizada, para os valores de entrada, de saída, dos pesos sinápticos e do *bias*, a notação em ponto fixo, especialmente por falta de suporte aos valores não-inteiros das ferramentas de síntese, bem como pelo presumível aumento no número de CLBs a serem empregados, caso fosse utilizada uma notação em ponto flutuante.

Já com relação ao paralelismo, procurou-se preservá-lo entre os neurônios da mesma camada, entre as suas ligações e, internamente, entre os blocos que o compõem.

E quanto à função de ativação, foi implementada a tangente sigmóide, por meio da técnica de aproximação por interpolação linear.

4.1 DESCRIÇÃO ESTRUTURAL DO NEURÔNIO EM FPGA

Como forma de otimizar a implementação do neurônio artificial em FPGA, sua modularidade e o seu paralelismo, tendo como referência o neurônio proposto por McCulloch e Pitts, representado na Figura 4, item 3.1, a arquitetura do neurônio, proposta neste trabalho, denominada de NEURONIO, mostrada na Figura 9 e cuja descrição em VHDL encontra-se no Apêndice A, esta descrição foi dividida em dois blocos funcionais: o primeiro bloco é um combinador linear, responsável pelo somatório das estradas ponderadas pelos pesos sinápticos; o segundo é o bloco responsável pelos cálculos da função de ativação, denominados, respectivamente, blocos NET e FNET.

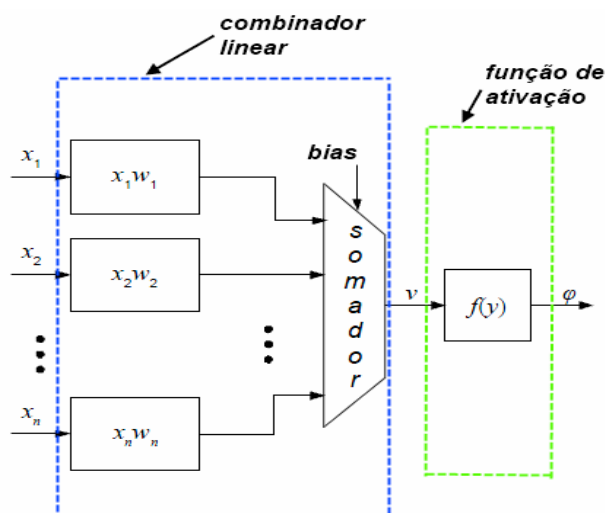


Figura 9 – Estrutura proposta para o neurônio em *hardware*

Os blocos NET e FNET, que compõem o bloco NEURONIO mostrado na Figura 10, foram descritos em VHDL. Um de seus diferenciais é a forma de implementação da função de ativação pelo bloco FNET, como será demonstrado ainda nesta seção.

O bloco NET, representado na Figura 11 e cuja descrição em VHDL é fornecida no Apêndice A, apresenta um caminho de dados (*datapath*) com uma unidade denominada MULTIPLICADOR e uma unidade denominada SOMADOR. Sua construção se baseou em uma abordagem de *RTL design* (projeto em nível de transferência entre registros), através da qual se garante paralelismo e maior sincronização nas transferências dos dados da entrada para a saída do neurônio, o que pode ser percebido pela presença dos dois registradores que complementam a sua organização. A função desse bloco é a de calcular o campo local induzido do neurônio, a partir de uma ou mais entradas de 16 *bits* que lhes são impostas.

Dessa forma, para processar a saída de um neurônio, primeiramente, deve-se realizar o cálculo da regra de propagação do campo local induzido. Esse cálculo é executado pela multiplicação de todas as entradas, x_1 a x_n , pelos seus correspondentes pesos sinápticos, w_{i1} a w_n , sucedidos pelo somatório dos valores resultantes dessas multiplicações mais o valor do *bias*.

Em uma arquitetura completamente paralela, toda entrada do neurônio tem um multiplicador exclusivo, o que faz com que a ponderação dos valores de entrada com os pesos sejam computados simultaneamente. Neste contexto, foi adotada uma arquitetura para o bloco NET onde prevalecesse o desempenho face à solução empregada de paralelizar o cálculo das multiplicações por meio da replicação do MULTIPLICADOR, caso haja mais de uma entrada para o neurônio. Uma organização com duas entradas de dados é mostrada na Figura 12, na

qual pode ser observada uma duplicação da unidade MULTIPLICADOR. Todavia, salienta-se que o único problema gerado por esse tipo de paralelização é o aumento na área utilizada em razão da replicação do bloco MULTIPLICADOR, caso existam muitas entradas de dados. Ele é desprezível, no entanto, considerando a grande quantidade de elementos lógicos presentes nos atuais FPGA.

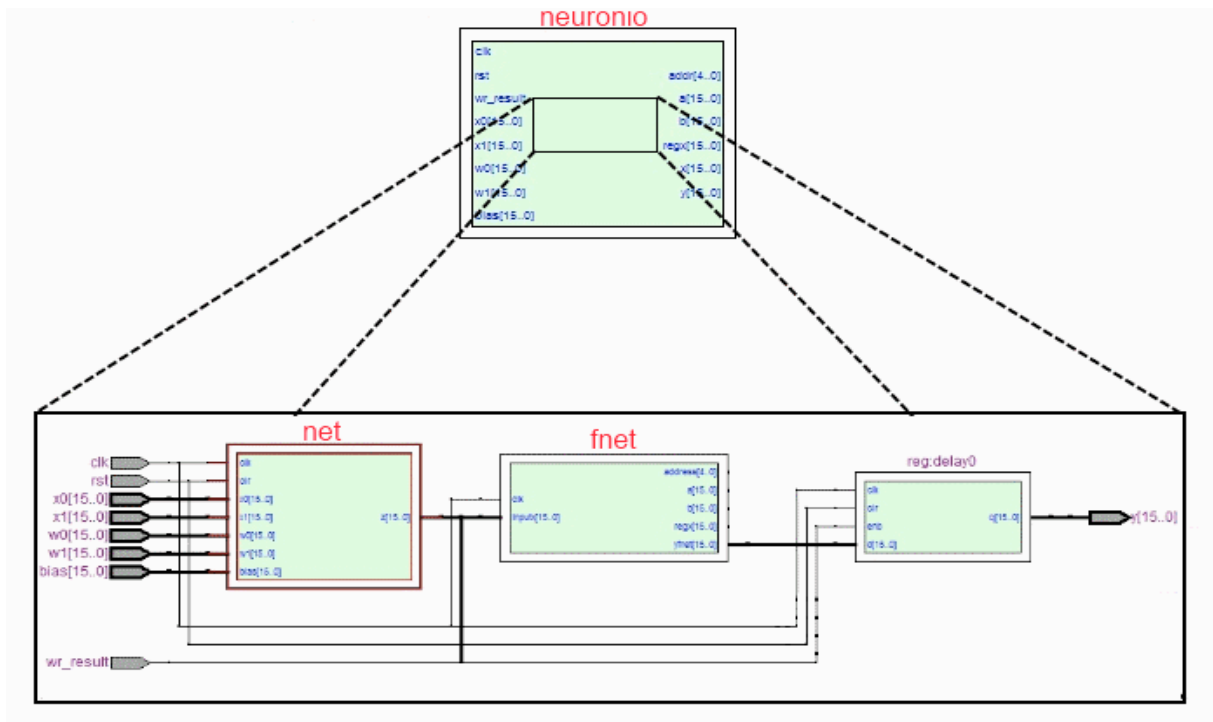


Figura 10 - Arquitetura do bloco NEURONIO

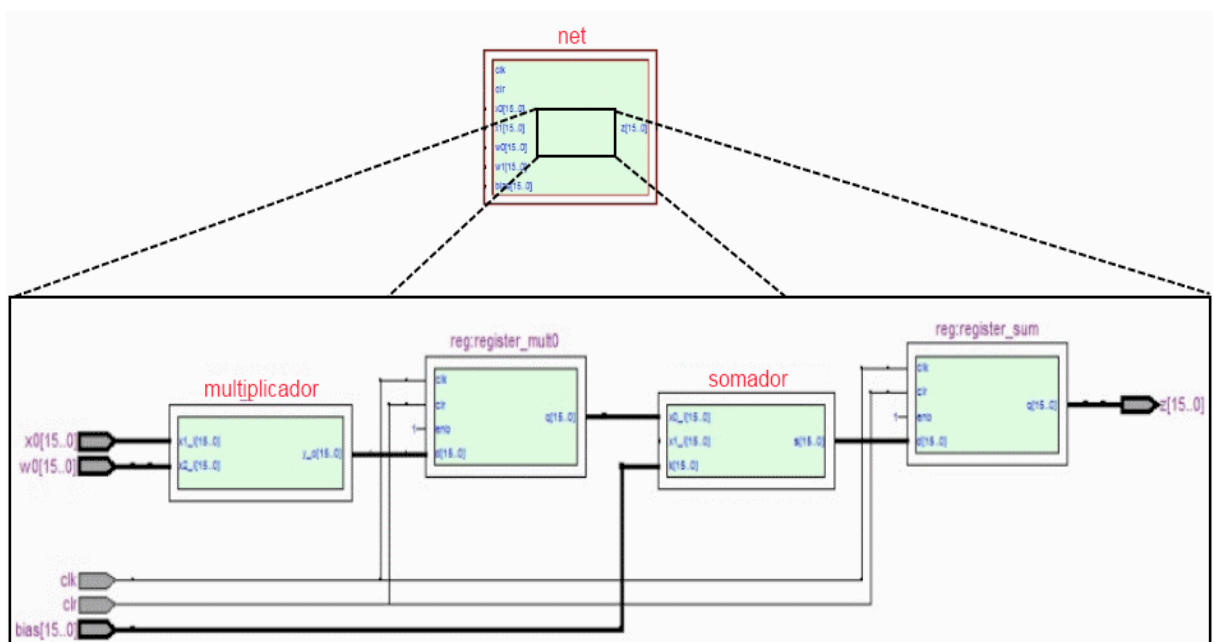


Figura 11 - Arquitetura do bloco NET

O SOMADOR é a unidade responsável pela soma dos resultados das multiplicações e do *bias*. Essa operação é viabilizada instanciando-se a entrada do *bias* e as saídas das diversas unidades multiplicadoras às entradas da unidade somadora.

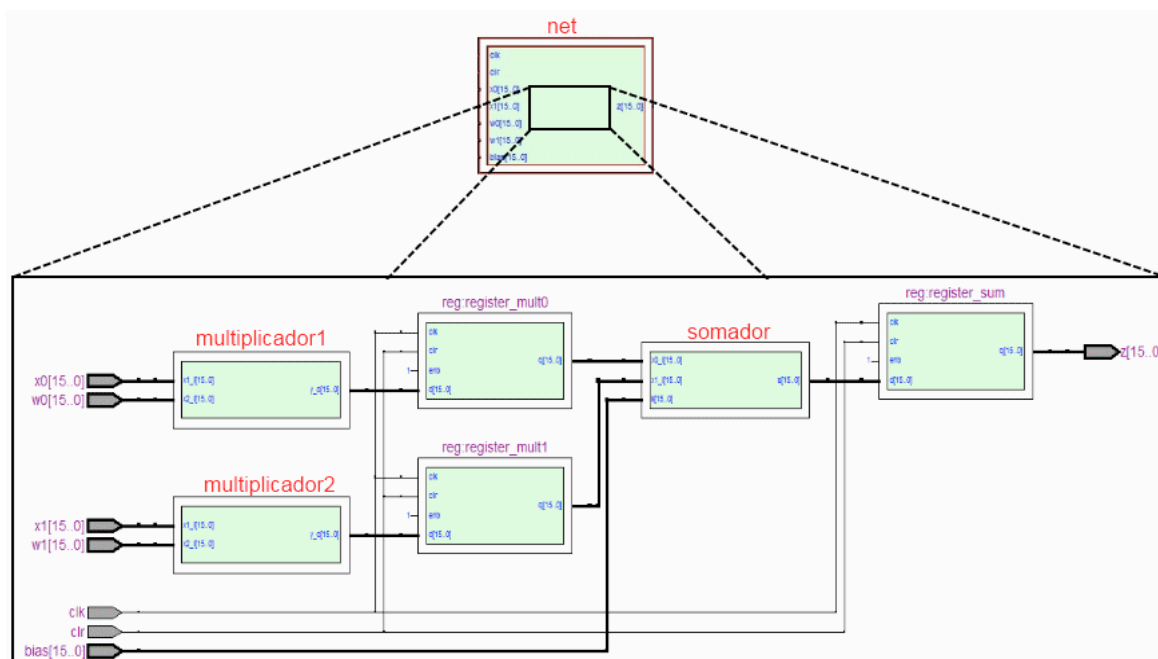


Figura 12 - Arquitetura do bloco NET com mais de uma entrada de dados

Deve-se observar que, no MULTIPLICADOR, além de ser feita a multiplicação de um valor de entrada x_i do neurônio, de tamanho, em ponto fixo de tamanho de 16 *bits* com sinal, pelo peso sináptico w_{ij} correspondente, também em ponto fixo de tamanho de 16 *bits* com sinal, gerando um resultado de 32 *bits* com sinal, mas não em notação de ponto fixo, deve ser feita a correção do número binário resultante da multiplicação para sua representação em ponto fixo com 16 *bits* com sinal, de tal forma que, na saída do MULTIPLICADOR, se tenha um valor em representação de ponto fixo compatível com a representação de ponto fixo do valor do *bias*.

Para a conclusão do bloco NEURONIO, após o cálculo do campo local induzido, prossegue-se com o cálculo da função de ativação. Neste caso a função utilizada foi a tangente sigmóide.

O bloco FNET, visualizado na Figura 13, é composto por uma *lookup table* unidade LUT, por uma unidade chamada de INTERPOLADOR e por duas unidades de atraso, usadas para a sincronização dos dados de entrada do INTERPOLADOR. A estrutura da LUT é constituída de um comparador e duas ROMs paralelas de 16 x 21 *bits* de dados. Uma das ROM armazena valores correspondentes aos coeficientes angulares de segmentos de retas, usados para a interpolação linear da função sigmoide; a outra ROM armazena valores

correspondentes aos coeficientes lineares dos mesmos segmentos de reta. Desse modo, o INTERPOLADOR é responsável pelo cálculo de saída do bloco FNET, a partir dos valores obtidos das duas ROMs que compõem a unidade LUT.

Nesse termos, o cálculo do valor de saída do bloco FNET é executado da seguinte maneira: a partir do valor de entrada, proveniente do bloco NET, é definido um endereço da LUT, comum às duas ROMs, nas quais estão presentes os correspondentes coeficientes angular e linear do segmento de reta, a serem usados pelo INTERPOLADOR, para a geração do sinal de saída. O código usado para definição do endereço de acesso aos coeficientes na *lookup table* pode ser encontrado na descrição VHDL do bloco FNET, fornecido no Apêndice A.

A saída da função de ativação foi montada a partir de uma tabela de 21 pontos, e os seus pontos intermediários foram obtidos por meio de uma interpolação linear. Na aquisição dos 21 pontos, utilizou-se uma técnica de inteligência computacional conhecida como algoritmos genéticos, cujo objetivo é a minimização do erro entre a função tangente sigmóide e a função a ser aproximada (SILVA, 2005).

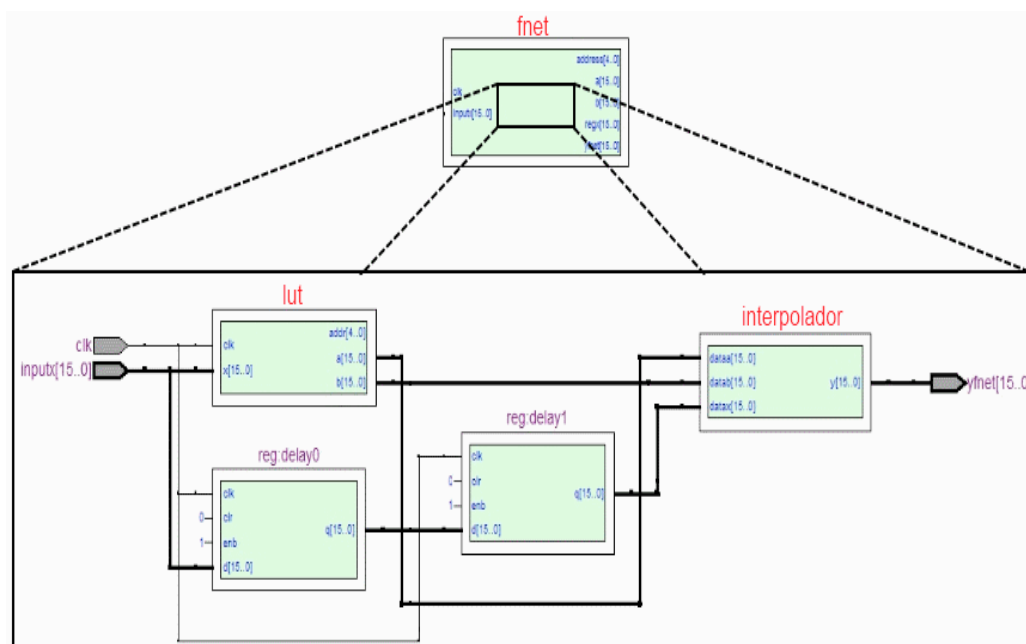


Figura 13 - Arquitetura do bloco FNET

4.2 DESCRIÇÃO ESTRUTURAL DAS REDES NEURAIS IMPLEMENTADAS EM FPGA

As soluções propostas para viabilizar o estudo das Redes Neurais Artificiais em FPGA tiveram como embasamento a descrição em VHDL do bloco NEURONIO. Desse modo, o

fluxo natural da implementação é a descrição de uma arquitetura de rede neural, mediante a replicação do bloco NEURONIO.

Com base no exposto anteriormente, quanto à replicação do bloco NEURONIO, foram construídas três estruturas de redes neurais, todas com características de um *perceptron* de múltiplas camadas. Na primeira delas implementou-se uma rede com duas entradas, dois neurônios na camada oculta e um neurônio na camada de saída; na segunda implementou-se uma rede com uma entrada, três neurônios na camada oculta e um neurônio na camada de saída; e na terceira foi implementada uma rede com uma entrada, com cinco neurônios na camada oculta e um neurônio na camada de saída. As descrições em VHDL dessas três estruturas de redes encontram-se no Apêndice A.

Nesse sentido, tomando como referência a construção da primeira rede neural, cujo esquemático em nível RTL é mostrado na Figura 14, pode-se observar a distribuição dos neurônios da camada oculta, NEURONIO0 e NEURONIO1, e do neurônio da camada de saída, NEURONIO2. Os esquemáticos, em nível RTL, das outras duas estruturas de redes neurais implementadas podem ser vistas no Apêndice B.

Para as análises funcionais dessas redes neurais, foram fornecidos os valores das entradas, os pesos e os *bias* correspondentes dos neurônios das camadas oculta e de saída, previamente definidos por implementações por *software*, com uso dos procedimentos detalhados em 4.3.

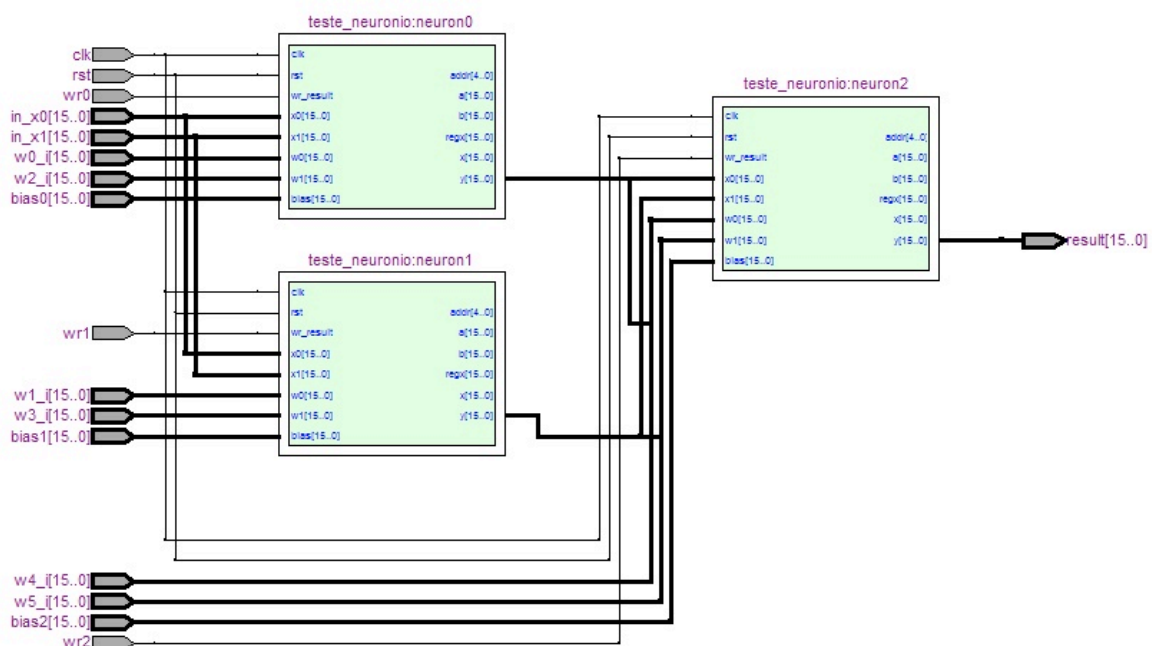


Figura 14 - Arquitetura de um *perceptron* de múltiplas camadas com dois neurônios na camada oculta e um neurônio na camada de saída

4.3 DEFINIÇÃO DOS PARÂMETROS DA REDE: PESOS SINÁPTICOS E *BIAS*

Considerando que o objetivo deste trabalho é a implementação de redes neurais em *hardware*, bem como sua validação com respeito à precisão, rapidez e robustez de resultados, foi feito todo um trabalho em *software*, usando o Matlab[®] e a programação em C, o que possibilitou produzir valores precisos que pudessem ser utilizados como parâmetros de entrada da rede neural em *hardware* a ser desenvolvida. Para tal, primeiramente, buscou-se desenvolver o algoritmo de aprendizado *backpropagation*, baseado na regra delta de Widrow-Hoff no MATLAB[®], com a utilização da *toolbox Neural Network Tool*. Em seguida, testou-se a viabilidade da RNA com alguns exemplos clássicos. Os exemplos escolhidos para a realização do teste foram simples e os resultados desejados são conhecidos, facilitando, dessa forma, a comparação entre a execução em *hardware* e a execução em *software*.

Assim sendo, com a finalidade de testar o *backpropagation* para camadas ocultas e resolver problemas não linearmente separáveis, foram escritos códigos de arquiteturas neurais mais complexas. Os problemas propostos foram o do OU-EXCLUSIVO (XOR), o da função exponencial e o da função *sinc*.

Os códigos foram desenvolvidos no MATLAB[®], utilizando-se as funções pré-definidas do *toolbox Neural Network Tool* (NNT), o que permitiu simular o funcionamento de uma porta lógica XOR com duas entradas, usando-se, para tanto, um único neurônio na camada de saída e dois neurônios na camada oculta. Para as funções *exponencial* e *sinc*, as arquiteturas neurais implementadas foram similares à anterior, diferenciando-se na quantidade de neurônios da camada de entrada, representada por um único neurônio, quanto no número de neurônios da camada oculta.

O treinamento da rede foi realizado através da apresentação um vetor de entradas com as respectivas saídas desejadas, analisando-se a diferença entre as saídas obtidas e as saídas desejadas. A diferença resultante era armazenada em uma variável, denominada erro, manipulada pelo algoritmo *backpropagation*, para atualizar os pesos das conexões da RNA.

Com o término da fase de treinamento e de posse de todos os pesos e *bias* atualizados, foram realizadas as validações das redes por meio de outras entradas antes não apresentadas. Isso foi realizado com o propósito de verificar a capacidade de generalização da rede, possibilitando, portanto, a validação da mesma, em um ambiente de programação, para os problemas propostos.

Nesta seção foram apresentados os métodos para a implementação do neurônio e de algumas redes neurais artificiais em FPGA, a partir da descrição estrutural do neurônio e de seus blocos internos, NET e FNET, até a finalização da descrição de uma RNA em FPGA. Na próxima seção serão apresentados os dados dos testes, simulações e resultados, das implementações do neurônio implementado, bem como de três redes executadas em FPGA.

5 TESTES, SIMULAÇÕES E RESULTADOS OBTIDOS

Nesta seção são apresentados os resultados das simulações e as informações resultantes da síntese em *hardware* do neurônio, e das topologias de redes neurais desenvolvidas.

Foram realizados cinco experimentos no total, dos quais dois foram exemplos numéricos para validar o processamento dos dados propagados no interior do bloco NEURONIO, com uma e duas entradas, dois foram exemplos numéricos para aproximação da função exponencial e da função *sinc*, respectivamente, e um foi usado na resolução do função XOR. Os três últimos exemplos foram usados para validar a implementação em *hardware* de redes neurais artificiais usando FPGA.

O método abordado para a construção das arquiteturas neurais dos experimentos foi o seguinte: inicialmente as redes foram escritas em *software* e treinadas usando-se o MATLAB[®], com o objetivo de extrair os seus pesos e *bias*. Posteriormente, os pesos e *bias* foram normalizados e transformados para ponto fixo. Assim, de posse desses dados, pôde-se dar início às simulações e às sínteses em *hardware* utilizando-se o ambiente de desenvolvimento e de prototipagem Quartus[®] II da Altera[®].

Neste contexto, almejou-se, também, demonstrar que as redes neurais do tipo *perceptrons* de múltiplas camadas produzidas em FPGA seriam capazes de produzir saídas adequadas para entradas que não estavam presentes durante o treinamento.

5.1 TESTES E SIMULAÇÕES FEITAS COM O BLOCO NEURÔNIO

Nesta etapa, para a validação do bloco NEURONIO, a ser utilizado como unidade básica na construção de redes neurais artificiais em FPGA, foram feitas simulações simples de propagação com uma e duas entradas de dados, usando valores de pesos e *bias* resultantes dos treinamentos. Os resultados das simulações do bloco NEURONIO, realizadas no Quartus[®] II, comparativamente aos resultados obtidos no MATLAB[®] estão apresentadas nos quadros 3 e 4.

Com base no Quadro 3, pode-se constatar um erro relativo máximo de 0,391% entre a saída obtida no MATLAB[®] e a saída obtida na implementação em FPGA.

	Saída do MATLAB			Saída do FPGA
	Em ponto flutuante (PFLUT)	Normalizado $N=PFLUT/5,3$	Em ponto fixo $PF=N*(2^{15})$	Em ponto fixo
Entrada	1,00	0,1887	6182	6182
Peso	1,7911	0,3379	11073	11073
<i>Bias</i>	-4,4975	-0,8486	-27806	-27806
Resultado	-0,9911	0,1870	-6127	-6103

Quadro 3 – Resultados das simulações com o neurônio usando uma única entrada

Enquanto que no Quadro 4, apresentado em seguida, o erro relativo é ainda menor, é de 0,0487%.

	Saída do MATLAB			Saída do FPGA
	Em ponto flutuante (PFLUT)	Normalizado (N) $=PFLUT/5,3$	Em ponto fixo (PF) $=N*(2^{15})$	Em ponto fixo
Entrada 1	0	0	0	0
Entrada 2	1,00	0,1887	6182	11073
Peso 1	-1,5661	-0,2955	-9682	-9682
Peso 2	-3,6876	-0,6958	-22799	-22799
<i>Bias</i>	0,6737	0,1297	4164	4164
Resultado	-0,9952	-0,1878	-6152	-6149

Quadro 4 – Resultados das simulações com o neurônio usando duas entradas

No Quadro 5 é mostrado um comparativo do número de elementos lógicos utilizados e da área de silício ocupada no FPGA, nas implementações do bloco NEURONIO com uma (primeira coluna) e duas entradas (segunda coluna). É possível observar nos dois casos que foram utilizados menos de 2% do número de elementos lógicos disponíveis.

	Bloco NEURONIO com uma entrada	Bloco NEURONIO com duas entradas
Dispositivo	EP2C35F672C6	EP2C35F672C6
Elementos lógicos	562 / 33.216 (2%)	736 / 33.216 (2%)
Total de registradores	112 / 33.216 (< 1%)	128 / 33.216 (< 1%)
Total de pinos	136 / 475 (29 %)	168 / 475 (29 %)
Número de <i>bits</i> de memória	0 / 483.840 (0%)	0 / 483.840 (0%)
Multiplicadores dedicados de 9 <i>bits</i>	4 / 70 (6%)	6 / 70 (9%)
Frequência do <i>clock</i>	54,16 MHz	54,16 MHz

Quadro 5 - Número de elementos lógicos utilizados e área de silício ocupada no FPGA, nas implementações do bloco NEURONIO com uma e duas entradas

5.2 TESTES E SIMULAÇÕES FEITAS COM AS REDES NEURAIS EM FPGA

5.2.1 Função exponencial

Para este teste, foi utilizada a estrutura de rede mostrada na Figura 15, com uma entrada, três neurônios na camada oculta e um neurônio na camada de saída. Como entrada, foi empregada uma função exponencial simples, dada por $y(x) = e^x$, a fim de se observar, com mais facilidade, o comportamento da rede neural.

No Quadro 6 estão representados os resultados obtidos com a simulação da rede em FPGA e os obtidos no MATLAB[®]. Com isso, pode-se constatar, para os quatro valores de entrada usados, um erro médio quadrático de aproximadamente 0,0036.

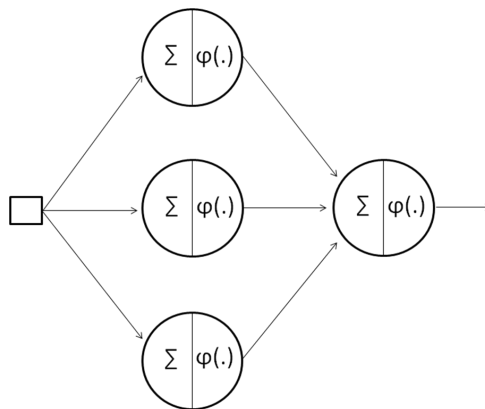


Figura 15 - Arquitetura da rede neural usada para executar a função exponencial

Entrada	Saída do MATLAB em ponto flutuante	Saída do MATLAB em ponto fixo	Saída do FPGA em ponto fixo
-0,9	0,4065	1687	1678
-0,7	0,4965	2061	2064
-0,4	0,6703	2783	2770
-0,1	0,9048	3756	3763

Quadro 6 – Resultados obtidos para a função exponencial

A Figura 16 contém informações quanto à quantidade de elementos lógicos utilizados e da área de silício ocupada no FPGA, pela rede. Esses dados foram obtidos a partir da

compilação feita no *software* Quartus® II da Altera®. Pela figura, observa-se uma taxa de uso de elementos lógicos de apenas 8%.

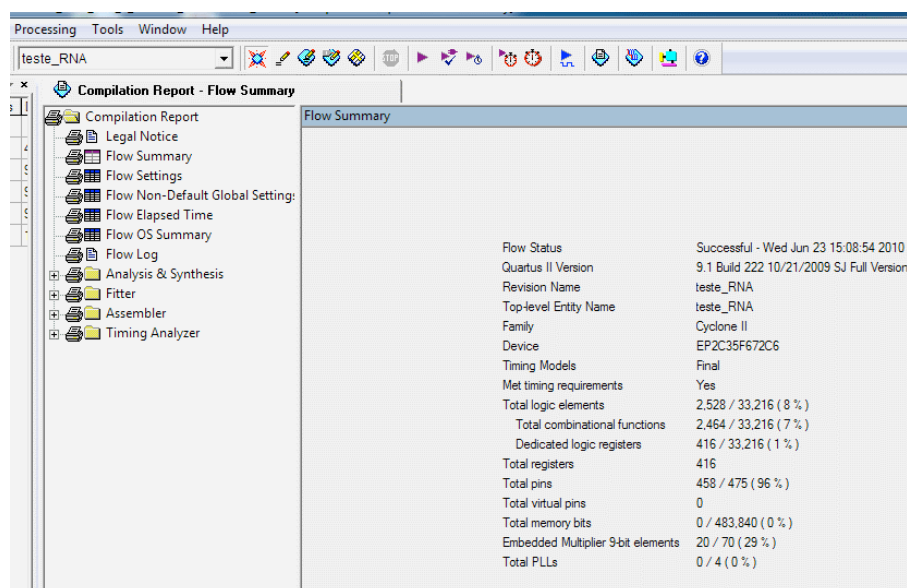


Figura 16 - Quantidade de elementos lógicos e a área de silício utilizada do FPGA

5.2.2 Função *Sinc*

Neste exemplo, a arquitetura neural empregada também foi a dos *perceptrons* de múltiplas camadas, quase idêntica à do exemplo da exponencial analisada anteriormente, diferenciando-se apenas na quantidade de neurônios da camada oculta, já que foram usados cinco neurônios, e no valor usado para a normalização dos valores da rede, igual ao maior valor em módulo dos dados obtidos no treinamento da rede *off-line*. A partir deste maior valor, todos os outros foram normalizados e transformados para ponto fixo para, daí, serem aplicados à rede, juntamente com a função *sinc* de entrada, visualizada no Gráfico 5.

Verifica-se que as saídas do FPGA, quando comparadas à saídas obtidas no MATLAB®, comportaram-se bem próximo ao esperado, como pode ser visto no Quadro 7, o que confirma a possibilidade de se ter como base os dados simulados.

A Figura 17 contém informações obtidas a partir da compilação e síntese no *software* Quartus® II da Altera®. Observa-se um pequeno aumento na quantidade de unidades lógicas utilizadas com relação à implementação anterior. No entanto, o valor final está muito próximo de apenas 12% da quantidade total de unidades lógicas do FPGA.

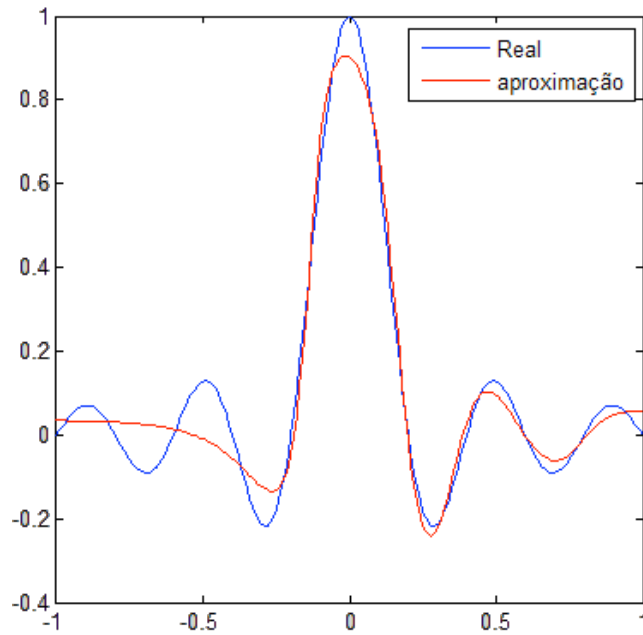


Gráfico 5: Resultados da função *Sinc* executada no MATLAB®

Entrada	Saída do MATLAB em ponto flutuante	Saída do MATLAB em ponto fixo	Saída do FPGA em ponto fixo
-0,2	-0,0446	-131	-125
-0,1	0,7029	2068	2057
0	0,9003	2648	2644
0,3	-0,2234	-657	-629
0,5	0,0946	278	267

Quadro 7 – Resultados comparativos da rede neural utilizada para aproximar a função *Sinc*

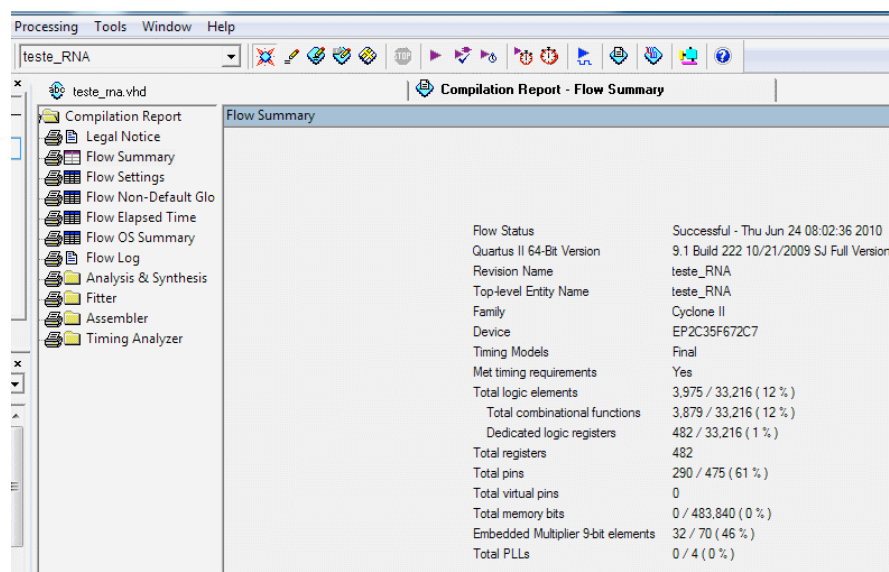


Figura 17 – Dados de síntese obtidos no *software* Quartus® II da Altera®

5.2.3 XOR

Para a execução do XOR, utilizou-se uma arquitetura neural com duas entradas, dois neurônios na camada oculta e um neurônio na camada de saída, representada na Figura 18. Apesar de simples, o problema foi suficiente, também, para mais uma vez validar a descrição de redes neurais em FPGA, visto que, como sempre, os pesos e os *bias* foram extraídos do treinamento realizado no MATLAB[®], utilizando-se de uma arquitetura neural idêntica à desenvolvida em *hardware*.

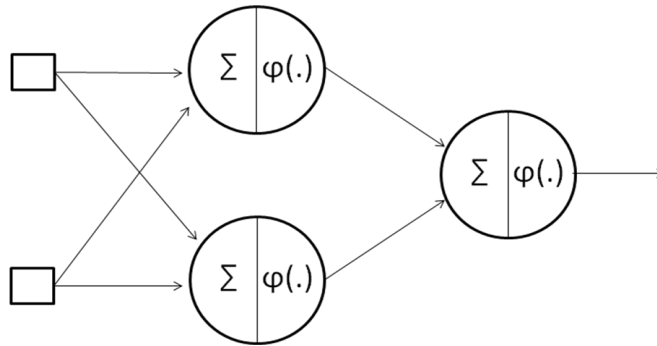


Figura 18 - Arquitetura da rede neural usada para executar a função XOR

Entrada	Saída desejada	Saída do MATLAB ponto flutuante	Saída do MATLAB ponto fixo	Saída do FPGA
00	0	0,0145	103	105
01	1	0,9187	5657	5651
10	1	0,9233	5679	5672
11	0	0,0136	91	90

Quadro 8 – Resultados comparativos da rede neural utilizada para execução da função XOR

Observa-se que a implementação da rede neural, a partir do modelo fornecido, viabilizou a resolução do problema do XOR; e os valores das saídas obtidos, mostrados no Quadro 8 validam a implementação feita no FPGA, apresentando uma aproximação extremamente relevante para a saída da rede neural, com um erro relativo máximo de apenas 0,25%. Na Figura 19 são mostradas duas telas obtidas no Quartus[®] II, após a simulação do XOR.

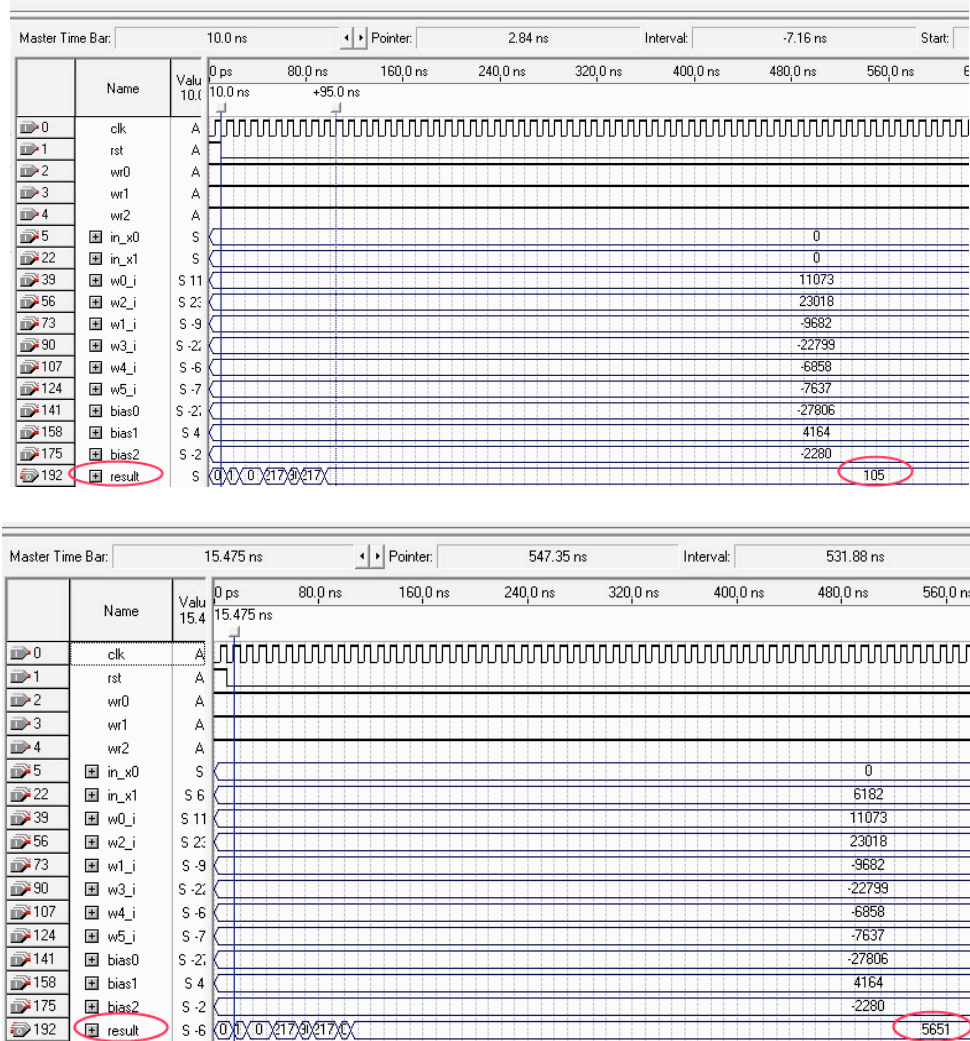


Figura 19 – Resultados de simulação da função XOR

No Quadro 9 são apresentados o total de dispositivos usados e o total de área ocupada no FPGA EP2C35 pela rede neural na resolução do XOR. Como se pode observar, as taxas de uso são baixas, evidenciando, dessa forma, que podem ser implementadas redes muito maiores.

Dispositivo	EP2C35F672C6
Elementos lógicos	2.205 / 33.216 (7%)
Total de registradores	384 / 33.216 (1%)
Total de pinos	436 / 475 (92 %)
Número de <i>bits</i> de memória	0 / 483.840 (0%)
Multiplicadores dedicados de 9 <i>bits</i>	18 / 70 (26%)
Frequência do <i>clock</i>	54,31 MHz

Quadro 9 – Taxas de ocupação do FPGA na execução da função XOR

Nesta seção foram apresentados dados de testes, simulações e resultados obtidos nas implementações do neurônio e de três redes implementadas em FPGA. Na próxima seção serão apresentadas as conclusões, algumas considerações finais e encaminhamentos para trabalhos futuros.

6 CONCLUSÃO

O presente estudo mostrou a viabilidade da construção de arquitetura de redes neurais em dispositivos reconfiguráveis do tipo FPGA, a partir da descrição de um neurônio básico.

Neste contexto, foi feita a descrição de um neurônio usando notação numérica de ponto fixo, e empregando uma estratégia para a implementação da função de ativação do tipo tangente sigmóide, desenvolvida a partir da utilização de uma interpolação linear, com uso de *lookup table* e de blocos multiplicadores em paralelo.

Com a especificação do bloco NEURONIO, constituído pelos blocos NET e FNET, verificou-se que as descrições das arquiteturas utilizadas neste estudo tornaram-se bastante modulares, possibilitando facilmente o aumento ou a diminuição do número de neurônios, bem como a modificação da estrutura da rede. Em decorrência disto, foram formadas redes neurais completas inseridas em FPGA, com uma adequada precisão numérica e capacidade de processamento paralelo.

Assim, as arquiteturas neurais desenvolvidas em um dispositivo reconfigurável presente na placa de desenvolvimento DE2 da Altera[®], com FPGA EP2C35, demonstraram eficiência na resolução da lógica XOR e precisão de valores obtidos durante a execução das funções exponencial e *sinc*.

As limitações encontradas durante a execução deste trabalho estiveram relacionadas com a notação em ponto flutuante, não suportada pelas ferramentas de síntese, e, também, com a implementação da função de ativação do tipo tangente sigmóide. Com isso, as estratégias adotadas para superar tais limitações foram, primeiramente, usar a notação numérica em ponto fixo. O método utilizado para a realização do cálculo da função de ativação foi o uso de *lookup table* para o armazenamento dos valores da função de transferência obtidos, com a realização *off-line* do treinamento da rede neural no MATLAB[®].

A arquitetura da rede neural desenvolvida em FPGA, apesar de simples, qualificou os métodos e as abordagens desenvolvidas, como aptos para o transporte da fase de simulação para sistemas reais, atendendo, dessa forma, aos requisitos estabelecidos para o projeto.

Os estudos de caso realizados comprovaram a importância de se conhecer como as ferramentas de síntese lógica e física se comportam. Isto ocorreu em virtude da necessidade de se substituir a lógica formal de algumas descrições usuais de redes neurais, por lógicas alternativas que fornecessem resultados, se não semelhantes, ao menos muito aproximados. Um exemplo disso foi a dificuldade na implementação da função de ativação.

Contudo, diante do que foi exposto, com os resultados deste estudo, buscou-se contribuir para a área do ensino e pesquisa, com a disseminação do conhecimento sobre redes neurais em *hardware* que possibilitem o desenvolvimento de trabalhos futuros. Nessa ordem de ideias, vale contribuir com sugestões para trabalhos futuros, tais como: a construção de estruturas neuronais, capazes de suportar mais de uma função de ativação, ou a implementação de arquiteturas de redes neurais com pesos e *bias* não fixos, ou seja, eles estariam armazenados em *lookup table*. Por último, o desenvolvimento de uma matriz de neurônios, cujas interligações entre eles sejam reconfiguráveis graficamente, na qual possam ser desenvolvidos diversos modelos de redes neurais em FPGA.

REFERÊNCIAS

ALTERA. Devices. 2010. Disponível em:< <http://www.altera.com/products/devices/dev-index.jsp>>. Acesso em: 26 mar 2010.

AMORE, Robert D'. **VHDL: descrição e síntese de circuitos digitas**. Rio de Janeiro: Ltc, 2005.

ARAGÃO, A.; ROMERO, R.; MARQUES, E. **Computação Reconfigurável Aplicada á Robótica (FPGA)**. Disponível em: <<http://www2.eletronica.org/artigos/eletronica-digital/computacao-reconfiguravel-aplicada-a-robotica-fpga>>. Acesso em: 23 nov. 2009.

BRAGA, A. L. S. **VANNGen: uma ferramenta CAD flexível para a implementação de redes neurais artificiais em hardware**. 2005. 135 f. Dissertação (Mestrado) - Universidade de Brasília, Brasília, 2005.

BRAGA, A. P.; CARVALHO, A. C. P. L. F.; LUDEMIR, T. B. **Redes neurais artificiais: teoria e aplicações**. 2. ed. Rio de Janeiro: Ltc, 2007. 226 p.

BRITO, A. V.; ROCHA, R. G. C. Uma Introdução aos Sistemas Dinamicamente reconfiguráveis. Disponível em:< http://www.unibratex.com.br/revistacientifica/n2_artigos/n2_brito_av.pdf>. Acesso em: 26 nov. 2009.

CAGNI JÚNIOR, Eloi. **Software Inteligente Embarcado Aplicado à Correção de erro na Medição de Vazão em Gás Natural**. 2007. 72 f. Dissertação (Mestrado) - Departamento de Engenharia da Computação, Universidade Federal do Rio Grande do Norte - Ufrn, Natal, 2007.

CARRIÓN, D. **FPGA**. 2001. Disponível em: <<http://www.leopoldina.cefetmg.br/moodle/login/index.php>>. Acesso em: 20 ago. 2009.

CASILLO, L. A. **Projeto e implementação em FPGA de um processador com conjunto de instrução reconfigurável utilizando VHDL**. 2005. 110 f. Dissertação (Mestrado) - Universidade Federal do Rio Grande do Norte, Natal-rn, 2005.

CHELTON, W. N.; BENAÏSSA, M. Fast elliptic curve cryptography on FPGA. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, New Jersey, v. 16, n. 6, p. 198-205, fev. 2008.

FERNANDES, R. G. *et al.* **Identificação neural de um sistema de níveis em ambiente *foundation fieldbus***. Disponível em:< <ftp://users.dca.ufrn.br/artigos/2006/clca2006.pdf>> Acesso em: 12 jan 2010.

GIACOMINI, R. **Apostila básica de VHDL**. Disponível em <http://ipsulinha.net78.net/materias/labi1/giacomini/vhdl.rtf>>. Acesso em: 12 nov. 2009.

GOMES, V. C. F.; CHARÃO, A. S.; VELHO, H. F. C. **Field Programmable Gate Array - FPGA**. Disponível em: <<http://www.vconrado.com/chr/fpga.pdf>>. Acesso em: 19 dez. 2009.

GOMES, V. C. F. ; CHARAO, A. S. ; VELHO, H. F. C.. **Implementação de FFT em Hardware Reconfigurável**. In: 23a. Jornada Acadêmica Integrada – UFSM, 2009, Santa Maria. Anais da 23a. Jornada Acadêmica Integrada – UFSM, 2009.

HASSAN, A. A.; ELNAKIB, A.; ABO-ELSOUD, M. **FPGA-Based Neuro-Architecture Intrusion Detection System**, *Proceedings of the Internatinal Conference on Computer Engineering & Systems*, Cairo, pp. 268-273, 2008.

HAYKIN, S. **Redes neurais: princípios e práticas**. 2. ed. Porto Alegre: Bookman, 2001. 900 p.

LOPES, D. C. **Implementação e avaliação de maquina de comitê em um ambiente com múltiplos processadores embarcados em um único chip**. 2009. 107 f. Tese (Doutorado) - Unuversidade Federal do Rio Grande do Norte, Natal, 2009.

LUDWIG Jr., O.; COSTA, E. M. M. **Redes Neurais: Fundamentos e Aplicações com Programas em C**. Rio de Janeiro: Editora Ciência Moderna LTDA, 2007.

MARTINS, C. A. P. da S. *et al.* **Computação Reconfigurável: conceitos, tendências e aplicações**. Disponível em: <http://www.ppgee.pucminas.br/gsd/papers/martins_eri02.pdf>. Acesso em: 05 jun. 2009.

MOLZ, R. F.; ENGEL, P.M.; MORAES, F. G. **Uso de um Ambiente Codesign para a Implementação de Redes Neurais**. *Proceedings of the IV Brazilian Conference on Neural Networks - IV Congresso Brasileiro de Redes Neurais* pp. 013-018, July 20-22, 1999 - ITA, São José dos Campos - SP – Brazil.

NACER, H.C.; BARATTO, G. **Rede Neural Artificial em Hardware Reconfigurável**. Disponível em:< <http://www.usp.br/siicusp/Resumos/16Siicusp/5015.pdf>>. Acesso em: 12 jan. 2010.

NASCIMENTO Jr, C. L.; YONEYAMA, T. **Inteligência artificial em controle e automação**. São Paulo: Edgard Blücher, 2004.

NASCIMENTO, P. S. B. do *et al.* **Sistemas Reconfiguráveis: novo paradigma para o desenvolvimento de aplicações de computação massiva de dados**. Disponível em: <<http://www.unibrattec.com.br/jornadacientifica/diretorio/NOVOSIS.pdf>>. Acesso em: 15 dez. 2009.

NAVABI, Z. **Digital Design and Implementation with Field Programmable Devices**. United States: Kap, 2005.

SILVA, D. R. C. **Redes Neurais Artificiais no ambiente de redes industriais foundation fieldbus usando blocos funcionais padrões**. 2005. 67 f. Dissertação

(Mestrado) - Departamento de Engenharia da Computação, Universidade Federal do Rio Grande do Norte - UFRN, Natal, 2005.

SKLIAROVA, I. ; FERRARI, A. B. Introdução à computação reconfigurável. **Revista do DETUA**, v. 2, n. 6, p. 1-16, set. 2003. Disponível em: < http://www.ieeta.pt/~iouliia/Papers/2003/1_SF_ETSet2003.pdf>. Acesso em: 15 dez. 2009.

SOUZA, A. R. C. de. **Desenvolvimento e Implementação em FPGA de um sistema portátil para aquisição sem perdas de eletrocardiogramas**. 2008. 160 f. Dissertação (Mestrado) - Universidade Federal da Paraíba - UFPB, João Pessoa, 2008.

TAFNER, Malcon Anderson. **As Redes Neurais Artificiais: aprendizado e plasticidade**. Disponível em:<<http://www.cerebromente.org.br/n05/tecnologia/plasticidade2.html>>. Acesso em: 30 mar. 2010.

TOCCI, R. J.; WIDMER, N. S.; MOSS, G. L. **Sistemas digitais: princípios e aplicações**. 10. ed. São Paulo: Person Prentice Hall, 2007.

VAHID, F. **Sistemas digitais: projeto, otimização e HDLS** / Frank Vahid; tradução Anatólio Laschuk. Porto Alegre: Artmed, 2008.

VALENÇA, M. **Aplicando Redes Neurais: Um Guia Completo**. Olinda, PE: Ed. do Autor, 2005.

XILINX. Silicon Devices. Disponível em:<<http://www.xilinx.com/products/devices.htm>>. Acesso em: 26 mar 2010.

APÊNDICES

APÊNDICE A: DESCRIÇÃO EM VHDL DE UM *PERCEPTRON* DE MÚLTIPLAS CAMADAS COM DOIS NEURÔNIOS NA CAMADA OCULTA E UM NA CAMADA DE SAÍDA.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
-----
-- teste da rede neural
-----
entity teste_RNA is
  generic(
    data_length   : integer := 16;
    addr_length   : integer := 5
  );
  port(
    -- entrada de dados
    clk,rst,wr0,wr1,wr2 : in std_logic;
    in_x0,in_x1   : in std_logic_vector(data_length-1 downto 0); -- entradas x0 e x1
    w0_i,w1_i,w2_i,w3_i,w4_i,w5_i : in std_logic_vector(data_length-1 downto 0); -- pesos
    bias0,bias1,bias2 : in std_logic_vector(data_length-1 downto 0); -- bias
    -- saída
    addr0,addr1,addr2 : out std_logic_vector(addr_length-1 downto 0); -- endereço do
intervalo
    a0,b0,a1,b1,a2,b2 : out std_logic_vector(data_length - 1 downto 0);
    regx0,regx1,regx2 : out std_logic_vector(data_length - 1 downto 0);
    net0,net1,net2    : out std_logic_vector(data_length - 1 downto 0);
    neuron_x0,neuron_x1 : out std_logic_vector(data_length - 1 downto 0);
    result            : out std_logic_vector(data_length - 1 downto 0)
  );
end teste_RNA;

architecture rede of teste_RNA is
  component teste_neuronio is
    generic(
      data_length   : integer := 16;
      addr_length   : integer := 5
    );
    port(
      -- entrada de dados
      clk,rst,wr_result : in std_logic;
      x0,x1,w0,w1,bias : in std_logic_vector(data_length-1 downto 0);
      -- saída
      addr            : out std_logic_vector(addr_length-1 downto 0); -- endereço do intervalo
      a,b,regx,x,y    : out std_logic_vector(data_length - 1 downto 0)
    );
  end component;

  signal y0,y1 : std_logic_vector(data_length - 1 downto 0);

begin

-- conexoes
neuron0 : teste_neuronio
generic map(data_length)
port map(

```



```

    clk => clk,
    rst => rst,
    wr_result => wr0,
    x0 => in_x0, -- entrada x0
    x1 => in_x1, -- entrada x1
    w0 => w0_i, -- w1
    w1 => w2_i, -- w3
    bias => bias0,
    addr => addr0,
    a => a0,
    b => b0,
    regx => regx0,
    x => net0,
    y => y0
);

```

```

neuron1 : teste_neuronio
generic map(data_length)
port map(
    clk => clk,
    rst => rst,
    wr_result => wr1,
    x0 => in_x0,
    x1 => in_x1,
    w0 => w1_i, -- w2
    w1 => w3_i, -- w4
    bias => bias1,
    addr => addr1,
    a => a1,
    b => b1,
    regx => regx1,
    x => net1,
    y => y1
);

```

```

neuron2 : teste_neuronio
generic map(data_length)
port map(
    clk => clk,
    rst => rst,
    wr_result => wr2,
    x0 => y0, -- neuronio0
    x1 => y1, -- neuronio1
    w0 => w4_i, -- w5
    w1 => w5_i, -- w6
    bias => bias2,
    addr => addr2,
    a => a2,
    b => b2,
    regx => regx2,
    x => net2,
    y => result
);

```

```

neuron_x0 <= y0;
neuron_x1 <= y1;

```

```

end rede;

```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
-----
-- net
-----
entity teste_neuronio is
  generic(
    data_length : integer := 16;
    addr_length  : integer := 5
  );
  port(
    -- entrada de dados
    clk,rst,wr_result : in std_logic;
    x0,x1,w0,w1,bias : in std_logic_vector(data_length-1 downto 0);
    -- saída
    addr : out std_logic_vector(addr_length-1 downto 0); -- endereço do intervalo
    a,b,regx,x,y : out std_logic_vector(data_length - 1 downto 0)
  );
end teste_neuronio;

architecture net of teste_neuronio is

-- net
component neuronio is
  generic(
    N : integer := 16
  );
  port(
    -- entrada de dados
    clk,clr : in std_logic;
    x0,x1,w0,w1,bias : in std_logic_vector(N-1 downto 0);
    -- saída
    z : out std_logic_vector(N - 1 downto 0)
  );
end component;

-- fnet
component fnet_test is
  generic (
    data_length : integer := 16;
    addr_length : integer := 5
  );
  port (
    clk : std_logic;
    inputx : in std_logic_vector(data_length-1 downto 0);
    address : out std_logic_vector(addr_length-1 downto 0); -- teste
    a,b,regx : out std_logic_vector(data_length-1 downto 0); -- teste
    yfnet : out signed(data_length-1 downto 0)
  );
end component;

component reg is
  generic(
    num_bit : integer := 16
  );

```

```

    port
    (
        clk : in std_logic;
        enb : in std_logic := '1';
        clr : in std_logic := '0';
        d   : in std_logic_vector(num_bit-1 downto 0);
        q   : out      std_logic_vector(num_bit-1 downto 0)
    );
end component;

signal z,yfnet_delay : std_logic_vector(data_length - 1 downto 0);
signal yfnet : signed(data_length - 1 downto 0);

begin

-- conexoes
net0 : neuronio
generic map(data_length)
port map(
    clk => clk,
    clr => rst,
    x0 => x0, -- entrada x0
    x1 => x1, -- entrada x1
    w0 => w0, -- peso w0
    w1 => w1, -- peso w1
    bias => bias, -- bias
    z => z
);

fnet0 : fnet_test
generic map(data_length,addr_length)
port map(
    clk => clk,
    inputx => z,
    address => addr, -- endereco de intervalo
    a => a,
    b => b,
    regx => regx,
    yfnet => yfnet
);

delay0 : reg
generic map(data_length)
port map(clk,wr_result,rst,std_logic_vector(yfnet),yfnet_delay);

x <= z; -- saida net
y <= yfnet_delay;

end net;

```

```

library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all;

-----
-- net
-----

entity neuronio is

generic(
  N      : integer := 16
);
port(clk,clr      : in      std_logic;
     -- entrada de dados

     x0,x1,w0,w1,bias : in  std_logic_vector(N-1 downto 0);
     -- saída
     z              : out std_logic_vector(N - 1 downto 0)

);

end neuronio;

architecture net of neuronio is

-- registrador
component reg is
  generic(
    num_bit : integer := 16
  );
  port
  (
    clk : in std_logic;
    enb : in std_logic := '1';
    clr : in std_logic := '0';
    d   : in std_logic_vector(num_bit-1 downto 0);
    q   : out      std_logic_vector(num_bit-1 downto 0)
  );
end component;

-- multiplicador generico
component mult_gen is
  generic
  (
    N      : integer := 16
  );
  port
  (
    x1_i : in std_logic_vector(N-1 downto 0);
    x2_i : in std_logic_vector(N-1 downto 0);
    y_o   : out signed(N-1 downto 0)
  )
end component;

```

```

    );
end component;

-- somador de tres entradas
component somador_gen is
    generic
    (
        N      : integer := 16
    );
    port
    (
        x0_i : in std_logic_vector(N-1 downto 0);
        x1_i : in std_logic_vector(N-1 downto 0);
        k    : in std_logic_vector(N-1 downto 0);
        s    : out std_logic_vector(N-1 downto 0)
    );
end component;

--component acum is
--    port
--    (
--        aclr      : IN STD_LOGIC := '0';
--        clock     : IN STD_LOGIC := '0';
--        data      : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
--        result    : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
--    );
--end component;

-----
signal mult0_o,mult1_o : signed(N-1 downto 0);
signal reg_mult0,reg_mult1,s : std_logic_vector(N-1 downto 0);

begin

mult0 : mult_gen -- x0*W0
generic map(N)
port map(x0,w0,mult0_o);

mult1 : mult_gen -- x1*W1
generic map(N)
port map(x1,w1,mult1_o);

register_mult0 : reg
generic map(N)
port map(clk,'1',clr,std_logic_vector(mult0_o),reg_mult0);

register_mult1 : reg
generic map(N)
port map(clk,'1',clr,std_logic_vector(mult1_o),reg_mult1);

somador0 : somador_gen -- x0*W0 + x1*W1 + bias
generic map(N)
port map(reg_mult0,reg_mult1,bias,s);

register_sum : reg
generic map(N)
port map(clk,'1',clr,s,z);

end net;

```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity calc_net is
generic (
    data_length : integer := 16
);
port (
    dataa,datab,datax : in std_logic_vector(data_length-1 downto 0);
    y : out signed(data_length-1 downto 0)
);
end calc_net;

architecture bloco of calc_net is

begin

process(dataa,datab,datax)
    variable a,b,x,aux,r : integer range -2**(data_length-1)-1 to 2**(data_length-1);
begin
    a := to_integer(signed(dataa));
    b := to_integer(signed(datab));
    x := to_integer(signed(datax));
    aux := a*x/2**(data_length-1);
    aux := aux*173670/2**(data_length-1);
    r := aux + b;
    y <= to_signed(r,data_length);
end process;
end bloco;

```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mult_gen is
  generic
    (
      N      : integer := 16
    );
  port
    (
      x1_i : in std_logic_vector(N-1 downto 0);
      x2_i : in std_logic_vector(N-1 downto 0);
      y_o   : out signed(N-1 downto 0)
    );
end mult_gen;

architecture unica of mult_gen is

begin

process(x1_i,x2_i)

variable x1,x2,aux : integer range -2**(N-1) to 2**(N-1)-1;

begin
  x1 := to_integer(signed(x1_i));
  x2 := to_integer(signed(x2_i));
  aux := x1*x2/2**(N-1);
  aux := aux*173670/2**(N-1);
  y_o <= to_signed(aux,N);
end process;
end unica;

```

```
library ieee;

use ieee.std_logic_1164.all;

entity reg is

generic(
    num_bit : integer := 16
);

port
    (
    clk : in std_logic;

        enb : in std_logic := '1';

        clr : in std_logic := '0';

        d : in std_logic_vector(num_bit-1 downto 0);

        q : out std_logic_vector(num_bit-1 downto 0)
    );

end reg;

architecture unica of reg is
begin

process(clk, clr, enb)

begin
    if clr = '1' then q <= (others => '0');

        elsif clk = '1' and clk'event and enb = '1' then

            q <= d;

        end if;

    end process;

end unica;
```



```
library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_signed.all;

use ieee.numeric_std.all;

entity somador_gen is

    generic

    (

        N      : integer := 16

    );

    port

    (

        x0_i : in std_logic_vector(N-1 downto 0);

        x1_i : in std_logic_vector(N-1 downto 0);

        k    : in std_logic_vector(N-1 downto 0);

        s    : out std_logic_vector(N-1 downto 0)

    );

end somador_gen;

architecture unica of somador_gen is

begin

    s <= x0_i + x1_i + k;

end unica;
```

```

-- teste
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity fnet_test is
generic (
    data_length : integer := 16;
    addr_length : integer := 5
);
port (
    clk : std_logic;
    inputx : in std_logic_vector(data_length-1 downto 0);
    address : out std_logic_vector(addr_length-1 downto 0); -- teste
    a,b,regx : out std_logic_vector(data_length-1 downto 0); -- teste
    yfnet : out signed(data_length-1 downto 0)
);
end fnet_test;

architecture bloco of fnet_test is

component lut is

    generic(
        data_length : integer := 16;
        addr_length : integer := 5
    );

    port(
        clk : in std_logic;
        x : in std_logic_vector(data_length-1 downto 0);
        addr : out std_logic_vector(addr_length-1 downto 0);
        a : out std_logic_vector(data_length-1 downto 0);
        b : out std_logic_vector(data_length-1 downto 0)
    );
end component;

component reg is
    generic(
        num_bit : integer := 16
    );
    port
    (
        clk : in std_logic;
        enb : in std_logic := '1';
        clr : in std_logic := '0';
        d : in std_logic_vector(num_bit-1 downto 0);
        q : out std_logic_vector(num_bit-1 downto 0)
    );
end component;

component calc_net is
generic (
    data_length : integer := 16
);
port (
    dataa,datab,datax : in std_logic_vector(data_length-1 downto 0);

```

```

    y : out signed(data_length-1 downto 0)
);
end component;
-----
signal addr_lut : std_logic_vector(addr_length-1 downto 0);
signal adata,bdata,regx0,regx1 : std_logic_vector(data_length-1 downto 0);

begin

-- sincronismo
delay0: reg
generic map(data_length)
port map(
    clk => clk,
    enb => '1',
    clr => '0',
    d=> inputx,
    q => regx0
);

-- sincronismo
delay1: reg
generic map(data_length)
port map(
    clk => clk,
    enb => '1',
    clr => '0',
    d=> regx0,
    q => regx1
);

-- valores de a e b para cada intervalo de x
lut_test0 : lut
generic map(data_length,addr_length)
port map(
    clk => clk,
    x => inputx,
    addr => addr_lut,
    a => adata,
    b => bdata
);

calc_net0 : calc_net
generic map(data_length)
port map(
    dataa => adata,
    datab => bdata,
    datax => regx1,
    y => yfnet
);

address <= addr_lut;
a <= adata;
b <= bdata;
regx <= regx1;

end bloco;

```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lut is

    generic(
        data_length : integer := 16;
        addr_length : integer := 5
    );

    port(
        clk : in std_logic;
        x   : in std_logic_vector(data_length-1 downto 0);
        addr : out std_logic_vector(addr_length-1 downto 0);
        a   : out std_logic_vector(data_length-1 downto 0);
        b   : out std_logic_vector(data_length-1 downto 0)
    );

end lut;

architecture bloco of lut is

component data_a is
    generic(
        range_size : integer := 32;
        data_length : integer := 16;
        addr_length : integer := 5
    );
    port(
        clk      : in std_logic;
        addr_a    : in std_logic_vector(addr_length-1 downto 0);--endereço da memoria
        dataa    : out std_logic_vector(data_length-1 downto 0) --saida real e imaginaria
    );
end component;

component data_b is
    generic(
        range_size : integer := 32;
        data_length : integer := 16;
        addr_length : integer := 5
    );
    port(
        clk      : in std_logic;
        addr_b    : in std_logic_vector(addr_length-1 downto 0);--endereço da memoria
        datab    : out std_logic_vector(data_length-1 downto 0) --saida real e imaginaria
    );
end component;

component test is
generic (
    data_length : integer := 16;

```

```

        addr_length : integer := 5
    );
    port (
        clk : in std_logic;
        datax : in std_logic_vector(data_length-1 downto 0);
        addr_lut : out unsigned(addr_length-1 downto 0)
    );

    end component;
-----
    signal addr_lut : unsigned(addr_length-1 downto 0);

    begin

    -- verifica qual o intervalo x se encontra
    test0 : test
    -----generic map(data_length,addr_length)
    port map(
        clk => clk,
        datax => x, -- net
        addr_lut => addr_lut -- LUT
    );

    lut0 : data_a
    port map(
        clk => clk,
        addr_a => std_logic_vector(addr_lut),
        dataa => a
    );

    lut1 : data_b
    port map(
        clk => clk,
        addr_b => std_logic_vector(addr_lut),
        datab => b
    );

    addr <= std_logic_vector(addr_lut);

    end bloco;
-----
    library ieee;
    use ieee.std_logic_1164.all;
    use ieee.numeric_std.all;

    entity data_a is
        generic(
            range_size : integer := 32;
            data_length : integer := 16;
            addr_length : integer := 5
        );
        port(
            clk : in std_logic;
            addr_a : in std_logic_vector(addr_length-1 downto 0);--endereço da memoria

```



```

    variable data : integer;
begin
    if (clk = '1' and clk'event) then
        data := mem_s(to_integer(unsigned(addr_b)));
        datab <= std_logic_vector(to_signed(data,data_length));
    end if;
end process;
end modulo;
=====
-- teste
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity test is
generic (
    -----data_length : integer := 16;
    -----addr_length : integer := 2
        data_length : integer := 16; -- comprimento de dados
        addr_length : integer := 5  -- comprimento do endereço
);
port (
    clk : in std_logic;
    datax : in std_logic_vector(data_length-1 downto 0);
    addr_lut : out unsigned(addr_length-1 downto 0)
);
end test;

architecture bloco of test is

begin

process(clk,datax)

variable aux : integer range -2**(data_length-1) to 2**(data_length-1)-1;

begin
aux := to_integer(signed(datax));
if clk'event and clk = '1' then
    if (aux >= -32768 and aux < -18549) then
        addr_lut <= "00000"; -- addr = 0
    elsif (aux >= -18549 and aux < -13638) then
        addr_lut <= "00001"; -- addr = 1
    elsif (aux >= -13638 and aux < -10862) then
        addr_lut <= "00010"; -- addr = 2
    elsif (aux >= -10862 and aux < -8877) then
        addr_lut <= "00011"; -- addr = 3
    elsif (aux >= -8877 and aux < -7291) then
        addr_lut <= "00100"; -- addr = 4
    elsif (aux >= -7291 and aux < -5927) then
        addr_lut <= "00101"; -- addr = 5
    elsif (aux >= -5927 and aux < -4685) then
        addr_lut <= "00110"; -- addr = 6
    elsif (aux >= -4685 and aux < -3484) then

```



```

    addr_lut <= "00111"; -- addr = 7
    elsif (aux >= -3484 and aux < -2227) then
        addr_lut <= "01000"; -- addr = 8
    elsif (aux >= -2227 and aux < 0) then
        addr_lut <= "01001"; -- addr = 9          -- COMO NAO PRECISA-SE DO VALOR
"0" NA ROM DO COEFICIENTE A ACRESCENTA-SE "1" NO ENDEREÇAMENTO
        elsif (aux >= 0 and aux < 2227) then          --    elsif (aux >= 0 and aux <
2227) then
            addr_lut <= "01011"; -- addr = 11          --
            addr_lut <= "01010"; -- addr = 10
            elsif (aux >= 2227 and aux < 3484) then          --    elsif (aux >= 2227 and
aux < 3484) then
                addr_lut <= "01100"; -- addr = 12          --
                addr_lut <= "01011"; -- addr = 11
                elsif (aux >= 3484 and aux < 4685) then          --    elsif (aux >= 3484 and
aux < 4685) then
                    addr_lut <= "01101"; -- addr = 13          --
                    addr_lut <= "01100"; -- addr = 12
                    elsif (aux >= 4685 and aux < 5927) then          --    elsif (aux >= 4685 and
aux < 5927) then
                        addr_lut <= "01110"; -- addr = 14          --
                        addr_lut <= "01101"; -- addr = 13
                        elsif (aux >= 5927 and aux < 7291) then          --    elsif (aux >= 5927 and
aux < 7291) then
                            addr_lut <= "01111"; -- addr = 15          --
                            addr_lut <= "01110"; -- addr = 14
                            elsif (aux >= 7291 and aux < 8877) then          --    elsif (aux >= 7291 and
aux < 8877) then
                                addr_lut <= "10000"; -- addr = 16          --
                                addr_lut <= "01111"; -- addr = 15
                                elsif (aux >= 8877 and aux < 10862) then          --    elsif (aux >= 8877 and aux <
10862) then
                                    addr_lut <= "10001"; -- addr = 17          --
                                    addr_lut <= "10000"; -- addr = 16
                                    elsif (aux >= 10862 and aux < 13638) then          --    elsif (aux >= 10862 and aux <
13638) then
                                        addr_lut <= "10010"; -- addr = 18          --
                                        addr_lut <= "10001"; -- addr = 17
                                        elsif (aux >= 13638 and aux < 18549) then          --    elsif (aux >= 13638 and aux <
18549) then
                                            addr_lut <= "10011"; -- addr = 19          --
                                            addr_lut <= "10010"; -- addr = 18
                                            elsif (aux >= 18549 and aux < 32767) then          --    elsif (aux >= 18549 and aux <
32767) then
                                                addr_lut <= "10100"; -- addr = 20          --
addr_lut <= "10011"; -- addr = 19
            else
                addr_lut <= "XXXXXX";
            end if;
        end if;
    end process;
end bloco;

```

APENDICE A: DESCRIÇÃO EM VHDL DE UM *PERCEPTRON* DE MÚLTIPLAS CAMADAS COM TRÊS NEURÔNIOS NA CAMADA OCULTA E UM NA CAMADA DE SAÍDA.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
-----
-- teste da rede neural
-----
entity teste_RNA is
    generic(
        data_length    : integer := 16;
        addr_length    : integer := 5
    );
    port(
        -- entrada de dados
        clk,rst,wr0,wr1,wr2,wr3 : in std_logic;
        in_x0 : in std_logic_vector(data_length-1 downto 0); -- entradas x0 e x1
        w0_i,w1_i,w2_i,w3_i,w4_i,w5_i : in std_logic_vector(data_length-1 downto 0); --
        pesos
        bias0,bias1,bias2,bias3 : in std_logic_vector(data_length-1 downto 0); -- bias
        -- saída
        addr0,addr1,addr2,addr3 : out std_logic_vector(addr_length-1 downto 0); --
        endereço do intervalo
        a0,b0,a1,b1,a2,b2,a3,b3 : out std_logic_vector(data_length - 1 downto 0);

        --regx0,regx1,regx2,regx3 : out std_logic_vector(data_length - 1 downto 0);
        net0,net1,net2,net3 : out std_logic_vector(data_length - 1 downto 0);
        neuron_x0,neuron_x1,neuron_x2 : out std_logic_vector(data_length - 1 downto 0);

        result : out std_logic_vector(data_length - 1 downto 0)
    );
end teste_RNA;

```

architecture rede of teste_RNA is

component teste_neuronio is

generic(

data_length : integer := 16;

addr_length : integer := 5

);

port(

-- entrada de dados

clk,rst,wr_result : in std_logic;

x0,w0,bias : in std_logic_vector(data_length-1 downto 0);

-- saída

addr : out std_logic_vector(addr_length-1 downto 0); -- endereço do

intervalo

a,b,regx,x,y : out std_logic_vector(data_length - 1 downto 0)

);

end component;

component teste_neuronio_3_in is

generic(

data_length : integer := 16;

addr_length : integer := 5

);

port(

-- entrada de dados

clk,rst,wr_result : in std_logic;

x0,w0,x1,w1,x2,w2,bias : in std_logic_vector(data_length-1 downto 0);

-- saída

addr : out std_logic_vector(addr_length-1 downto 0); -- endereço do

intervalo

a,b,regx,x,y : out std_logic_vector(data_length - 1 downto 0)

);

end component;

signal y0,y1,y2 : std_logic_vector(data_length - 1 downto 0);

signal regx0,regx1,regx2,regx3 : std_logic_vector(data_length - 1 downto 0);

```
begin

-- conexoes
neuron0 : teste_neuronio
generic map(data_length)
port map(
    clk => clk,
    rst => rst,
    wr_result => wr0,
    x0 => in_x0, -- entrada x0
    w0 => w0_i, -- w1
    bias => bias0,
    addr => addr0,
    a => a0,
    b => b0,
    regx => regx0,
    x => net0,
    y => y0
);

neuron1 : teste_neuronio
generic map(data_length)
port map(
    clk => clk,
    rst => rst,
    wr_result => wr1,
    x0 => in_x0,
    w0 => w1_i, -- w2
    bias => bias1,
    addr => addr1,
    a => a1,
    b => b1,
    regx => regx1,
    x => net1,
    y => y1
```

```

);

neuron2 : teste_neuronio
generic map(data_length)
port map(
  clk => clk,
  rst => rst,
  wr_result => wr2,
  x0 => in_x0, -- neuronio0
  w0 => w2_i, -- w5
  bias => bias2,
  addr => addr2,
  a => a2,
  b => b2,
  regx => regx2,
  x => net2,
  y => y2
);

neuron3 : teste_neuronio_3_in
generic map(data_length)
port map(
  clk => clk,
  rst => rst,
  wr_result => wr3,
  x0 => y0, -- neuronio0
  x1 => y1, -- neuronio1
  x2 => y2, -- neuronio2
  w0 => w3_i, -- w3
  w1 => w4_i, -- w4
  w2 => w5_i, -- w5
  bias => bias3,
  addr => addr3,
  a => a3,
  b => b3,
  regx => regx3,

```

```
x => net3,  
y => result  
);  
  
neuron_x0 <= y0;  
neuron_x1 <= y1;  
neuron_x2 <= y2;  
  
end rede;
```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-----
-- net
-----

entity teste_neuronio is
    generic(
        data_length : integer := 16;
        addr_length  : integer := 5
    );
    port(
        -- entrada de dados
        clk,rst,wr_result : in std_logic;
        x0,w0,bias : in std_logic_vector(data_length-1 downto 0);
        -- saída
        addr      : out std_logic_vector(addr_length-1 downto 0); -- endereço do intervalo
        a,b,regx,x,y : out std_logic_vector(data_length - 1 downto 0)
    );
end teste_neuronio;

architecture net of teste_neuronio is

-- net
component neuronio is
    generic(
        N : integer := 16
    );
    port(
        -- entrada de dados
        clk,clr : in std_logic;
        x0,w0,bias : in std_logic_vector(N-1 downto 0);
        -- saída
        z : out std_logic_vector(N - 1 downto 0)
    );
end component;

-- fnet
component fnet_test is

```

```

generic (
    data_length : integer := 16;
    addr_length  : integer := 5
);
port (
    clk   : std_logic;
    inputx : in std_logic_vector(data_length-1 downto 0);
    address : out std_logic_vector(addr_length-1 downto 0); -- teste
    a,b,regx : out std_logic_vector(data_length-1 downto 0); -- teste
    yfnet : out signed(data_length-1 downto 0)
);
end component;

component reg is
    generic(
        num_bit : integer := 16
    );
    port
    (
        clk : in std_logic;
        enb : in std_logic := '1';
        clr : in std_logic := '0';
        d   : in std_logic_vector(num_bit-1 downto 0);
        q   : out std_logic_vector(num_bit-1 downto 0)
    );
end component;

signal z,yfnet_delay : std_logic_vector(data_length - 1 downto 0);
signal yfnet : signed(data_length - 1 downto 0);

begin

-- conexoes
net0 : neuronio
generic map(data_length)
port map(
    clk => clk,
    clr => rst,
    x0 => x0, -- entrada x0
    w0 => w0, -- peso w0

```



```
    bias => bias, -- bias
    z => z
);

fnet0 : fnet_test
generic map(data_length,addr_length)
port map(
    clk => clk,
    inputx => z,
    address => addr, -- endereco de intervalo
    a => a,
    b => b,
    regx => regx,
    yfnet => yfnet
);

delay0 : reg
generic map(data_length)
port map(clk,wr_result,rst,std_logic_vector(yfnet),yfnet_delay);

x <= z; -- saida net
y <= yfnet_delay;

end net;
```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-----

-- net

-----

entity neuronio_3_in is
    generic(
        N : integer := 16
    );

port(clk,clr : in std_logic;
     -- entrada de dados
     x0,w0,x1,w1,x2,w2,bias : in std_logic_vector(N-1 downto 0);
     -- saída
     z : out std_logic_vector(N - 1 downto 0)
    );
end neuronio_3_in;

architecture net of neuronio_3_in is

-- registrador
component reg is
    generic(
        num_bit : integer := 16
    );
    port
    (
        clk : in std_logic;
        enb : in std_logic := '1';
        clr : in std_logic := '0';
        d : in std_logic_vector(num_bit-1 downto 0);
        q : out std_logic_vector(num_bit-1 downto 0)
    );
end component;

-- multiplicador generico
component mult_gen is
    generic

```

```

(
    N      : integer := 16
);
port
(
    x1_i : in std_logic_vector(N-1 downto 0);
    x2_i : in std_logic_vector(N-1 downto 0);
    y_o   : out signed(N-1 downto 0)
);
end component;

-- somador de tres entradas
component somador_gen_3_in is
    generic
    (
        N      : integer := 16
    );
    port
    (
        x0_i : in std_logic_vector(N-1 downto 0);
        x1_i : in std_logic_vector(N-1 downto 0);
        x2_i : in std_logic_vector(N-1 downto 0);
        k    : in std_logic_vector(N-1 downto 0);
        s     : out std_logic_vector(N-1 downto 0)
    );
end component;

--component acum is
--    port
--    (
--        aclr      : IN STD_LOGIC := '0';
--        clock     : IN STD_LOGIC := '0';
--        data      : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
--        result    : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
--    );
--end component;

-----
signal mult0_o,mult1_o,mult2_o : signed(N-1 downto 0);
signal reg_mult0,reg_mult1,reg_mult2,s : std_logic_vector(N-1 downto 0);

```

```
begin

mult0 : mult_gen -- x0*W0
generic map(N)
port map(x0,w0,mult0_o);

mult1 : mult_gen -- x0*W0
generic map(N)
port map(x1,w1,mult1_o);

mult2 : mult_gen -- x0*W0
generic map(N)
port map(x2,w2,mult2_o);

register_mult0 : reg
generic map(N)
port map(clk,'1',clr,std_logic_vector(mult0_o),reg_mult0);

register_mult1 : reg
generic map(N)
port map(clk,'1',clr,std_logic_vector(mult1_o),reg_mult1);

register_mult2 : reg
generic map(N)
port map(clk,'1',clr,std_logic_vector(mult2_o),reg_mult2);

somador0 : somador_gen_3_in -- x0*W0 + x1*W1 + bias
generic map(N)
port map(reg_mult0,reg_mult1,reg_mult2,bias,s);

register_sum : reg
generic map(N)
port map(clk,'1',clr,s,z);

end net;
```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mult_gen is
    generic
    (
        N      : integer := 16
    );
    port
    (
        x1_i : in std_logic_vector(N-1 downto 0);
        x2_i : in std_logic_vector(N-1 downto 0);
        y_o   : out signed(N-1 downto 0)
    );
end mult_gen;

architecture unica of mult_gen is

begin

process(x1_i,x2_i)

variable x1,x2,aux : integer range -2**(N-1) to 2**(N-1)-1;

begin
    x1 := to_integer(signed(x1_i));
    x2 := to_integer(signed(x2_i));
    aux := x1*x2/2**(N-1);
    aux := aux*173670/2**(N-1);
    y_o <= to_signed(aux,N);

end process;
end unica;

```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;

entity somador_gen_3_in is
    generic
    (
        N      : integer := 16
    );
    port
    (
        x0_i : in std_logic_vector(N-1 downto 0);
        x1_i : in std_logic_vector(N-1 downto 0);
        x2_i : in std_logic_vector(N-1 downto 0);
        k    : in std_logic_vector(N-1 downto 0);
        s    : out std_logic_vector(N-1 downto 0)
    );
end somador_gen_3_in;

architecture unica of somador_gen_3_in is

begin

    s <= x0_i + x1_i + x2_i + k;

end unica;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity reg is

generic(
num_bit : integer := 16
);
port
(
clk : in std_logic;

enb : in std_logic := '1';

clr : in std_logic := '0';

d : in std_logic_vector(num_bit-1 downto 0);

q : out std_logic_vector(num_bit-1 downto 0)
);

end reg;
architecture unica of reg is
begin

process(clk, clr, enb)
begin
if clr = '1' then
q <= (others => '0');

elsif clk = '1' and clk'event and enb = '1' then

q <= d;

end if;
end process;
end unica;
```

```

-- teste
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity fnet_test is
generic (
    data_length : integer := 16;
    addr_length : integer := 5
);
port (
    clk : std_logic;
    inputx : in std_logic_vector(data_length-1 downto 0);
    address : out std_logic_vector(addr_length-1 downto 0); -- teste
    a,b,regx : out std_logic_vector(data_length-1 downto 0); -- teste
    yfnet : out signed(data_length-1 downto 0)
);
end fnet_test;

architecture bloco of fnet_test is

component lut is

generic(
    data_length : integer := 16;
    addr_length : integer := 5
);

port(
    clk : in std_logic;
    x : in std_logic_vector(data_length-1 downto 0);
    addr : out std_logic_vector(addr_length-1 downto 0);
    a : out std_logic_vector(data_length-1 downto 0);
    b : out std_logic_vector(data_length-1 downto 0)
);

end component;

component reg is
generic(

```



```

    num_bit : integer := 16
);
    port
    (
        clk : in std_logic;
        enb : in std_logic := '1';
        clr : in std_logic := '0';
        d   : in std_logic_vector(num_bit-1 downto 0);
        q   : out      std_logic_vector(num_bit-1 downto 0)
    );
end component;

component calc_net is
generic (
    data_length : integer := 16
);
port (
    dataa,datab,datax : in std_logic_vector(data_length-1 downto 0);
    y : out signed(data_length-1 downto 0)
);
end component;

-----
signal addr_lut : std_logic_vector(addr_length-1 downto 0);
signal adata,bdata,regx0,regx1 : std_logic_vector(data_length-1 downto 0);

begin

-- sincronismo
delay0: reg
generic map(data_length)
port map(
    clk => clk,
    enb => '1',
    clr => '0',
    d=> inputx,
    q => regx0
);

-- sincronismo

```

```
delay1: reg
generic map(data_length)
port map(
  clk => clk,
  enb => '1',
  clr => '0',
  d => regx0,
  q => regx1
);

-- valores de a e b para cada intervalo de x
lut_test0 : lut
generic map(data_length,addr_length)
port map(
  clk => clk,
  x => inputx,
  addr => addr_lut,
  a => adata,
  b => bdata
);

calc_net0 : calc_net
generic map(data_length)
port map(
  dataa => adata,
  datab => bdata,
  datax => regx1,
  y => yfnet
);

address <= addr_lut;
a <= adata;
b <= bdata;
regx <= regx1;

end bloco;
```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lut is

    generic(
        data_length : integer := 16;
        addr_length : integer := 5
    );

    port(
        clk : in std_logic;
        x   : in std_logic_vector(data_length-1 downto 0);
        addr : out std_logic_vector(addr_length-1 downto 0);
        a   : out std_logic_vector(data_length-1 downto 0);
        b   : out std_logic_vector(data_length-1 downto 0)
    );
end lut;

architecture bloco of lut is

    component data_a is
        generic(
            range_size : integer := 32;
            data_length : integer := 16;
            addr_length : integer := 5
        );
        port(
            clk      : in std_logic;
            addr_a   : in std_logic_vector(addr_length-1 downto 0);--endereço da memoria
            dataa   : out std_logic_vector(data_length-1 downto 0) --saida real e imaginaria
        );
    end component;

    component data_b is
        generic(
            range_size : integer := 32;
            data_length : integer := 16;

```

```

    addr_length : integer := 5
);
port(
    clk      : in std_logic;
    addr_b   : in std_logic_vector(addr_length-1 downto 0);--endereço da memoria
    datab   : out std_logic_vector(data_length-1 downto 0) --saida real e imaginaria
);
end component;

component test is
generic (
    data_length : integer := 16;
    addr_length : integer := 5
);
port (
    clk : in std_logic;
    datax : in std_logic_vector(data_length-1 downto 0);
    addr_lut : out unsigned(addr_length-1 downto 0)
);

end component;

-----
signal addr_lut : unsigned(addr_length-1 downto 0);

begin

-- verifica qual o intervalo x se encontra
test0 : test
-----generic map(data_length,addr_length)
port map(
    clk => clk,
    datax => x, -- net
    addr_lut => addr_lut -- LUT
);

lut0 : data_a
port map(
    clk => clk,
    addr_a => std_logic_vector(addr_lut),

```

```

        dataa => a
    );

    lut1 : data_b
port map(
    clk => clk,
    addr_b => std_logic_vector(addr_lut),
    datab => b
);

addr <= std_logic_vector(addr_lut);

end bloco;
-----
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity data_a is
    generic(
        range_size : integer := 32;
        data_length : integer := 16;
        addr_length : integer := 5
    );
    port(
        clk      : in std_logic;
        addr_a   : in std_logic_vector(addr_length-1 downto 0);--endereço da memoria
        dataa    : out std_logic_vector(data_length-1 downto 0) --saida real e imaginaria
    );
end data_a;

architecture modulo of data_a is

type MEM is array (0 to range_size-1) of integer range -2**(data_length-1) to 2**(data_length-1)-1;
    constant mem_s : mem :=
    ( 9,
      100,
      313,
      638,
      1063,

```



```

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity data_b is
  generic(
    range_size : integer := 32; -- comprimento da rom
    data_length : integer := 16; -- comprimento de dados
    addr_length : integer := 5 -- comprimento do endereço
  );
  port(
    clk      : in std_logic;
    addr_b   : in std_logic_vector(addr_length-1 downto 0);--endereço da memoria
    datab   : out std_logic_vector(data_length-1 downto 0) --saida real e imaginaria
  );
end data_b;

architecture modulo of data_b is

  type MEM is array (0 to range_size-1) of integer range -2**(data_length-1) to 2**(data_length-1)-1;
  constant mem_s : mem :=
    (-4086
    -3709
    -3079
    -2331
    -1553
    -818
    -183
    295
    563
    590
    0
    -590
    -563
    -295
    183
    818
    1553
    2331
    3079
    3709
    4086
    0,
    0,

```

```

0,
0,
0,
0,
0,
0,
0,
0,
0
);
begin
  process(clk)
    variable data : integer;
  begin
    if (clk = '1' and clk'event) then
      data := mem_s(to_integer(unsigned(addr_b)));
      datab <= std_logic_vector(to_signed(data,data_length));
    end if;
  end process;
end modulo;

=====

-- teste
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity test is
generic (
  -----data_length : integer := 16;
  -----addr_length : integer := 2
  data_length : integer := 16; -- comprimento de dados
  addr_length : integer := 5 -- comprimento do endereço
);
port (
  clk : in std_logic;
  datab : in std_logic_vector(data_length-1 downto 0);
  addr_lut : out unsigned(addr_length-1 downto 0)
);

end test;

```



```

architecture bloco of test is

begin
process(clk,datax)

variable aux : integer range -2**(data_length-1) to 2**(data_length-1)-1;

begin
aux := to_integer(signed(datax));
if clk'event and clk = '1' then
    if (aux >= -32768 and aux < -18549) then
        addr_lut <= "00000"; -- addr = 0
    elsif (aux >= -18549 and aux < -13638) then
        addr_lut <= "00001"; -- addr = 1
    elsif (aux >= -13638 and aux < -10862) then
        addr_lut <= "00010"; -- addr = 2
    elsif (aux >= -10862 and aux < -8877) then
        addr_lut <= "00011"; -- addr = 3
    elsif (aux >= -8877 and aux < -7291) then
        addr_lut <= "00100"; -- addr = 4
    elsif (aux >= -7291 and aux < -5927) then
        addr_lut <= "00101"; -- addr = 5
    elsif (aux >= -5927 and aux < -4685) then
        addr_lut <= "00110"; -- addr = 6
    elsif (aux >= -4685 and aux < -3484) then
        addr_lut <= "00111"; -- addr = 7
    elsif (aux >= -3484 and aux < -2227) then
        addr_lut <= "01000"; -- addr = 8
    elsif (aux >= -2227 and aux < 0) then
        addr_lut <= "01001"; -- addr = 9          -- COMO NAO PRECISA-SE DO VALOR "0" NA ROM DO
COEFICIENTE A ACRESCENTA-SE "1" NO ENDEREÇAMENTO
    elsif (aux >= 0 and aux < 2227) then          --      elsif (aux >= 0 and aux < 2227)
then
        addr_lut <= "01011"; -- addr = 11          --
    addr_lut <= "01010"; -- addr = 10
    elsif (aux >= 2227 and aux < 3484) then          --      elsif (aux >= 2227 and aux < 3484)
then
        addr_lut <= "01100"; -- addr = 12          --
    addr_lut <= "01011"; -- addr = 11
    elsif (aux >= 3484 and aux < 4685) then          --      elsif (aux >= 3484 and aux < 4685)

```

```

then
    addr_lut <= "01101"; -- addr = 13
addr_lut <= "01100"; -- addr = 12
    elsif (aux >= 4685 and aux < 5927) then -- elsif (aux >= 4685 and aux < 5927)
then
    addr_lut <= "01110"; -- addr = 14
addr_lut <= "01101"; -- addr = 13
    elsif (aux >= 5927 and aux < 7291) then -- elsif (aux >= 5927 and aux < 7291)
then
    addr_lut <= "01111"; -- addr = 15
addr_lut <= "01110"; -- addr = 14
    elsif (aux >= 7291 and aux < 8877) then -- elsif (aux >= 7291 and aux < 8877)
then
    addr_lut <= "10000"; -- addr = 16
addr_lut <= "01111"; -- addr = 15
    elsif (aux >= 8877 and aux < 10862) then -- elsif (aux >= 8877 and aux < 10862) then
        addr_lut <= "10001"; -- addr = 17
addr_lut <= "10000"; -- addr = 16
    elsif (aux >= 10862 and aux < 13638) then -- elsif (aux >= 10862 and aux < 13638) then
        addr_lut <= "10010"; -- addr = 18
addr_lut <= "10001"; -- addr = 17
    elsif (aux >= 13638 and aux < 18549) then -- elsif (aux >= 13638 and aux < 18549) then
        addr_lut <= "10011"; -- addr = 19
addr_lut <= "10010"; -- addr = 18
    elsif (aux >= 18549 and aux < 32767) then -- elsif (aux >= 18549 and aux < 32767) then
        addr_lut <= "10100"; -- addr = 20
        addr_lut
<= "10011"; -- addr = 19
    else
        addr_lut <= "XXXXXX";
    end if;
end if;
end process;
end bloco;
--
=====
=

```

```

-- teste
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity calc_net is
generic (
    data_length : integer := 16
);
port (
    dataa,datab,datax : in std_logic_vector(data_length-1 downto 0);
    y : out signed(data_length-1 downto 0)
);

end calc_net;

architecture bloco of calc_net is

begin

process(dataa,datab,datax)
    variable a,b,x,aux,r : integer range -2**(data_length-1)-1 to 2**(data_length-1);

begin
    a := to_integer(signed(dataa));
    b := to_integer(signed(datab));
    x := to_integer(signed(datax));
    aux := a*x/2**(data_length-1);
    aux := aux*173670/2**(data_length-1);
    r := aux + b;
    y <= to_signed(r,data_length);

end process;
end bloco;

```

APENDICE A: DESCRIÇÃO EM VHDL DE UM *PERCEPTRON* DE MÚLTIPLAS CAMADAS COM CINCO NEURÔNIOS NA CAMADA OCULTA E UM NA CAMADA DE SAÍDA.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
-----
-- teste da rede neural
-----
entity teste_RNA is
    generic(
        data_length    : integer := 16;
        addr_length    : integer := 5
    );
    port(
        -- entrada de dados
        clk,rst      : in std_logic;
        in_x0        : in std_logic_vector(data_length-1 downto 0); -- entradas x0 e x1
        w0_i,w1_i,w2_i,w3_i,w4_i,w5_i,w6_i,w7_i,w8_i,w9_i : in std_logic_vector(data_length-1
downto 0); -- pesos
        bias0,bias1,bias2,bias3,bias4,bias5  : in std_logic_vector(data_length-1 downto 0); -- bias
        -- saída
        --addr0,addr1,addr2,addr3  : out std_logic_vector(addr_length-1 downto 0); -- endereço do
intervalo
        --a0,b0,a1,b1,a2,b2,a3,b3  : out std_logic_vector(data_length - 1 downto 0);
        --regx0,regx1,regx2,regx3  : out std_logic_vector(data_length - 1 downto 0);
        --net0,net1,net2,net3      : out std_logic_vector(data_length - 1 downto 0);
        --neuron_x0,neuron_x1,neuron_x2 : out std_logic_vector(data_length - 1 downto 0);
        result          : out std_logic_vector(data_length - 1 downto 0)
    );
end teste_RNA;

architecture rede of teste_RNA is

component teste_neuronio is
    generic(
        data_length    : integer := 16;
        addr_length    : integer := 5
    );

```

```

port(
  -- entrada de dados
  clk,rst,wr_result : in std_logic;
  x0,w0,bias : in std_logic_vector(data_length-1 downto 0);
  -- saída
  addr      : out std_logic_vector(addr_length-1 downto 0); -- endereço do intervalo
  a,b,regx,x,y : out std_logic_vector(data_length - 1 downto 0)
);
end component;

component teste_neuronio_5_in is
  generic(
    data_length : integer := 16;
    addr_length  : integer := 5
  );
  port(
    -- entrada de dados
    clk,rst,wr_result : in std_logic;
    x0,x1,x2,x3,x4,w0,w1,w2,w3,w4,bias : in std_logic_vector(data_length-1 downto 0);
    -- saída
    addr      : out std_logic_vector(addr_length-1 downto 0); -- endereço do intervalo
    a,b,regx,x,y : out std_logic_vector(data_length - 1 downto 0)
  );
end component;

signal wr0,wr1,wr2,wr3,wr4,wr5 : std_logic := '1';
signal y0,y1,y2,y3,y4 : std_logic_vector(data_length - 1 downto 0);
signal regx0,regx1,regx2,regx3,regx4,regx5 : std_logic_vector(data_length - 1 downto 0);
signal net0,net1,net2,net3,net4,net5 : std_logic_vector(data_length - 1 downto 0);
signal neuron_x0,neuron_x1,neuron_x2,neuron_x3,neuron_x4 : std_logic_vector(data_length - 1 downto 0);

signal addr0,addr1,addr2,addr3,addr4,addr5 : std_logic_vector(addr_length-1 downto 0);
signal a0,b0,a1,b1,a2,b2,a3,b3,a4,b4,a5,b5 : std_logic_vector(data_length - 1 downto 0);

begin

-- conexoes
neuron0 : teste_neuronio
generic map(data_length)

```

```

port map(
    clk => clk,
    rst => rst,
    wr_result => wr0,
    x0 => in_x0, -- entrada x0
    w0 => w0_i, -- w1
    bias => bias0,
    addr => addr0,
    a => a0,
    b => b0,
    regx => regx0,
    x => net0,
    y => y0
);

```

neuron1 : teste_neuronio

generic map(data_length)

```

port map(
    clk => clk,
    rst => rst,
    wr_result => wr1,
    x0 => in_x0,
    w0 => w1_i, -- w2
    bias => bias1,
    addr => addr1,
    a => a1,
    b => b1,
    regx => regx1,
    x => net1,
    y => y1
);

```

neuron2 : teste_neuronio

generic map(data_length)

```

port map(
    clk => clk,
    rst => rst,
    wr_result => wr2,
    x0 => in_x0, -- neuronio0
    w0 => w2_i, -- w5

```

```

    bias => bias2,
    addr => addr2,
    a => a2,
    b => b2,
    regx => regx2,
    x => net2,
    y => y2
);

neuron3 : teste_neuronio
generic map(data_length)
port map(
    clk => clk,
    rst => rst,
    wr_result => wr3,
    x0 => in_x0, -- neuronio0
    w0 => w3_i, -- w5
    bias => bias3,
    addr => addr3,
    a => a3,
    b => b3,
    regx => regx3,
    x => net3,
    y => y3
);

neuron4 : teste_neuronio
generic map(data_length)
port map(
    clk => clk,
    rst => rst,
    wr_result => wr4,
    x0 => in_x0, -- neuronio0
    w0 => w4_i, -- w5
    bias => bias4,
    addr => addr4,
    a => a4,
    b => b4,
    regx => regx4,
    x => net4,

```

```
    y => y4
);

neuron5 : teste_neuronio_5_in
generic map(data_length)
port map(
    clk => clk,
    rst => rst,
    wr_result => wr5,
    x0 => y0,
    x1 => y1,
    x2 => y2,
    x3 => y3,
    x4 => y4,
    w0 => w0_i,
    w1 => w1_i,
    w2 => w2_i,
    w3 => w3_i,
    w4 => w4_i,
    bias => bias5,
    addr => addr5,
    a => a5,
    b => b5,
    regx => regx5,
    x => net5,
    y => result
);
--neuron_x0 <= y0;
--neuron_x1 <= y1;
--neuron_x2 <= y2;

end rede;
```



```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-----

-- net

-----

entity neuronio_5_in is
    generic(
        N : integer := 16
    );

    port(
        -- entrada de dados
        clk,clr : in std_logic;

        x0,x1,x2,x3,x4,w0,w1,w2,w3,w4,bias : in std_logic_vector(N-1 downto 0);

        -- saída
        z : out std_logic_vector(N - 1 downto 0)
    );

end neuronio_5_in;

architecture net of neuronio_5_in is

-- registrador
component reg is
    generic(
        num_bit : integer := 16
    );
    port
    (
        clk : in std_logic;
        enb : in std_logic := '1';
        clr : in std_logic := '0';
        d : in std_logic_vector(num_bit-1 downto 0);
        q : out std_logic_vector(num_bit-1 downto 0)
    );
end component;

```

```

-- multiplicador generico
component mult_gen is
  generic
  (
    N      : integer := 16
  );
  port
  (
    x1_i : in std_logic_vector(N-1 downto 0);
    x2_i : in std_logic_vector(N-1 downto 0);
    y_o   : out signed(N-1 downto 0)
  );
end component;

-- somador de tres entradas
component somador_gen_5_in is
  generic
  (
    N      : integer := 16
  );
  port
  (
    x0_i : in std_logic_vector(N-1 downto 0);
    x1_i : in std_logic_vector(N-1 downto 0);
    x2_i : in std_logic_vector(N-1 downto 0);
    x3_i : in std_logic_vector(N-1 downto 0);
    x4_i : in std_logic_vector(N-1 downto 0);
    k    : in std_logic_vector(N-1 downto 0);
    s     : out std_logic_vector(N-1 downto 0)
  );
end component;

--component acum is
--  port
--  (
--    aclr      : IN STD_LOGIC := '0';
--    clock     : IN STD_LOGIC := '0';
--    data      : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
--    result    : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)

```

```

--      );
--end component;

-----

signal mult0_o,mult1_o,mult2_o,mult3_o,mult4_o : signed(N-1 downto 0);
signal reg_mult0,reg_mult1,reg_mult2,reg_mult3,reg_mult4,s : std_logic_vector(N-1 downto 0);

begin

mult0 : mult_gen -- x0*W0
generic map(N)
port map(x0,w0,mult0_o);

mult1 : mult_gen -- x0*W0
generic map(N)
port map(x1,w1,mult1_o);

mult2 : mult_gen -- x0*W0
generic map(N)
port map(x2,w2,mult2_o);

mult3 : mult_gen -- x0*W0
generic map(N)
port map(x3,w3,mult3_o);

mult4 : mult_gen -- x0*W0
generic map(N)
port map(x4,w4,mult4_o);

register_mult0 : reg
generic map(N)
port map(clk,'1',clr,std_logic_vector(mult0_o),reg_mult0);

register_mult1 : reg
generic map(N)
port map(clk,'1',clr,std_logic_vector(mult1_o),reg_mult1);

register_mult2 : reg
generic map(N)
port map(clk,'1',clr,std_logic_vector(mult2_o),reg_mult2);

```

```
register_mult3 : reg
generic map(N)
port map(clk,'1',clr,std_logic_vector(mult3_o),reg_mult3);

register_mult4 : reg
generic map(N)
port map(clk,'1',clr,std_logic_vector(mult4_o),reg_mult4);

somador0 : somador_gen_5_in --  $x0*W0 + x1*W1 .. + bias$ 
generic map(N)
port map(reg_mult0,reg_mult1,reg_mult2,reg_mult3,reg_mult4,bias,s);

register_sum : reg
generic map(N)
port map(clk,'1',clr,s,z);

end net;
```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-----
-- net
-----

entity teste_neuronio_5_in is
    generic(
        data_length : integer := 16;
        addr_length  : integer := 5
    );
    port(
        -- entrada de dados
        clk,rst,wr_result : in std_logic;
        x0,x1,x2,x3,x4,w0,w1,w2,w3,w4,bias : in std_logic_vector(data_length-1 downto 0);
        -- saída
        addr          : out std_logic_vector(addr_length-1 downto 0); -- endereço do intervalo
        a,b,regx,x,y  : out std_logic_vector(data_length - 1 downto 0)
    );
end teste_neuronio_5_in;

architecture net of teste_neuronio_5_in is

-- net
component neuronio_5_in is
    generic(
        N : integer := 16
    );
    port(
        -- entrada de dados
        clk,clr      : in std_logic;
        x0,x1,x2,x3,x4,w0,w1,w2,w3,w4,bias : in std_logic_vector(N-1 downto 0);
        -- saída
        z            : out std_logic_vector(N - 1 downto 0)
    );
end component;

-- fnet
component fnet_test is

```

```

generic (
    data_length : integer := 16;
    addr_length : integer := 5
);
port (
    clk : std_logic;
    inputx : in std_logic_vector(data_length-1 downto 0);
    address : out std_logic_vector(addr_length-1 downto 0); -- teste
    a,b,regx : out std_logic_vector(data_length-1 downto 0); -- teste
    yfnet : out signed(data_length-1 downto 0)
);
end component;

component reg is
    generic(
        num_bit : integer := 16
    );
    port
    (
        clk : in std_logic;
        enb : in std_logic := '1';
        clr : in std_logic := '0';
        d : in std_logic_vector(num_bit-1 downto 0);
        q : out std_logic_vector(num_bit-1 downto 0)
    );
end component;

signal z,yfnet_delay : std_logic_vector(data_length - 1 downto 0);
signal yfnet : signed(data_length - 1 downto 0);

begin

-- conexoes
net_5_in : neuronio_5_in
generic map(data_length)
port map(
    clk => clk,
    clr => rst,
    x0 => x0,
    x1 => x1,

```

```

    x2 => x2,
    x3 => x3,
    x4 => x4,
    w0 => w0,
    w1 => w1,
    w2 => w2,
    w3 => w3,
    w4 => w4,
    bias => bias,
    z => z
);

fnet0 : fnet_test
generic map(data_length,addr_length)
port map(
    clk => clk,
    inputx => z,
    address => addr, -- endereco de intervalo
    a => a,
    b => b,
    regx => regx,
    yfnet => yfnet
);

delay0 : reg
generic map(data_length)
port map(clk,wr_result,rst,std_logic_vector(yfnet),yfnet_delay);

x <= z; -- saida net
y <= yfnet_delay;

end net;

```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mult_gen is
    generic
    (
        N      : integer := 16
    );
    port
    (
        x1_i : in std_logic_vector(N-1 downto 0);
        x2_i : in std_logic_vector(N-1 downto 0);
        y_o   : out signed(N-1 downto 0)
    );
end mult_gen;

architecture unica of mult_gen is

begin

process(x1_i,x2_i)

variable x1,x2,aux : integer range -2**(N-1) to 2**(N-1)-1;

begin
    x1 := to_integer(signed(x1_i));
    x2 := to_integer(signed(x2_i));
    aux := x1*x2/2**(N-1);
    aux := aux* 364950 /2**(N-1);
    y_o <= to_signed(aux,N);

end process;
end unica;

```



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;

entity somador_gen_5_in is
    generic
    (
        N      : integer := 16
    );
    port
    (
        x0_i : in std_logic_vector(N-1 downto 0);
        x1_i : in std_logic_vector(N-1 downto 0);
        x2_i : in std_logic_vector(N-1 downto 0);
        x3_i : in std_logic_vector(N-1 downto 0);
        x4_i : in std_logic_vector(N-1 downto 0);
        k    : in std_logic_vector(N-1 downto 0);
        s    : out std_logic_vector(N-1 downto 0)
    );
end somador_gen_5_in;

architecture unica of somador_gen_5_in is

begin

    s <= x0_i+x1_i+x2_i+x3_i+x4_i+k;

end unica;
```

```
library ieee;
use ieee.std_logic_1164.all;

entity reg is
  generic(
    num_bit : integer := 16
  );
  port
  (
    clk : in std_logic;
    enb : in std_logic := '1';
    clr : in std_logic := '0';
    d   : in std_logic_vector(num_bit-1 downto 0);
    q   : out      std_logic_vector(num_bit-1 downto 0)
  );
end reg;

architecture unica of reg is
begin

process(clk, clr, enb)
  begin
    if clr = '1' then
      q <= (others => '0');
    elsif clk = '1' and clk'event and enb = '1' then
      q <= d;
    end if;
  end process;

end unica;
```

```

-- teste
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity fnet_test is
generic (
    data_length : integer := 16;
    addr_length : integer := 5
);
port (
    clk : std_logic;
    inputx : in std_logic_vector(data_length-1 downto 0);
    address : out std_logic_vector(addr_length-1 downto 0); -- teste
    a,b,regx : out std_logic_vector(data_length-1 downto 0); -- teste
    yfnet : out signed(data_length-1 downto 0)
);
end fnet_test;

architecture bloco of fnet_test is

component lut is

generic(
    data_length : integer := 16;
    addr_length : integer := 5
);

port(
    clk : in std_logic;
    x : in std_logic_vector(data_length-1 downto 0);
    addr : out std_logic_vector(addr_length-1 downto 0);
    a : out std_logic_vector(data_length-1 downto 0);
    b : out std_logic_vector(data_length-1 downto 0)
);

end component;

component reg is
generic(

```

```

    num_bit : integer := 16
);
    port
    (
        clk : in std_logic;
        enb : in std_logic := '1';
        clr : in std_logic := '0';
        d   : in std_logic_vector(num_bit-1 downto 0);
        q   : out      std_logic_vector(num_bit-1 downto 0)
    );
end component;

component calc_net is
generic (
    data_length : integer := 16
);
port (
    dataa,datab,datax : in std_logic_vector(data_length-1 downto 0);
    y : out signed(data_length-1 downto 0)
);
end component;

-----
signal addr_lut : std_logic_vector(addr_length-1 downto 0);
signal adata,bdata,regx0,regx1 : std_logic_vector(data_length-1 downto 0);

begin

-- sincronismo
delay0: reg
generic map(data_length)
port map(
    clk => clk,
    enb => '1',
    clr => '0',
    d=> inputx,
    q => regx0
);

-- sincronismo

```

```

delay1: reg
generic map(data_length)
port map(
  clk => clk,
  enb => '1',
  clr => '0',
  d => regx0,
  q => regx1
);

-- valores de a e b para cada intervalo de x
lut_test0 : lut
generic map(data_length,addr_length)
port map(
  clk => clk,
  x => inputx,
  addr => addr_lut,
    a => adata,
    b => bdata
);

calc_net0 : calc_net
generic map(data_length)
port map(
  dataa => adata,
  datab => bdata,
  datax => regx1,
  y => yfnet
);

address <= addr_lut;
a <= adata;
b <= bdata;
regx <= regx1;

end bloco;

```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lut is

    generic(
        data_length : integer := 16;
        addr_length  : integer := 5
    );

    port(
        clk : in std_logic;
        x   : in std_logic_vector(data_length-1 downto 0);
        addr : out std_logic_vector(addr_length-1 downto 0);
        a    : out std_logic_vector(data_length-1 downto 0);
        b    : out std_logic_vector(data_length-1 downto 0)
    );
end lut;

architecture bloco of lut is

    component data_a is
        generic(
            range_size : integer := 32;
            data_length : integer := 16;
            addr_length : integer := 5
        );
        port(
            clk      : in std_logic;
            addr_a   : in std_logic_vector(addr_length-1 downto 0);--endereço da memoria
            dataa    : out std_logic_vector(data_length-1 downto 0) --saida real e imaginaria
        );
    end component;

    component data_b is
        generic(
            range_size : integer := 32;
            data_length : integer := 16;

```

```

    addr_length : integer := 5
);
port(
    clk      : in std_logic;
    addr_b   : in std_logic_vector(addr_length-1 downto 0);--endereço da memoria
    datab   : out std_logic_vector(data_length-1 downto 0) --saida real e imaginaria
);
end component;

component test is
generic (
    data_length : integer := 16;
    addr_length  : integer := 5
);
port (
    clk  : in std_logic;
    datax : in std_logic_vector(data_length-1 downto 0);
    addr_lut : out unsigned(addr_length-1 downto 0)
);
end component;

-----
signal addr_lut : unsigned(addr_length-1 downto 0);

begin

-- verifica qual o intervalo x se encontra
test0 : test
-----generic map(data_length,addr_length)
port map(
    clk => clk,
    datax => x, -- net
    addr_lut => addr_lut -- LUT
);

lut0 : data_a
port map(
    clk => clk,
    addr_a => std_logic_vector(addr_lut),

```

```

        dataa => a
    );

    lut1 : data_b
port map(
    clk => clk,
    addr_b => std_logic_vector(addr_lut),
    datab => b
);

addr <= std_logic_vector(addr_lut);

end bloco;
-----
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity data_a is
    generic(
        range_size : integer := 32;
        data_length : integer := 16;
        addr_length : integer := 5
    );
    port(
        clk      : in std_logic;
        addr_a   : in std_logic_vector(addr_length-1 downto 0);--endereço da memoria
        dataa    : out std_logic_vector(data_length-1 downto 0) --saida real e imaginaria
    );
end data_a;

architecture modulo of data_a is

type MEM is array (0 to range_size-1) of integer range -2**(data_length-1) to 2**(data_length-1)-1;
    constant mem_s : mem :=
    ( 6,
      71,
      221,
      452,
      753,

```



```

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity data_b is
  generic(
    range_size : integer := 32; -- comprimento da rom
    data_length : integer := 16; -- comprimento de dados
    addr_length : integer := 5 -- comprimento do endereço
  );
  port(
    clk      : in std_logic;
    addr_b   : in std_logic_vector(addr_length-1 downto 0);--endereço da memoria
    datab   : out std_logic_vector(data_length-1 downto 0) --saida real e imaginaria
  );
end data_b;

architecture modulo of data_b is

  type MEM is array (0 to range_size-1) of integer range -2**(data_length-1) to 2**(data_length-1)-1;
  constant mem_s : mem :=
  ( -2909,
    -2716,
    -2383,
      -1978,
      -1545,
      -1120,
      -729,
      -400,
      -155,
        0,
        0,
        0,
      155,
      400,
      729,
      1120,
      1545,
      1978,
      2383,
      2716,

```

```

        2909,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0
    );
begin
    process(clk)
        variable data : integer;
    begin
        if (clk = '1' and clk'event) then
            data := mem_s(to_integer(unsigned(addr_b)));
            datab <= std_logic_vector(to_signed(data,data_length));
        end if;
    end process;
end modulo;

=====

-- teste
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity test is
generic (
    -----data_length : integer := 16;
    -----addr_length : integer := 2
        data_length : integer := 16; -- comprimento de dados
        addr_length : integer := 5  -- comprimento do endereço
);
port (
    clk : in std_logic;
    datax : in std_logic_vector(data_length-1 downto 0);
    addr_hut : out unsigned(addr_length-1 downto 0)

```

```

);

end test;

architecture bloco of test is

begin

process(clk,datax)

variable aux : integer range -2**(data_length-1) to 2**(data_length-1)-1;

begin
aux := to_integer(signed(datax));
if clk'event and clk = '1' then
    if (aux >= -32768 and aux < -18549) then
        addr_lut <= "00000"; -- addr = 0
    elsif (aux >= -18549 and aux < -13638) then
        addr_lut <= "00001"; -- addr = 1
    elsif (aux >= -13638 and aux < -10862) then
        addr_lut <= "00010"; -- addr = 2
    elsif (aux >= -10862 and aux < -8877) then
        addr_lut <= "00011"; -- addr = 3
    elsif (aux >= -8877 and aux < -7291) then
        addr_lut <= "00100"; -- addr = 4
    elsif (aux >= -7291 and aux < -5927) then
        addr_lut <= "00101"; -- addr = 5
    elsif (aux >= -5927 and aux < -4685) then
        addr_lut <= "00110"; -- addr = 6
    elsif (aux >= -4685 and aux < -3484) then
        addr_lut <= "00111"; -- addr = 7
    elsif (aux >= -3484 and aux < -2227) then
        addr_lut <= "01000"; -- addr = 8
    elsif (aux >= -2227 and aux < 0) then
        addr_lut <= "01001"; -- addr = 9          -- COMO NAO PRECISA-SE DO VALOR "0" NA ROM
DO COEFICIENTE A ACRESCENTA-SE "1" NO ENDEREÇAMENTO
    elsif (aux >= 0 and aux < 2227) then          --          elsif (aux >= 0 and aux < 2227)
then
        addr_lut <= "01011"; -- addr = 11          --
    addr_lut <= "01010"; -- addr = 10

```

```

        elsif (aux >= 2227 and aux < 3484) then          --      elsif (aux >= 2227 and aux <
3484) then
                addr_lut <= "01100"; -- addr = 12          --
        addr_lut <= "01011"; -- addr = 11
        elsif (aux >= 3484 and aux < 4685) then          --      elsif (aux >= 3484 and aux <
4685) then
                addr_lut <= "01101"; -- addr = 13          --
        addr_lut <= "01100"; -- addr = 12
        elsif (aux >= 4685 and aux < 5927) then          --      elsif (aux >= 4685 and aux <
5927) then
                addr_lut <= "01110"; -- addr = 14          --
        addr_lut <= "01101"; -- addr = 13
        elsif (aux >= 5927 and aux < 7291) then          --      elsif (aux >= 5927 and aux <
7291) then
                addr_lut <= "01111"; -- addr = 15          --
        addr_lut <= "01110"; -- addr = 14
        elsif (aux >= 7291 and aux < 8877) then          --      elsif (aux >= 7291 and aux <
8877) then
                addr_lut <= "10000"; -- addr = 16          --
        addr_lut <= "01111"; -- addr = 15
        elsif (aux >= 8877 and aux < 10862) then --      elsif (aux >= 8877 and aux < 10862) then
                addr_lut <= "10001"; -- addr = 17          --
        addr_lut <= "10000"; -- addr = 16
        elsif (aux >= 10862 and aux < 13638) then --      elsif (aux >= 10862 and aux < 13638) then
                addr_lut <= "10010"; -- addr = 18          --
        addr_lut <= "10001"; -- addr = 17
        elsif (aux >= 13638 and aux < 18549) then --      elsif (aux >= 13638 and aux < 18549) then
                addr_lut <= "10011"; -- addr = 19          --
        addr_lut <= "10010"; -- addr = 18
        elsif (aux >= 18549 and aux < 32767) then --      elsif (aux >= 18549 and aux < 32767) then
                addr_lut <= "10100"; -- addr = 20          --      addr_lut
<= "10011"; -- addr = 19
        else
                addr_lut <= "XXXXX";
        end if;
end if;
end process;
end bloco;
--
=====

```

```

-- teste
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity calc_net is
generic (
    data_length : integer := 16
);
port (
    dataa,datab,datax : in std_logic_vector(data_length-1 downto 0);
    y : out signed(data_length-1 downto 0)
);

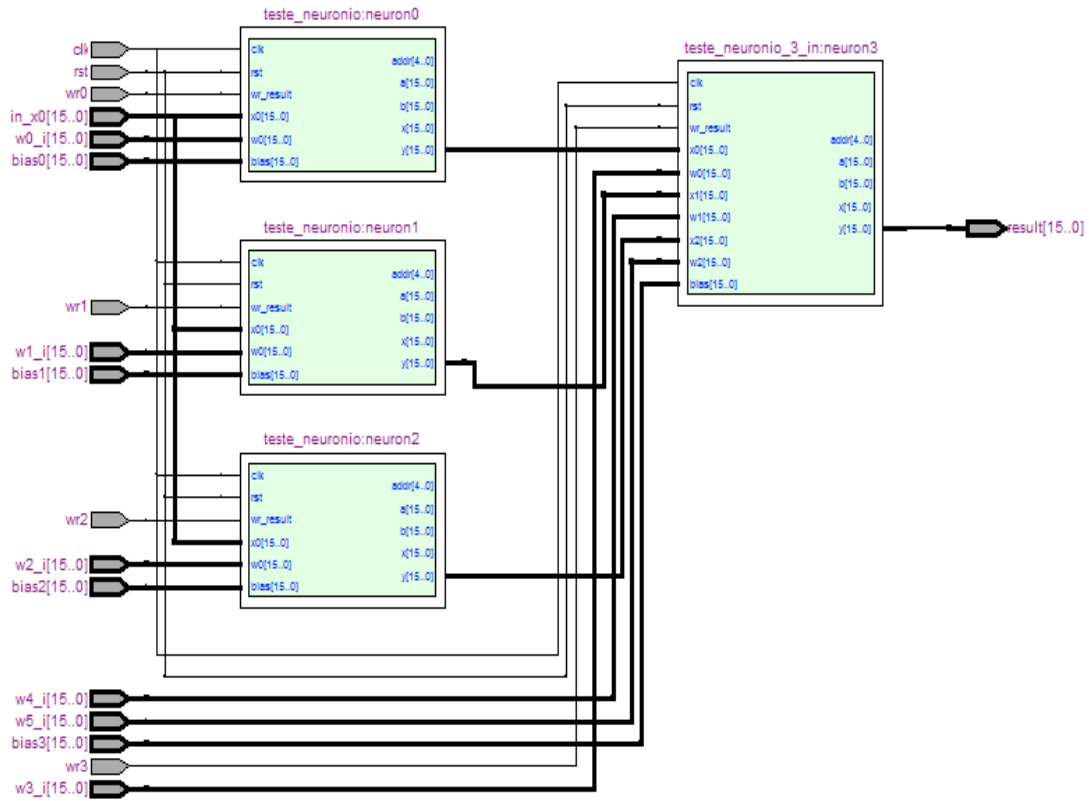
end calc_net;
architecture bloco of calc_net is

begin
    process(dataa,datab,datax)
        variable a,b,x,aux,r : integer range -2**(data_length-1)-1 to 2**(data_length-1);
--        variable a,b,x,r : signed(data_length-1 downto 0);
--        variable aux0,aux1 : signed(2*data_length-1 downto 0);
    begin
        a := to_integer(signed(dataa));
        b := to_integer(signed(datab));
        x := to_integer(signed(datax));
        aux := a*x/2**(data_length-1);
        aux := aux*364950/2**(data_length-1);
        r := aux + b;
        y <= to_signed(r,data_length);

--        a := signed(dataa);
--        b := signed(datab);
--        x := signed(datax);
--        aux0 := a*x;
--        aux1 := shift_right(aux0,4);
--        r := aux1(data_length-1 downto 0) + b;
--        y <= std_logic_vector(r);
    end process;
end bloco;

```

APENDICE B: ARQUITETURA DE UM *PERCEPTRON* DE MÚLTIPLAS CAMADAS COM TRÊS NEURÔNIOS NA CAMADA OCULTA E UM NA CAMADA DE SAÍDA.



APENDICE B: ARQUITETURA DE UM *PERCEPTRON* DE MÚTIPLAS CAMADAS COM CINCO NEURÔNIO NA CAMADA OCULTA E UM NA CAMADA DE SAÍDA.

