UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Uma arquitetura reconfigurável de Rede Neural Artificial utilizando FPGA

Janaína da Glória Moreira de Oliveira

Itajubá, Maio de 2017

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Janaína da Glória Moreira de Oliveira

Uma arquitetura reconfigurável de Rede Neural Artificial utilizando FPGA

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica como parte dos requisitos para obtenção do Título de Mestre em Ciências em Engenharia Elétrica.

Área de Concentração: Microeletrônica

Orientador: Robson Luiz Moreno
Coorientador: Odilon de Oliveira Dutra

Maio de 2017
Itajubá - MG

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Uma arquitetura reconfigurável de Rede Neural Artificial utilizando FPGA

Janaína da Glória Moreira de Oliveira

Dissertação submetida a banca...

Banca Examinadora:
Prof. Dr. Robson Luiz Moreno
Prof. Dr. Odilon de Oliveira Dutra

Itajubá
2017

"Não importa, não faz mal
Você ainda pensa e é melhor do que nada
Tudo que você consegue ser
Ou nada!"
Tudo o que você podia ser, Lô Borges e Márcio Borges

Agradecimentos

Primeiramente, gostaria de agradecer e dedicar este trabalho aos meus pais Francisco e Gilda e ao meu irmão Gabriel pelo apoio e compreensão não somente durante este projeto, como em toda minha vida.

Agradeço aos meus amigos e amigas e também ao meu namorado que sempre me apoiaram e me ajudaram durante este trabalho.

Agradeço também aos professores Robson Luiz Moreno e Odilon de Oliveira Dutra pela paciência e dedicação durante este estudo.

Agradeço à CNPq que através da disponibilidade de bolsas de mestrado viabilizou financeiramente este trabalho.

Resumo

Este trabalho apresenta uma nova implementação em hardware de Rede Neural Artificial que permite reconfiguração da arquitetura que é implementada. Este tipo de design é importante em aplicações em que o ambiente varia de tal maneira que é necessária uma mudança na arquitetura da Rede Neural para que os resultados continuem adequados. A topologia usada foi a MultiLayer Perceptron, onde os neurônios são organizados em camadas e cada camada recebe como entrada as saídas da camada anterior, ou seja, elas têm uma execução sequencial. A implementação desenvolvida permite mudanças no número de neurônios de cada camada, número de entradas e saídas da Rede Neural e do tipo de função de ativação que os neurônios de cada camada irão executar. Apesar de implementada em FPGA, a Rede Neural proposta não depende de nenhum de seus modelos, já que nenhum bloco proprietário foi usado. Esta característica permite que o sistema aqui proposto seja implementado com facilidade em um circuito integrado a ser usado em implantes médicos, por exemplo. A Rede Neural foi submetida a três testes práticos que provaram seu funcionamento e os resultados em termos de erros atingidos foram analisados.

Palavras Chaves: Rede Neural Artificial, Multi-Layer Perceptron, Reconfiguração, FPGA.

Abstract

This work presents a new hardware implementation of Artificial Neural Network which allows reconfiguration of the architecture implemented. This type of design is important in applications where the environment vary in such a way that it is necessary a change in Neural Network architecture to keep results corrects. The topology used was the Multi-Layer Perceptron, where neurons are arranged in layers and each layer receives as input the outputs of the previous layer, i.e. they have a sequential execution. The implementation developed allows changes in the number of neurons in each layer, number of inputs and outputs of the Neural Network and the type of activation function that each layer will perform. Although implemented in FPGA, the Neural Network proposed does not depend on any of their models, since no block owner was used. This feature allows the system proposed here to be easily implemented in an integrated circuit. The Neural Network was submitted to three practical tests that proved its functioning and the results in terms of errors were analyzed.

Key-words: Artificial Neural Network, Multi-Layer Perceptron, Reconfiguration, FPGA.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

1.1 Motivação

Redes Neurais Artificiais (RNAs) são sistemas baseados no funcionamento do cérebro humano e em sua capacidade de adaptar, aprender e generalizar (1). São soluções importantes em problemas de controle, diagnósticos médicos, reconhecimento de padrões, aproximações, projeções ou problemas em que a operação não é completamente entendida ou que nem todos os parâmetros estão disponíveis (1, 2, 3, 4). O neurônio é o principal elemento da RNA e as Redes Neurais Artificiais são formadas por neurônios interconectados entre si de modo que o tipo de interconexão, chamado de topologia, define qual função a Rede Neural Artificial (RNA) executará (3).

Várias topologias de RNAs já foram definidas na literatura(1, 3, 5), cada uma delas adequada para um tipo de problema (1, 2). Uma das mais importantes e mais usadas topologias é a *MultiLayer Perceptron* (MLP) (2, 6), onde os neurônios são ajustados em camadas e, cada camada recebe como entrada o resultado dos neurônios da camada anterior.

Outro fato importante no desenvolvimento da RNA é se ela será implementada em software ou hardware (4, 7, 8). Implementações em software tem um design simples e rápido, e por isso permitem que se tenha uma grande flexibilidade na topologia da Rede Neural Artificial (3). Porém, a velocidade de operação é baixa, já que se trata de um processador sequencial simulando o comportamento paralelo da RNA (9). Deste fato vem a vantagem do uso de sistemas em hardware, já que o intrínseco paralelismo da Rede Neural Artificial pode ser usado, trazendo sistemas rápidos e com maior eficiência em termos de área (3, 4, 7). Entretanto, o desenvolvimento de hardware é lento e mais complexo que o desenvolvimento de sistemas em software(10). Para tentar ter as vantagens tanto de software quanto de hardware, ou seja, um rápido desenvolvimento com altas velocidades de operação, faz-se uso de FPGAs (Field Programmable Gate Arrays), que são plataformas que permitem um rápido desenvolvimento de sistemas digitais, além de prover flexibilidade ao hardware (8, 10, 11). É interessante, porém, que a implementação da RNA seja independente de células proprietárias da FPGA, para que no futuro, ela possa ser usada em implementações de circuito integrado (CI).

Uma característica interessante para RNAs é que elas sejam capazes de se adaptar ao ambiente em que estão inseridas, principalmente se este ambiente é de difícil acesso, como sistemas médicos implantados, sistemas para missões espaciais, comunicação móvel, entre outros, já que a modificação do hardware se torna difícil. Surge então a necessidade

de Redes Neurais Artificiais Reconfiguráveis em hardware. Este tipo de RNA procura trazer um sistema com possibilidade de modificação da arquitetura da RNA, enquanto mantém fixa a área usada do hardware (3, 12).

1.2 Objetivos

O objetivo deste trabalho é o desenvolvimento de uma Rede Neural Artificial capaz de se reconfigurar em tempo de execução, mantendo sua área fixa. Visando uma rápida prototipação e validação da arquitetura da Rede Neural Artificial proposta, a RNA foi implementada em FPGA, mantendo uma independência de células proprietárias para que esta arquitetura possa ser usada em futuras implementações de CIs. Também, a RNA a ser desenvolvida não deverá depender de sistemas em software para executar, o que garante que a modificação do circuito seja de forma fácil e simples.

Objetivos específicos:

- Encontrar uma topologia de multiplicador digital que melhor se encaixe na Rede Neural Artificial a ser desenvolvida;
- Encontrar aproximações de funções não lineares para que a Rede Neural Artificial implemente;
- Encontrar uma arquitetura que garanta que a Rede Neural Artificial seja reconfigurável a tempo de execução, sem grande perda em sua operação;
- Desenvolver uma implementação de RNA que não seja muito custosa em termos de área e que atinja altas velocidades de processamento;
- Provar o funcionamento da Rede Neural Artificial por meio de testes práticos.

1.3 Organização do trabalho

Este trabalho se distribui ao longo de 5 capítulos. No Capítulo 2 é feita uma revisão da bibliografia de Redes Neurais Artificiais (RNAs). Todo o funcionamento da Rede Neural Artificial e sua similaridade com o cérebro humano são discutidos. Também, diferentes formas de implementação de Redes Neurais Artificiais, como sistemas em software, sistemas analógicos, digitais e em FPGA, são apresentadas e suas vantagens e desvantagens são discutidas. Por fim, é feita uma revisão de métodos de aproximação de funções não lineares e de implementação de multiplicadores digitais.

No Capítulo 3 a arquitetura aqui proposta é discutida. São dados detalhes de implementação e o seu funcionamento é analisado. Também são mostradas cartas de tempo de cada etapa do funcionamento da Rede Neural Artificial.

O Capítulo 4 traz simulações de três testes: aproximação de função, classificação de padrões e previsão de série temporal. Os testes são analisados em detalhes e os resultados e erros são mostrados e discutidos.

Por fim, o Capítulo 5 traz conclusões sobre a Rede Neural Artificial Proposta e sugestões para trabalhos futuros a serem desenvolvidos.

Também são incluídos alguns anexos com detalhes das aproximações de funções não lineares.

2 Revisão Bibliográfica

2.1 Redes Neurais Artificiais

O cérebro humano é uma estrutura com milhares de neurônios formando um sistema complexo com capacidade de aprendizado e generalização. Acredita-se que a imensa capacidade de computação presente no cérebro humano seja um reflexo da operação paralela e distribuída feita pelos neurônios (4).

A observação do funcionamento dos neurônios levou ao estudo e criação de Redes Neurais Artificiais, que são, em geral, um modelo que tem como objetivo operar de maneira semelhante ao que o cérebro humano trabalha (13). Em outras palavras, as RNAs podem ser vistas como um modelo matemático do cérebro humano (14, 15).

Algumas características do cérebro são buscadas pelas Redes Neurais Artificiais, pois ultrapassam limitações de sistemas de computação clássicos, como por exemplo (14, 16, 17, 18):

- Paralelismo massivo: Redes Neurais Artificiais são construídas de forma que os neurônios operam de maneira paralela, permitindo altas velocidades de processamento (18, 19);
- Capacidade de aprender com exemplos: a Rede Neural Artificial recebe grupos de entradas e suas respectivas saídas e ajusta os pesos de suas conexões a fim de mapear a relação entre entrada/saída. Este processo é chamado de aprendizado (14);
- Capacidade de generalização: as Redes Neurais Artificiais têm capacidade de aprender regras através de treinamento e, através da informação previamente aprendida, responder a novos padrões de entrada (18, 16);
- Não linearidade: capacidade de a Rede Neural Artificial resolver problemas não lineares (14);
- Tolerância à falhas: os neurônios presentes na Rede Neural Artificial estão distribuídos e operando em paralelo. Assim, a perda de algum neurônio não afeta significativamente a execução da rede, já que ela pode redistribuir os pesos para que o sistema continue a operar de forma correta (14, 16, 18);
- Adaptação dinâmica: os pesos das Redes Neurais Artificiais podem se adaptar à mudanças no ambiente (19).

Estas características das Redes Neurais Artificiais permitem a solução de problemas complexos, não lineares e até mesmo problemas em que nem todos os parâmetros são conhecidos ou de solução ainda não entendida, onde a computação convencional não tem sucesso ou pode ter os resultados melhorados (1, 2, 3, 4). Vários tipos de problemas se encaixam nestas características, entre eles pode-se citar reconhecimento e classificação de padrões, processamento de imagens, processamento de sinais, aproximações, controle de sistemas, projeções, diagnósticos médicos, otimização, entre outros (1, 2, 3, 4, 16, 18, 20).

Uma vez que a Rede Neural Artificial é proposta para simular o comportamento do cérebro humano, ela é formada por um conjunto de neurônios, que são unidades de processamento simples interconectados entre si através de conexões unidirecionais com pesos associados (13, 17, 18). A Figura 1 exibe um exemplo de Rede Neural Artificial.

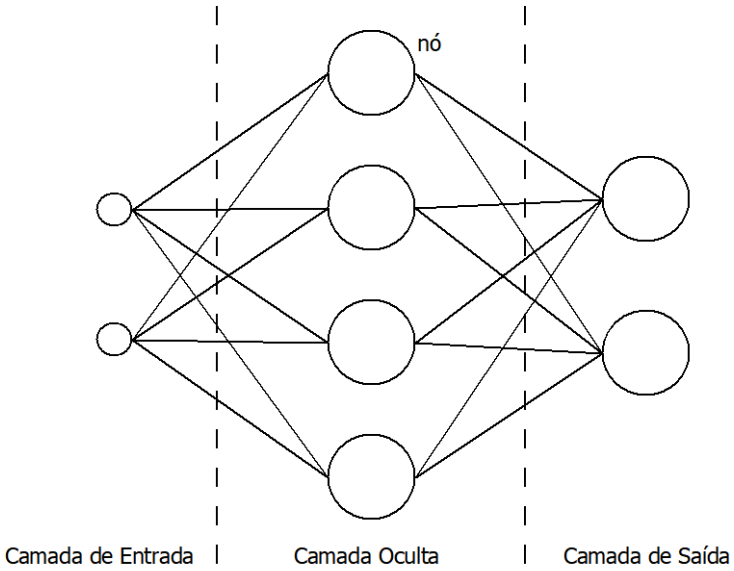

Figura 1 – Rede Neural Artificial

Na Figura 1, os nós representam os neurônios e as linhas representam a conexão entre eles. Enquanto os neurônios são responsáveis pelo processamento aritmético da rede e operam em paralelo entre si, os pesos das conexões são onde o conhecimento da rede está armazenado. Cada neurônio executa algumas operações simples, mas quando conectados em uma rede, eles exibem um comportamento geral complexo (21). Com este tipo de formação, a RNA consegue ter um processamento massivamente paralelo e distribuído: o conhecimento é armazenado nas conexões, as informações são distribuídas através da Rede Neural Artificial e processadas paralelamente por vários neurônios conectados entre si (15, 16).

A função exercida pela RNA é definida pela forma em que os neurônios estão conectados entre si e pelos valores dos pesos atribuídos a essas conexões (17). O ajuste destes pesos de acordo com o problema a ser solucionado é conhecido como treinamento da RNA (14). Estas características aumentam a capacidade da Rede Neural Artificial de

resolver problemas complexos e com alta velocidade.

Dois pontos são importantes para a operação de uma RNA: o funcionamento do neurônio, já que este é a unidade básica de processamento da Rede Neural Artificial, e a estrutura de conexão dos neurônios, mais conhecida como topologia da Rede Neural Artificial. As próximas seções discutem com detalhes estes assuntos.

2.1.1 Neurônio

O neurônio é a unidade de processamento de informação da Rede Neural Artificial. Ele tem sua operação bem definida e seu funcionamento é baseado no neurônio biológico (3, 20, 22)(Figura 2).

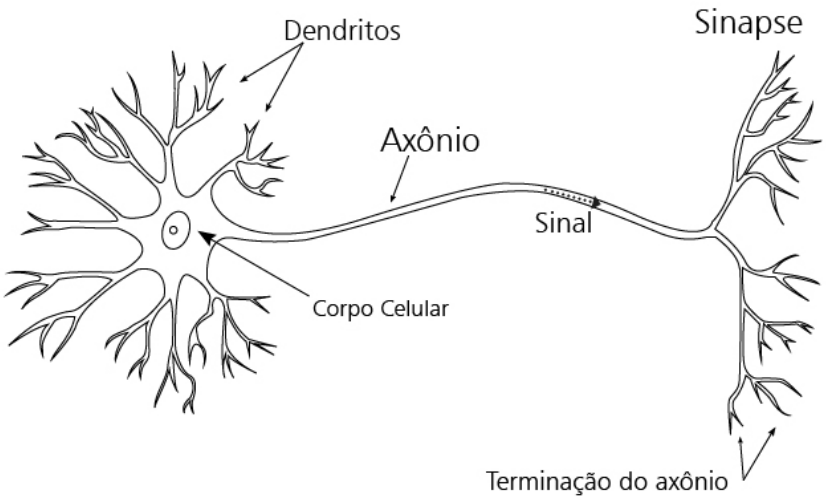

Figura 2 – Neurônio Biológico

No neurônio biológico, os sinais passados são impulsos elétricos excitatórios ou inibitórios que entram através dos dendritos, passam pelo corpo celular e são transmitidos através do axônio e pelas sinapses para outros neurônios (22). A transmissão de sinais é um processo químico complexo. As entradas recebidas são combinadas e se seu potencial atinge um certo nível, o neurônio é ativado e passa o sinal através das sinapses para os próximos neurônios a ele conectados (1).

Com o objetivo de simular o funcionamento de neurônio biológico, em 1943, McCulloch e Pitts propuseram o primeiro neurônio artificial (23). Ele explora o caráter "tudo ou nada" do neurônio biológico, significando que a saída só é ativada caso a entrada atinja um certo nível e caso contrário, a saída fica desativada. As entradas e saídas são sinais binários e o modelo matemático é bem definido (20). As entradas funcionam como estímulos do ambiente e cada uma delas tem um peso fixo associado. A saída é definida através da função degrau, ou seja, se a soma das entradas multiplicadas por seus pesos

for maior que um limite, a saída é ativada; caso contrário a saída fica desativada (1, 22). Vários problemas booleanos básicos podem ser resolvidos com a utilização deste tipo de neurônio (20). Porém, como o neurônio de McCulloch e Pitts só trabalha com entradas binárias, os problemas que podem ser resolvidos por ele se tornam limitados.

Em 1949, Hebb postulou as primeiras regras para o treinamento dos neurônios (14). Em 1958, Rosenblatt propôs um novo modelo de neurônio, chamado de Perceptron, que permitia um treinamento supervisionado (14, 20).

Ressalta-se que o treinamento é uma característica importante para o neurônio pois permite que o neurônio aprenda e se adapte ao problema que deve resolver, dando maior poder à Rede Neural Artificial.

O modelo Perceptron (Figura 3) é usado até hoje, e sozinho, é capaz de classificar padrões linearmente separáveis.

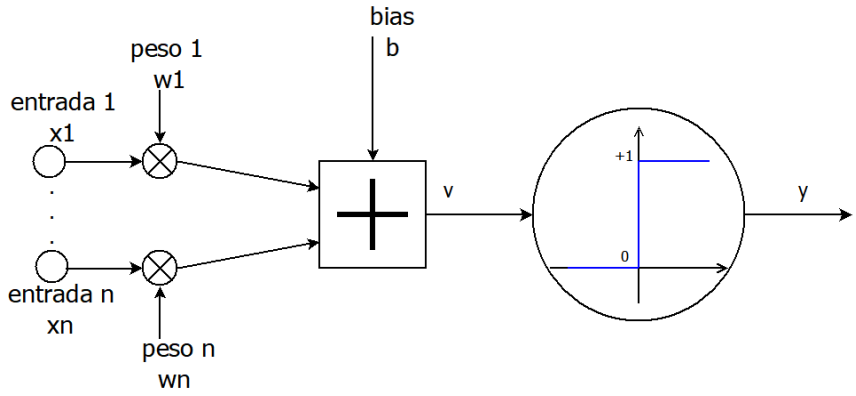

Figura 3 – Perceptron de Rosenblatt

As entradas do Perceptron (x_i), ao contrário do neurônio de McCulloch e Pitts, são números reais, aumentando a gama de problemas que podem ser resolvidos por este neurônio. Cada entrada tem um peso relacionado (w_i). Também há um sinal de *bias* (b) que tem como função adicionar um grau de liberdade ao neurônio. Este sinal funciona como um peso multiplicado a uma entrada de valor 1. As entradas dos neurônios são multiplicadas pelos seus respectivos pesos e em seguida somadas ao *bias*, chegando a um valor v, como mostrado a seguir:

$$v = \sum_{i=1}^{n} x_i.w_i + b \tag{2.1}$$

Em seguida, este valor v é usado como entrada da função de ativação Degrau. Esta função dá como saída 1 se o valor v é maior ou igual que um limite e 0, caso contrário:

$$y = \begin{cases} 0, & v < 0 \\ 1, & v \geq 0 \end{cases} \tag{2.2}$$

A saída y da função degrau define a saída do neurônio.

A multiplicação dos pesos pelas entradas indica o quanto cada entrada influencia na saída do neurônio. Os pesos e *bias* são ajustáveis através de treinamento, permitindo que o neurônio seja usado em vários problemas diferentes.

Apesar do Perceptron sozinho ser capaz de resolver somente problemas linearmente separáveis e separá-los em somente duas classes, ao se criar Redes Neurais Artificiais com mais de um neurônio pode-se resolver problemas mais complexos (14). As RNAs compostas por camadas de Perceptrons são chamadas de *MultiLayer Perceptron* (MLP) e são formadas por neurônios Perceptron organizados em camadas, onde a saída de um neurônio é conectada na entrada dos neurônios da camada seguinte, propagando a informação da entrada da RNA até a saída.

Uma evolução natural do Perceptron é permitir outras funções de ativação para classificar sistemas não-linearmente separáveis. Surgem então Perceptrons onde as funções de ativação podem ser de vários tipos e não só a função degrau. O modelo deste neurônio pode ser visto na Figura 4 e seu funcionamento é semelhante ao neurônio da Figura 3, a única diferença é o cálculo da função de ativação.

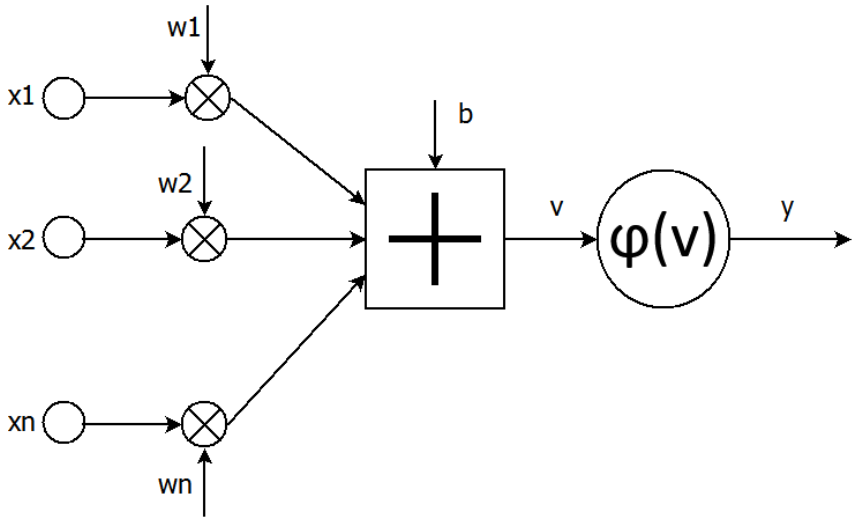

Figura 4 – Perceptron com uma função de ativação diferente

A equação que agora define a saída do neurônio é

$$y = \varphi\left(\sum_{i=1}^{n} x_i.w_i + b\right) \tag{2.3}$$

A função de ativação φ controla a saída do neurônio através da soma ponderada v, transformando os dados de entrada em uma saída (24, 25). Ela tem como objetivo limitar a saída do neurônio em um dado intervalo e se necessário, inserir uma não linearidade ao neurônio (26). A escolha da função de ativação pode mudar consideravelmente o com-

portamento do neurônio (24), sendo assim uma etapa importante no desenvolvimento de Redes Neurais Artificiais.

Em geral, qualquer função pode ser usada como função de ativação de um neurônio artificial. Elas podem ser divididas em dois grandes grupos: função lineares e funções não lineares. As funções não lineares são interessantes pois permitem que diferentes funções possam ser aproximadas pela Rede Neural Artificial e que esta possa resolver problemas mais complexos (27). Outra característica importante das funções de ativação é se estas são ou não totalmente diferenciáveis. Funções que possuem derivadas em todos os pontos (isto é, funções totalmente diferenciáveis) são interessantes pois permitem o treinamento do tipo *backpropagation* (que será detalhado adiante) (25).

Entre as principais funções de ativação usadas em redes MLP, pode-se citar a função Degrau, Degrau Bipolar e Rampa Limitada como exemplos de função lineares e Sigmóide Logística e Tangente Hiperbólica como exemplos de funções não lineares (9, 25). A escolha entre elas depende da arquitetura e da aplicação da Rede Neural Artificial a ser usada.

Detalhes destas funções são mostrados a seguir (3, 28).

2.1.1.1 Função Degrau

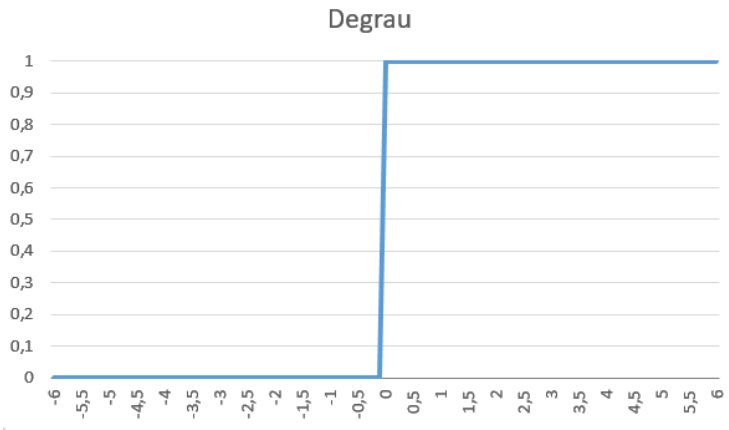

Figura 5 – Função Degrau

Esta função é a mais simples e foi usada no neurônio proposto por McCulloch e Pitts. A equação que a define é mostrada a seguir:

$$y(v) = \begin{cases} 1, & v \geq 0 \\ 0, & v < 0 \end{cases} \tag{2.4}$$

A saída é definida em 1 se a entrada é maior ou igual a 0 e é definida em 0 se a entrada é menor que 0. Vale notar que esta função não tem derivada em todos os pontos, pois há uma descontinuidade da função quando a entrada é 0.

2.1.1.2 Função Degrau Bipolar

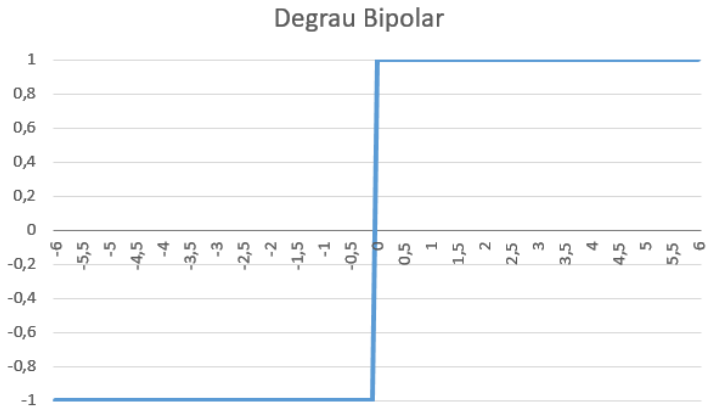

Figura 6 – Função Degrau Bipolar

Esta função também é conhecida como função sinal, pois sua saída é igual ao sinal da entrada. Sua equação é mostrada a seguir:

$$y(v) = \begin{cases} 1, & v \geq 0 \\ -1, & v < 0 \end{cases} \tag{2.5}$$

Se a entrada for maior ou igual a 0, a saída é iguala 1. Caso a entrada for menor que 0, a saída é definida como -1. Assim como na função Degrau, esta função não tem derivada em todos os pontos, pois também há descontinuidade quando a entrada é 0.

2.1.1.3 Função Rampa Limitada

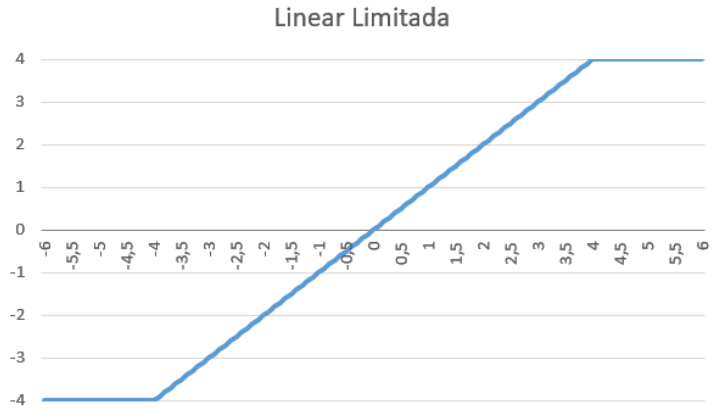

Figura 7 – Função Rampa Limitada

Esta função tem uma resposta linear em um intervalo e uma resposta constante

se a entrada está fora deste intervalo.

$$y(v) = \begin{cases} a, & v \geq a \\ v, & b < v < s \\ b, & v \leq b \end{cases} \tag{2.6}$$

Nota-se que esta função pode ter diversos limitadores, dependendo da solução esperada ou da precisão da rede a ser desenvolvida.

A função Rampa Limitada também não tem derivada em todos os pontos, já que há descontinuidade quando a entrada é igual a a ou igual a b.

2.1.1.4 Função Sigmóide Logística

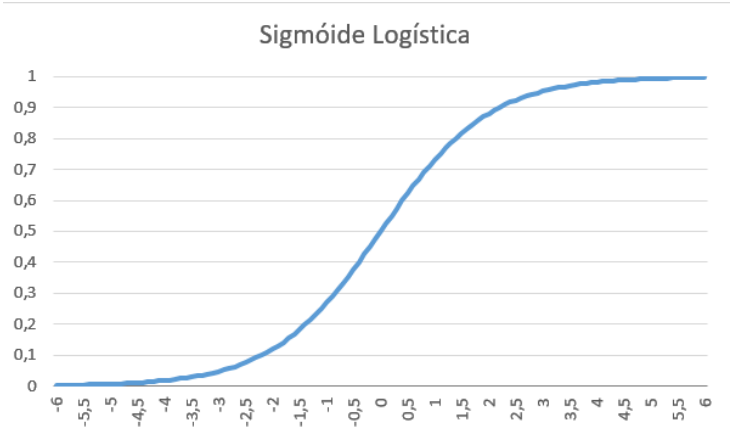

Figura 8 – Função Sigmóide Logística

A função Sigmóide Logística é uma função totalmente diferenciável que tem seu funcionamento próximo ao da função Degrau, fazendo assim com que se aproxime da função exercida pelo neurônio biológico (16). O fato de existirem derivadas em todos os pontos faz com que a função permita o treinamento do tipo *backpropagation*, que será visto adiante. A Sigmóide Logística pode ser entendida como a probabilidade de o neurônio ser ativado ou não (27).

$$y(v) = \frac{1}{1 + e^{-v}} \tag{2.7}$$

A saída da função está entre 0 e 1. Para valores de entrada muito grandes a saída tende a 1 e para valores muito pequenos a saída tende a 0. Outra característica importante desta função é a simetria entre as partes com entrada positiva ou negativa no ponto (0;0,5), definida como:

$$y(-v) = 1 - y(v) \tag{2.8}$$

A função também tem sua parte central semelhante a uma função linear. Esta característica pode ajudar quando se pretende implementar a Sigmóide Logística em estruturas de hardware digital.

Devido à sua robustez, a Sigmóide Logística é uma função bastante usada em redes MLP, entretanto, como já foi dito, se as entradas forem altamente negativas, a saída se aproxima de zero. Esta característica pode levar a um aprendizado mais lento. Uma solução para estes casos é o uso da função Tangente Hiperbólica (27).

2.1.1.5 Função Tangente Hiperbólica

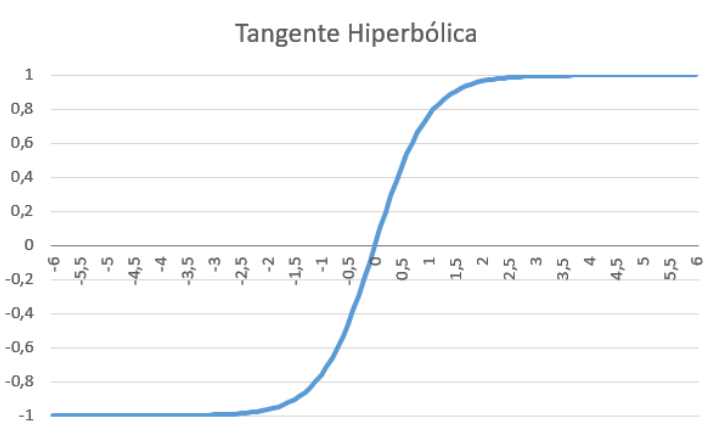

Figura 9 – Função Tangente Hiperbólica

A função Tangente Hiperbólica tem forma semelhante à função Sigmóide Logística, porém é definida entre -1 e 1 e tem uma precisão mais refinada (29). Ela também possui derivada em todos os pontos, permitindo o treinamento do tipo *backpropagation*.

$$y(v) = \frac{1 - e^{-v}}{1 + e^{-v}} \tag{2.9}$$

A simetria desta função é no ponto (0;0):

$$y(-v) = -y(v) \tag{2.10}$$

Esta função se mostrou robusta em seu uso nas MLPs nas mais variadas aplicações (25). Também, em alguns casos ela é preferida à função Sigmóide Logística, uma vez que seu aprendizado é mais rápido (27).

O neurônio Perceptron da Figura 4 é o modelo de neurônio mais usado atualmente (3). Vale notar que estes neurônios operam somente com seus dados locais e com as entradas que recebem através de suas conexões. As entradas dos neurônios podem vir tanto do meio externo quando da saída de outros neurônios (15) e cada uma delas tem

um peso associado que indica qual a participação desta entrada na saída do neurônio. A função do neurônio é simples: receber as entradas de fontes externas ou de outros neurônios e calcular um sinal que pode ser enviado tanto para outros neurônios quanto para o meio externo (13).

Os pesos e *bias* indicam o conhecimento da Rede Neural Artificial e são atualizados durante o treinamento da RNA (14). Como pesos devem ser multiplicados pelas entradas dos neurônios, cada entrada deva ter um multiplicador associado para garantir que seu funcionamento seja paralelo.

A saída y do neurônio é única e pode ser propagada para o meio externo ou dada como entrada para outros neurônios (15).

Nota-se pela equação que define a operação do neurônio (Equação 2.3) que a multiplicação das entradas pelos pesos é a operação mais computacionalmente relevante durante a execução, sendo assim, um ponto de atenção quando se desenvolve Redes Neurais Artificiais.

Apesar das funções de ativação inserirem certa não linearidade no neurônio, elas ainda não garantem que um único neurônio seja capaz de executar operações de maior dificuldade (27). Isto leva à necessidade de agrupar os neurônios de forma que sistemas complexos possam ser resolvidos, surgindo assim, redes de neurônios interconectados. A maneira que os neurônios podem se conectar, também conhecida como topologia da Rede Neural Artificial, e as características de cada uma delas serão vistas a seguir.

2.1.2 Topologias de Redes Neurais Artificiais

A topologia de uma Rede Neural Artificial é definida como a maneira com que os neurônios estão conectados entre si e pode ser dividida em Recorrente e Não Recorrente. Técnicas de treinamento podem ser escolhidas para a Rede Neural Artificial, dependendo de sua topologia, a fim de calcular o peso das conexões e trazer aprendizado a ela (18).

Os dois tipos de topologia serão detalhados a seguir.

2.1.2.1 Não recorrente

Nesta topologia, também chamada de rede *feedforward*, a informação se propaga da entrada para a saída, somente em um sentido (13). A estrutura é organizada em forma de camadas e como não existem laços de realimentação, a saída de uma camada não afeta o funcionamento da mesma, ou seja, neste tipo de Rede Neural Artificial a saída de uma camada só é determinada por sua entrada e pelos valores dos pesos das conexões (4, 13). Um exemplo de rede *feedforward* pode ser visto na Figura 10.

Esta topologia é organizada na forma de camadas, de forma que o neurônio na

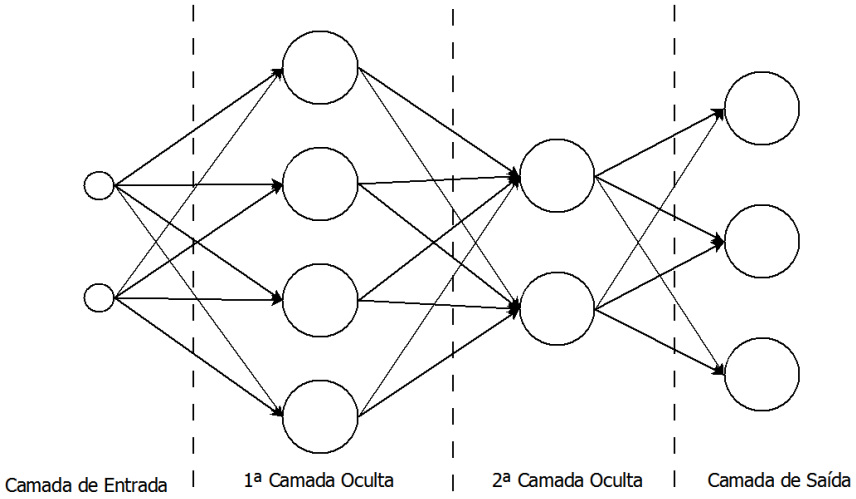

Figura 10 – Rede Neural Artificial *feedforward*

camada *i* recebe suas entradas da camada *i-1* e passa sua saída para a camada *i+1* (4). A primeira camada deste tipo de rede é chamada de camada de entrada e recebe os dados do meio externo. A última camada é chamada de camada de saída e envia seus dados para o meio externo como os resultados da Rede Neural Artificial. As demais camadas são chamadas de camadas ocultas. O número de camadas presentes em uma Rede Neural Artificial varia de acordo com a aplicação e a definição de quantas camadas são necessárias ainda é um fator experimental (20). Alguns exemplos de Redes Neurais Artificiais não recorrentes são redes Adaptative Linear Element (Adaline), Radial Basis Function (RBF) e Multiplayer Perceptron. Todas essas topologias são bastante usadas em reconhecimento de padrões (13).

Devido a sua importância, as Redes Neurais Artificiais do tipo MLP serão posteriormente comentadas.

2.1.2.2 Recorrente

Esta topologia possui laços de realimentação, ou seja, permite que os sinais percorram ambas as direções (13). Isto significa que os resultados de uma camada podem ser usados no cálculo da mesma, já que a saída de uma camada pode ser realimentada para a mesma ou para camadas anteriores. Um exemplo é mostrado na Figura 11.

Como os sinais trafegam em duas direções, esta topologia de RNA tem comportamento dinâmico e podem se tornar tanto poderosas quanto complexas (1). Exemplos de Redes Neurais Artificiais recorrentes são Redes de Hopfield e Kohonen's Self Organization Map (SOM).

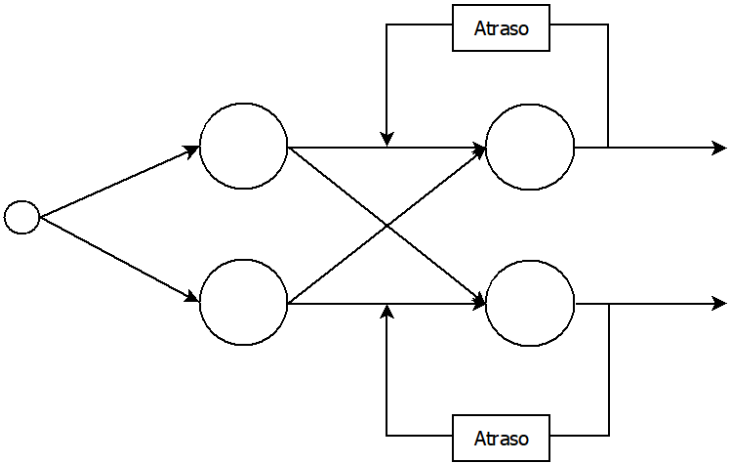

Figura 11 – Rede Neural Artificial Recorrente

2.1.3 Funcionamento de Redes Neurais Artificiais

O funcionamento de uma Rede Neural Artificial se dá em duas etapas: Treinamento e Operação Direta. A Operação Direta já foi mostrada nas seções anteriores e é a operação da RNA, partindo das entradas até atingir as saídas. Cada neurônio opera de maneira paralela e passa sua saída adiante, para que os próximos neurônios possam executar novamente, até que a saída da RNA esteja disponível.

Na fase de Treinamento, os pesos e *bias* são ajustados para a operação que a RNA deve executar, ou seja, no treinamento a Rede Neural Artificial aprende sua função. O treinamento da RNA pode ser classificado em Supervisionado e Não Supervisionado.

No treinamento Supervisionado, a rede é apresentada a entradas e suas respectivas saídas esperadas (16). A Rede Neural Artificial calcula a saída para pesos inicialmente aleatórios e então avalia o erro entre a saída esperada e a obtida. A partir deste erro os pesos e *bias* da RNA são ajustados. A Rede Neural Artificial calcula novamente a saída e continua esse processo até que o erro entre a saída esperada e a obtida seja mínimo (1). Ou seja, a RNA modela a relação entre entrada e saída (13). Com o treinamento finalizado, é esperado que a Rede Neural Artificial produza uma saída correta para uma entrada desconhecida (2), ou seja, que a RNA aprenda a solucionar entradas desconhecidas através dos exemplos apresentados. A estrutura geral do treinamento Supervisionado é vista na Figura 12.

Já no treinamento Não Supervisionado, somente a entrada é dada à Rede Neural Artificial, sem a resposta esperada (16). A RNA adapta seus pesos de maneira independente (13), explorando a correlação entre as entradas apresentadas, classificando-as em categorias similares e, através das semelhanças e diferenças entre as entradas, ajustando os pesos (1, 2). O treinamento Não Supervisionado é útil em casos em que a saída esperada não é conhecida, permitindo que a RNA em treinamento descubra qualquer regra que dê uma resposta correta a uma dada entrada (2). A estrutura é vista na Figura 13.

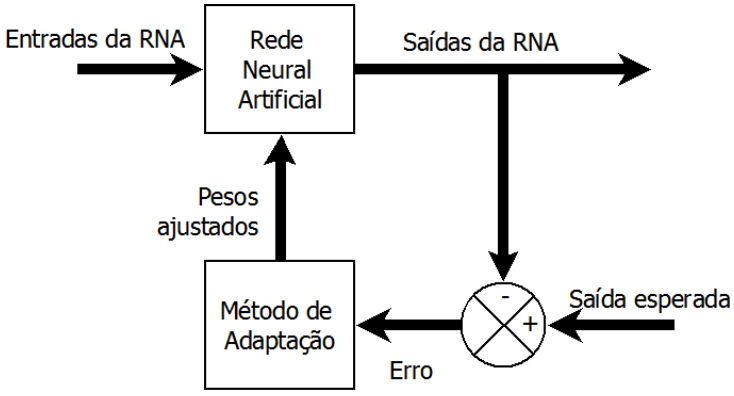

Figura 12 – Treinamento Supervisionado

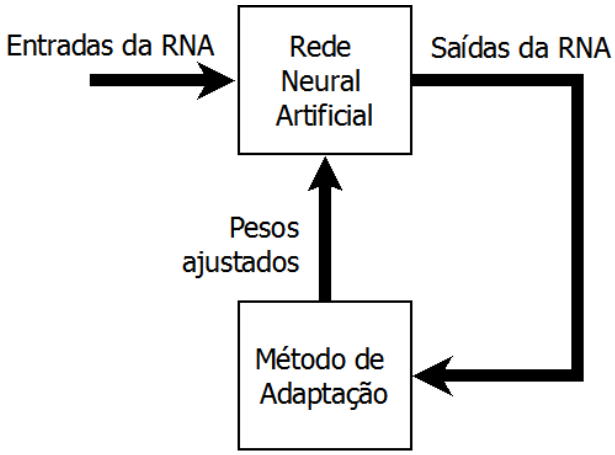

Figura 13 – Treinamento Não Supervisionado

2.1.4 Redes Neurais Artificiais MLP com Treinamento *Backpropagation*

Uma das topologias de RNAs mais usadas é a MLP, ou *MultiLayer Perceptron*, devido à sua versatilidade e facilidade de implementação (1, 30). Uma das características mais importantes de redes MLP é que elas podem aprender mapeamentos de complexidade arbitrária e após o treinamento, soluções para problemas complexos ou sem solução conhecida podem ser encontradas (22).

Devido à sua versatilidade, redes MLP podem ser usadas para a solução dos mais variados tipos de problemas, como aproximação de funções, reconhecimento de caracteres, classificação de padrões, reconhecimento de fala, controle dinâmico, previsão, memórias associativas, otimização, soluções médicas, entre outros (1, 20, 22). Por isso, neste trabalho, a topologia de RNA usada será a MLP.

Esta RNA é organizada em camadas e não permite realimentação, ou seja, é uma Rede Neural Artificial não recorrente. Sua estrutura geral já foi vista na Figura 10.

A primeira camada é chamada de camada de entrada e não é formada por neurônios, mas sim elementos sem processamento, responsáveis por receber as entradas da rede. As demais camadas são formadas por neurônios do tipo Perceptron. Podem existir tantas

camadas ocultas quanto necessário para a solução do problema. O número de neurônios em cada camada também varia de acordo com o problema a ser solucionado. Como já dito anteriormente, as camadas recebem como entradas as saídas das camadas anteriores, ou seja, para que uma camada execute, todas as camadas anteriores já devem ter sido executadas.

Um método de treinamento bastante usado para esta topologia de Rede Neural Artificial é o treinamento supervisionado *Backpropagation*. Para este treinamento, é necessário que as funções de ativação dos neurônios sejam diferenciáveis (14).

O *backpropagation* inicia com pesos aleatórios para os neurônios. Um grupo de entradas é então colocado na RNA que calcula a saída referente à estas entradas. A fórmula que define a saída de cada neurônio é mostrada na Equação 2.11.

$$y_i^l = f\left(v_i^l\right) = f\left(\sum_{j=1}^{N^{l-1}} \left(w_{ij}^l . y_j^{(l-1)}\right) + b_i^l\right) \tag{2.11}$$

Onde:

l: número da camada em análise;

L: número máximo de camadas;

N^{l-1}: número máximo de neurônios na camada $l-1$;

y_i^l: saída do neurônio em análise;

f: função de ativação do neurônio em análise;

v_i^l: soma ponderada das entradas do neurônio em análise;

y_j^{l-1}: saída do neurônio j da camada anterior;

w_{ij}^l: peso referente à entrada j do neurônio i em análise;

b_i^l: *bias* do neurônio em análise.

A saída obtida é então comparada com a saída esperada e um valor de erro é calculado, para que então os pesos e *bias* possam ser atualizados. Sendo t_i o valor esperado para o neurônio i da camada de saída (camada L), tem-se que o erro é dado por:

$$\varepsilon_i^l = \begin{cases} t_i - y_i^l, & l = L \\ \sum_{j=1}^{N^{l+1}} \left(w_{ji}^{(l+1)} . \delta_j^{(l+1)}\right), & l = 1, 2, ..., L-1 \end{cases} \tag{2.12}$$

Onde:

L: número máximo de camadas;

ε_i^l: erro do neurônio i da camada l;

$w_{ji}^{(l+1)}$: é o peso do neurônio da camada seguinte associado com a saída do neurônio

atual, ou seja, o valor que indica quanto a saída do neurônio atual afeta no neurônio da camada seguinte;

$\delta_j^{(l+1)}$: é o gradiente local para o neurônio j da camada $l+1$. Este gradiente é calculado da seguinte maneira:

$$\delta_j^{(l+1)} = \varepsilon_j^{(l+1)} f'\left(v_j^{(l+1)}\right), l = 1, 2, ..., L \tag{2.13}$$

Onde:

$\varepsilon_j^{(l+1)}$: erro do neurônio j da camada $l+1$;

$f'\left(v_j^{(l+1)}\right)$: derivada da função de ativação do neurônio j da camada $l+1$.

Vale notar que para a última camada o erro é calculado através da diferença entre o valor esperado e o obtido para a saída da rede. E para os neurônios das camadas ocultas, o cálculo do erro depende do valor do erro da camada seguinte e da derivada da função de ativação do neurônio, justificando a necessidade desta função ser diferenciável em todos os pontos.

Com todos os erros calculados, o valor usado para a atualização dos pesos pode ser calculado:

$$\Delta w_{ij}^l = \eta.\delta_i^l.y_j^{(l-1)}, i = 1, 2, ..., N^l; j = 1, 2, ..., N^(l-1) \tag{2.14}$$

$$\Delta b_i^l = \eta.\delta_i^l \tag{2.15}$$

Onde η é a taxa de aprendizado. Os pesos e *bias* são então atualizados:

$$w_{ij}^l(n+1) = w_{ij}^l(n) + \Delta w_{ij}^l(n) \tag{2.16}$$

Onde $w_{ij}^l(n)$ é o peso atual e $w_{ij}^l(n+1)$ é o valor do peso atualizado.

Assim que todos os pesos são atualizados, a RNA calcula as entradas novamente e todo o processo é repetido até que o valor do erro seja menor que um limite previamente estipulado.

Ao se analisar a maneira que o treinamento acontece, pode-se notar que o valor dos erros é propagado pela Rede Neural Artificial no sentido contrário da operação direta da RNA, justificando assim o nome do treinamento (31).

2.2 Implementação de Redes Neurais Artificiais

Além das características de topologia e tipo de treinamento da RNA, é importante decidir em que tecnologia a Rede Neural Artificial será desenvolvida, se em software ou hardware.

A implementação em software traz como vantagem uma grande facilidade e velocidade no desenvolvimento, já que várias ferramentas já estão disponíveis no mercado (18, 32), como por exemplo o software MATLAB que tem uma *Toolbox* para o desenvolvimento e treinamento de RNAs de várias topologias diferentes. A facilidade e rapidez no design de novas Redes Neurais Artificiais faz com que este tipo de implementação traga uma flexibilidade ao desenvolvimento, permitindo que RNAs de tipos e tamanhos diferentes possam ser implementadas no mesmo dispositivo (33). Porém, como se trata de um processador sequencial simulando o funcionamento de uma RNA, que é um sistema massivamente paralelo, a velocidade de operação é menor do que em um sistema totalmente paralelo em hardware (18, 33). A diferença de velocidade é mais notada quando grandes Redes Neurais Artificiais são necessárias (18) e também quando o sistema em que a RNA se encontra exige operação em tempo real. Em geral, quando não são necessárias altas velocidades de operação, o uso de implementações em software é interessante (4). Também as implementações em software são úteis para o teste e identificação de melhores topologias de RNAs para um dado problema para, em seguida, serem implementadas em hardware.

As implementações de RNAs em hardware, ao contrário das em software, aproveitam o inerente paralelismo da estrutura das Redes Neurais Artificiais, trazendo uma velocidade de processamento mais alta, que pode ser ordens de magnitudes maior do que a obtida em software (18, 32, 33, 34, 35, 36). Esta característica é importante quando a RNA será parte de um sistema que busca respostas em tempo real, ou quando a Rede Neural Artificial demanda alto número de neurônios e conexões (3, 15). Também, trocar o uso de um computador por um hardware específico pode reduzir o custo da Rede Neural Artificial (7, 33) e permitir aplicações especializadas como dispositivos dedicados mais baratos, dispositivos neuromórficos como retinas de silício, entre outros (4).

Uma desvantagem de se usar implementações em hardware de Redes Neurais Artificiais vem da dificuldade no desenvolvimento e modificação do circuito (15, 18) e também do tamanho que a Rede Neural Artificial pode chegar, já que grandes áreas de hardware podem ser necessárias devido à estrutura paralela (37, 38). Um dos grandes desafios do desenvolvimento de Redes Neurais Artificiais em hardware está na criação de sistemas que minimizem a área utilizada através de neurônios compactos e eficientes (39).

As características gerais de implementações em hardware e software são resumidas na Tabela 1.

Tabela 1 – Implementações em Software vs Implementações em Hardware

Característica	Software	Hardware
Velocidade	Baixa	Alta
Flexibilidade	Alta	Baixa
Facilidade de Design	Alta	Baixa
Processamento	Sequencial	Paralelo

Em geral, quando uma tarefa particular não demanda alta velocidade de processamento, é preferível implementar a Rede Neural Artificial em software, já que o desenvolvimento é mais rápido. Nos casos em que altas velocidades de processamento são requeridas, além de uma maior eficiência, implementações em hardware são escolhidas (3).

As implementações de RNAs em hardware se mostraram promissoras para aplicações especializadas, como processamento de imagens, síntese de fala, análise e reconhecimento de padrões, e dispositivos médicos implantáveis (4), motivo pelo qual elas foram escolhidas neste trabalho.

O desenvolvimento de RNAs em hardware ainda pode ser dividido em implementações analógicas e digitais.

2.2.1 Redes Neurais Artificiais Analógicas

Redes Neurais Artificiais implementadas em tecnologia analógica usam sinais de corrente ou tensão para representar seus sinais, tratando diretamente com o mundo real e sinais reais, sem a necessidade do uso de conversores, como é o caso de dispositivos digitais (4, 32, 40). Os sistemas analógicos trabalham com sinais contínuos, fazendo com que à primeira vista, aplicações analógicas sejam melhores opções para RNAs (18, 32).

O uso de dispositivos analógicos faz proveito de características não-lineares dos próprios dispositivos que podem ser usadas no design de Redes Neurais Artificiais (4, 37). Se os sinais forem representados por correntes, a soma pode ser feita através de nós no circuito, seguindo a Lei de Kirchhoff, e a multiplicação pode ser feita através de resistores adaptáveis. Assim, uma rede de resistores com valores específicos para cada peso necessário da Rede Neural Artificial pode simular a soma ponderada da entrada de cada neurônio (40). Funções não lineares complexas podem ser implementadas através de transistores e amplificadores operacionais (18, 32). Essa facilidade na realização das funções traz uma menor área gasta na implementação do circuito, além da realização de circuitos mais simples (4, 18, 32, 37). O consumo de potência também é menor se comparado a outros tipos de implementação (4, 18, 32).

A velocidade de processamento de RNAs em tecnologia analógica também é maior se comparada à digital, já que os circuitos são menores e mais simples (4, 15, 18, 32, 36)

Em geral, circuitos de Redes Neurais Artificiais analógicas são mais compactos e oferecem altas velocidades de processamento à baixas perdas por dissipação (4).

A desvantagem no uso de sistemas analógicos é a dificuldade de se implementar pesos adaptáveis para os neurônios. Ao se usar resistores como os pesos, sua variação fica inviável. Ou seja, as RNAs implementadas em tecnologia analógica são inflexíveis e o treinamento no próprio sistema em hardware é difícil de ser atingido (4, 18, 32, 37). Se, ao invés de resistores, outras técnicas forem usadas para permitir a mudança nos pesos, as variações no processo de produção dificultam o uso (4).

Outra característica inerente da Rede Neural Artificial que é prejudicada ao se produzir RNAs analógicas é a necessidade de um grande número de conexões entre os neurônios. Esta interconexão gasta muita área no silício. Uma solução possível é multiplexar as conexões, porém a complexidade do circuito aumenta muito (18).

Há também em circuitos analógicos uma grande susceptibilidade à ruídos e à variação dos parâmetros durante o processo de fabricação (4, 32, 41); devido a isso, a construção de sistemas precisos e funcionalmente idênticos é uma tarefa difícil (4, 37, 40).

2.2.2 Redes Neurais Artificiais Digitais

Ao contrário de sistemas implementados em tecnologia analógica, sistemas digitais têm grande confiabilidade já que os circuitos fabricados são funcionalmente idênticos. Os circuitos são, então, poderosos e maduros, pois as técnicas de fabricação usadas são bem conhecidas e confiáveis (4, 37). Outra vantagem sobre os sistemas analógicos é a imunidade à ruídos (32, 40).

Os sistemas e chips disponíveis no mercado são em sua maioria digitais e, como a Rede Neural Artificial é normalmente somente uma parte de um sistema mais complexo, implementá-las em tecnologia digital provê uma maior facilidade de integração com esses sistemas (18, 40). Também, poderosas ferramentas de design estão disponíveis no mercado, facilitando a implementação de sistemas digitais (4).

O uso da tecnologia digital também traz a possibilidade de o armazenamento de pesos ser feitos através de memórias voláteis ou não voláteis (4, 32). Isto permite a atualização de pesos, ou seja, o treinamento da RNA pode ser feito de modo online no mesmo chip, o que traz flexibilidade e maior poder à Rede Neural Artificial (4, 18, 32, 41).

A exatidão em Redes Neurais Artificiais digitais não é um fator limitante, como em sistemas analógicos, já que seu aumento só depende do tamanho da palavra usada pelos sinais e operações. Esta característica é importante durante o treinamento da RNA, já que altas precisões permitem que o treinamento seja mais eficaz. Embora a exatidão possa ser mudada apenas alterando o tamanho da palavra que a Rede Neural Artificial trabalha, o aumento no número de bits das palavras vem a custo de área (4, 32, 34, 40)

e desta forma, circuitos digitais são relativamente grandes se comparados aos analógicos (4, 18, 35, 36). Também, a implementação de funções não lineares é complexa e custosa em termos de área e velocidade (32).

O multiplicador é o bloco que mais se repete em Redes Neurais Artificiais e sua implementação em sistemas digitais necessita de grande área. Também, a velocidade de operação é menor que a de sistemas analógicos, sendo um ponto de atenção ao se desenvolver RNAs digitais (18, 32). É então importante, principalmente em implementações totalmente paralelas, encontrar implementações de multiplicação eficientes em área, pois o número de multiplicações em uma Rede Neural Artificial cresce de acordo com o número de entradas da rede e se torna arbitrariamente grande (41).

Em geral, sistemas digitais trazem uma maior exatidão, facilidade de implementação e confiabilidade ao custo de velocidade, eficiência e dissipação de potência (18, 32).

Sistemas digitais podem ser implementados em FPGA, para um desenvolvimento rápido e menos custoso. Os detalhes são vistos a seguir.

2.2.3 Redes Neurais Artificiais em FPGAs

FPGAs ou *Field Programmable Gate Arrays* são dispositivos compostos por elementos lógicos reprogramáveis, conectados entre si via conexões que podem ser ajustadas, formando uma estrutura regular e hierárquica (18, 38, 42, 43). Estes blocos e conexões estão em total controle do usuário e podem ser programados da maneira que for necessário para atender a aplicações específicas (18, 43). Em geral, FPGAs provêm circuito customizado através de linguagens de descrição de hardware para aplicações de design específico (15, 44).

A possibilidade de criar hardware que pode ser rapidamente customizado é uma das características mais importantes das FPGAs. Isto traz a possibilidade de se criar um sistema em hardware com flexibilidade semelhante ao software. Como as RNAs tem a habilidade de aprender com o ambiente e se adaptar constantemente e as FPGAs permitem a criação de sistemas de alta performance com flexibilidade, FPGAs são adequadas para esse tipo de sistema (12, 15, 16, 18, 19, 21, 38, 42, 43, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54).

Apesar de permitir a flexibilidade do software, implementações em FPGAs ainda mantém o paralelismo característico do hardware, o que é bastante adequado para RNAs (12, 15, 18, 19, 38, 49, 51). Este paralelismo permite implementações de alta velocidade, alta performance e alto poder de processamento, ou seja, a flexibilidade da FPGA vem sem grande custo no desempenho (15, 18, 21, 45, 49). Porém, como as FPGAs tem um tamanho e uma quantidade de blocos lógicos limitados, há uma limitação no tamanho que a Rede Neural Artificial pode atingir, já que cada neurônio, se implementado de maneira totalmente paralela, precisa de uma grande quantidade de multiplicações e de memória

para guardar seus pesos (18, 49, 53, 55). Como se sabe, as operações de multiplicação demandam circuitos complexos, o que faz com que a implementação de RNAs em FPGAs com grande número de neurônios seja difícil (56). Para contornar este problema, algumas implementações fazem operações de maneira sequencial, o que diminui a velocidade de operação (18, 19). Outra solução é implementar RNAs com uma menor quantidade de bits, o que traz maiores velocidades de processamento e circuitos menores, porém, os erros de quantização são maiores. Além disso, se o número de bits for muito baixo, a Rede Neural Artificial implementada pode não ser ideal para todas as aplicações (18, 43).

O custo de desenvolvimento de RNAs em FPGAs é menor, já que é possível implementar mais de uma arquitetura da Rede Neural Artificial em um mesmo hardware (12, 38, 43, 47, 48, 55, 51). Em resumo, as FPGAs são uma maneira de se ter um hardware barato e de baixo custo, o que as torna sistemas acessíveis (48, 57).

Existem ferramentas de desenvolvimento para FPGAs poderosas e de simples utilização, o que faz o design de novas soluções e a modificação de designs já existentes nessa plataforma mais fácil e rápido, semelhante ao desenvolvimento de soluções em software (15, 18, 38, 48, 53, 57, 58). O ciclo de desenvolvimento de novos projetos é menor e de baixo custo devido ao poder de programação da FPGA, característica importante quando se tem restrições no tempo de projeto (15, 42, 49).

A criação de novos protótipos precisa de vários ciclos de atualização até que o circuito final seja alcançado. Se este desenvolvimento for feito em circuitos integrados, o custo e o tempo seriam muito altos. Assim, o uso de FPGAs facilita a criação de novos protótipos, devido à sua flexibilidade. Ela permite uma simulação direta no hardware e uma melhor otimização da arquitetura da RNA a ser desenvolvida (18, 38, 42, 52, 57).

A Tabela 2 resume os pontos forte e fracos das tecnologias a se implementar RNAs (16, 18).

Tabela 2 – Comparação entre implementações Analógicas, Digitais e em FPGA

Característica	Analógica	Digital	FPGA
Área	Muito Baixa	Baixa	Baixa (Maior que Digital)
Velocidade	Muito Alta	Alta	Alta (Menor que Digital)
Custo	Alto	Alto	Baixo
Tempo de Design	Alto	Alto	Menor que Analógico e Digital
Confiabilidade	Baixa	Alta	Muito Alta

Pela análise da Tabela 2, ao se iniciar o desenvolvimento de RNAs é interessante o uso de FPGAs devido ao baixo custo e menor tempo de design, e também à alta confiabilidade do circuito. Por este motivo, a Rede Neural Artificial deste trabalho será implementada em FPGA.

2.3 Redes Neurais Artificiais Reconfiguráveis em Hardware

Em algumas situações em que RNAs são aplicadas, o ambiente tem grandes variações, fazendo necessárias mudanças na arquitetura da Rede Neural Artificial mesmo após esta ter sido fabricada. Robótica, missões espaciais, comunicação móvel, automação e sistemas médicos são alguns exemplos de aplicações com ambiente variável. É necessário então criar um sistema que seja capaz de se adaptar às mudanças de requisitos sem mudança no hardware, já que este está fixo e o acesso para trocas nem sempre é fácil (45, 54, 59). Nesses casos, o mesmo hardware seria usado para diferentes tarefas e o usuário seria capaz de decidir qual arquitetura a RNA deve executar (59, 60). Esse tipo de aplicação é chamado de Rede Neural Artificial Reconfigurável.

Em geral, a reconfiguração da RNA traz a possibilidade de se ter soluções generalizadas em hardware, ou seja, diferentes arquiteturas para diferentes problemas com um hardware fixo, permitindo que a Rede Neural Artificial se adapte às mudanças de requisitos (15, 54, 61, 62). Esta adaptação trazida pela reconfiguração pode ser feita a tempo de execução, de maneira rápida (45, 61), o que é importante para dispositivos embarcados em locais de difícil acesso (63, 64). Além da flexibilidade no circuito, esta implementação traz uma economia de área e de consumo de potência. A diminuição da área gasta pode permitir um aumento da resolução do sistema. Ainda, há uma facilidade nos testes que a Rede Neural Artificial precisa fazer, já que estes podem ser feitos diretamente no hardware, permitindo resultados reais (45, 61). Entretanto, a reconfiguração traz um certo impacto no tempo de execução, já que um tempo deve ser gasto para fazer mudanças no circuito. Dependendo da aplicação, este impacto no tempo pode ser desconsiderado (62).

Como já foi dito anteriormente, a MLP é uma das topologias de RNAs mais usadas atualmente (2). Seu funcionamento é organizado em camadas e a operação dessas camadas é sequencial, ou seja, apesar dos neurônios dentro de uma camada terem a operação paralela, a execução de uma camada depende das saídas das camadas anteriores. Este tipo de comportamento traz uma possibilidade de implementação de Redes Neurais Artificiais Reconfiguráveis, através da multiplexação das camadas, a fim de economizar área do circuito. Se somente uma camada for implementada e sua entrada for realimentada pela sua saída, RNAs do tipo MLP de arquiteturas variadas podem ser implementadas em um mesmo hardware (12, 65). Este tipo de implementação trabalha com a virtualização da Rede Neural Artificial, ou seja, somente uma camada física existe, enquanto as outras camadas são virtuais (21, 40). Esta técnica diminui a área gasta no circuito, já que menos neurônios precisam ser implementados, e também provê uma flexibilidade para o hardware a tempo de execução (21, 40, 60, 63).

É importante nesse ponto definir a diferença entre a adaptação da RNA através de treinamento, o desenvolvimento de Redes Neurais Artificiais reprogramáveis em FPGA e a criação de Redes Neurais Artificiais Reconfiguráveis. A adaptação que a Rede Neural

Artificial tem durante o treinamento modifica somente o valor dos pesos e *bias* referentes às conexões de cada neurônio. A maneira com que estas ligações estão definidas não muda, ou seja, a arquitetura da Rede Neural Artificial se mantém fixa. No caso do desenvolvimento em FPGAs, há a possibilidade de reprogramação, ou seja, mudar a aplicação que está sendo executada na FPGA. Porém, esta reprogramação necessita de uma nova compilação do circuito através de um diferente código e um novo sistema é executado pela FPGA. Esta reprogramação modifica o hardware, não sendo possível em sistemas fixos, como circuitos integrados. Já a criação de RNAs Reconfiguráveis aqui discutida trata da modificação da arquitetura da Rede Neural Artificial sem a mudança no hardware, ou seja, há uma mudança na maneira que os neurônios estão conectados entre si e uma nova arquitetura de RNA será executada enquanto o hardware se mantém fixo (61). Como a arquitetura foi modificada, um novo conjunto de pesos e *bias* é necessário, fazendo necessário um novo treinamento. A vantagem desta abordagem é trazer flexibilidade à RNA sem a necessidade do desenvolvimento de um novo circuito e sem depender de uma FPGA, mantendo o hardware fixo.

Algumas arquiteturas reconfiguráveis já foram propostas na literatura e três delas são discutidas a seguir.

2.3.1 Abordagem 1 - Multiplexação de Camadas

A primeira abordagem (12) tem duas soluções para a implementação de Redes Neurais Artificiais Reconfiguráveis: uma delas eficiente em termos de área e outra eficiente em termos de tempo de processamento. Ambas têm em comum o fato de usarem a multiplexação das camadas, ou seja, somente uma camada física é implementada e reutilizada para a execução completa da RNA. A abordagem eficiente em termos de área é mostrada na Figura 14.

Pode-se observar que uma camada com n neurônios é implementada. A saída desta camada é colocada em um multiplexador que realimenta a entrada da camada. Além dos neurônios e do multiplexador, esta implementação também contém memórias para guardar as entradas e pesos da RNA, além de uma memória que guarda as instruções que definem a arquitetura da Rede Neural Artificial. Também há um bloco responsável por executar a função de ativação através de uma aproximação, que é dividido por toda a RNA. Uma unidade de controle é responsável por controlar toda a execução de acordo com as instruções vindas do usuário. Nesta abordagem o usuário tem que definir o passo a passo da execução da rede.

Esta implementação é dita econômica em termos de área pois a operação da função de ativação é dividida por todos os neurônios. Porém, como a função de ativação de cada neurônio deve ser executada de maneira sequencial, o tempo para que a saída da RNA seja calculada é maior.

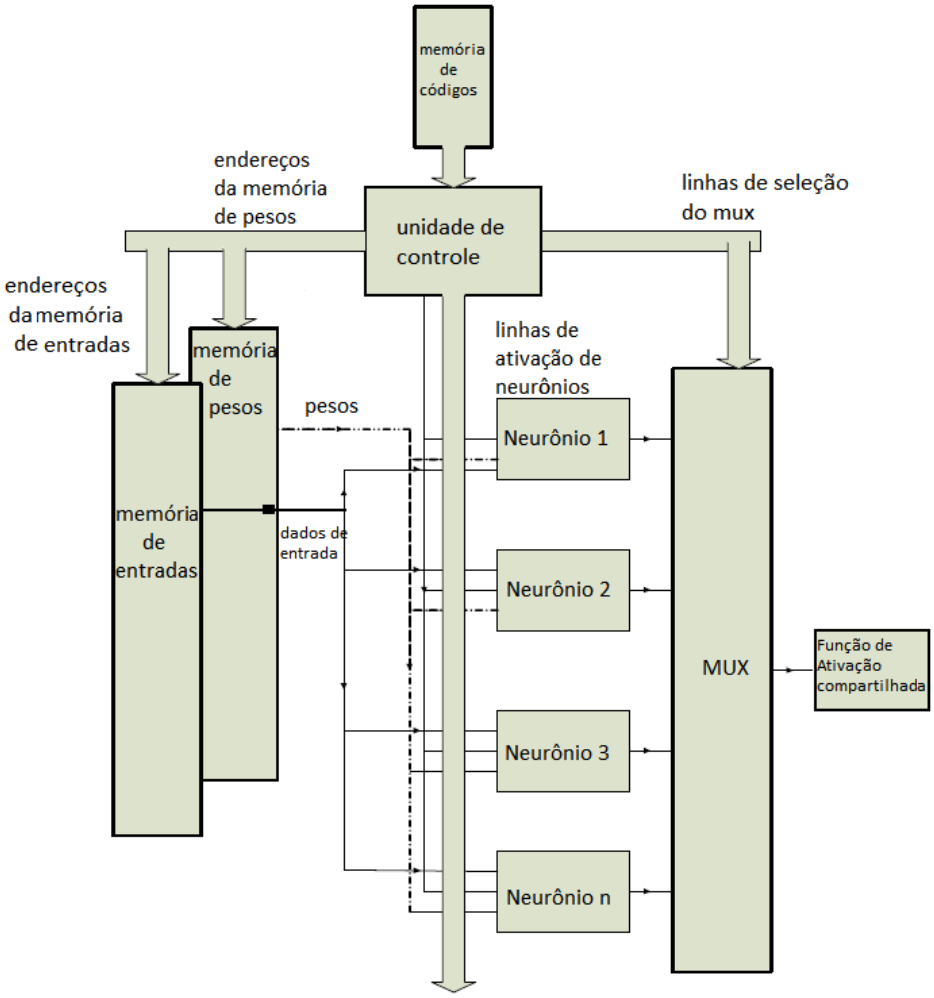

Figura 14 – RNA Reconfigurável com multiplexação de camada - abordagem eficiente em área (12)

A segunda técnica nesta abordagem é econômica em termos de tempo de execução. Sua arquitetura pode ser vista na Figura 15.

Novamente, existe somente uma camada de neurônios. As memórias de pesos e de instruções funcionam da mesma maneira que a primeira abordagem. A unidade de controle garante a correta operação da RNA. A diferença nesta implementação cada neurônio tem sua própria função de ativação, permitindo que as saídas de todos eles sejam calculados de forma paralela, diminuindo o tempo de execução. A memória de dados e o multiplexador são responsáveis por realimentar a saída na entrada da camada. Como no primeiro caso, o usuário é responsável por dar o passo a passo da execução.

2.3.2 Abordagem 2 - Multiplexação de Camadas com representação numérica baseada em frações

Uma segunda abordagem para a implementação de RNAs Reconfiguráveis usa uma representação numérica baseada em frações a fim de diminuir a área da implementação e

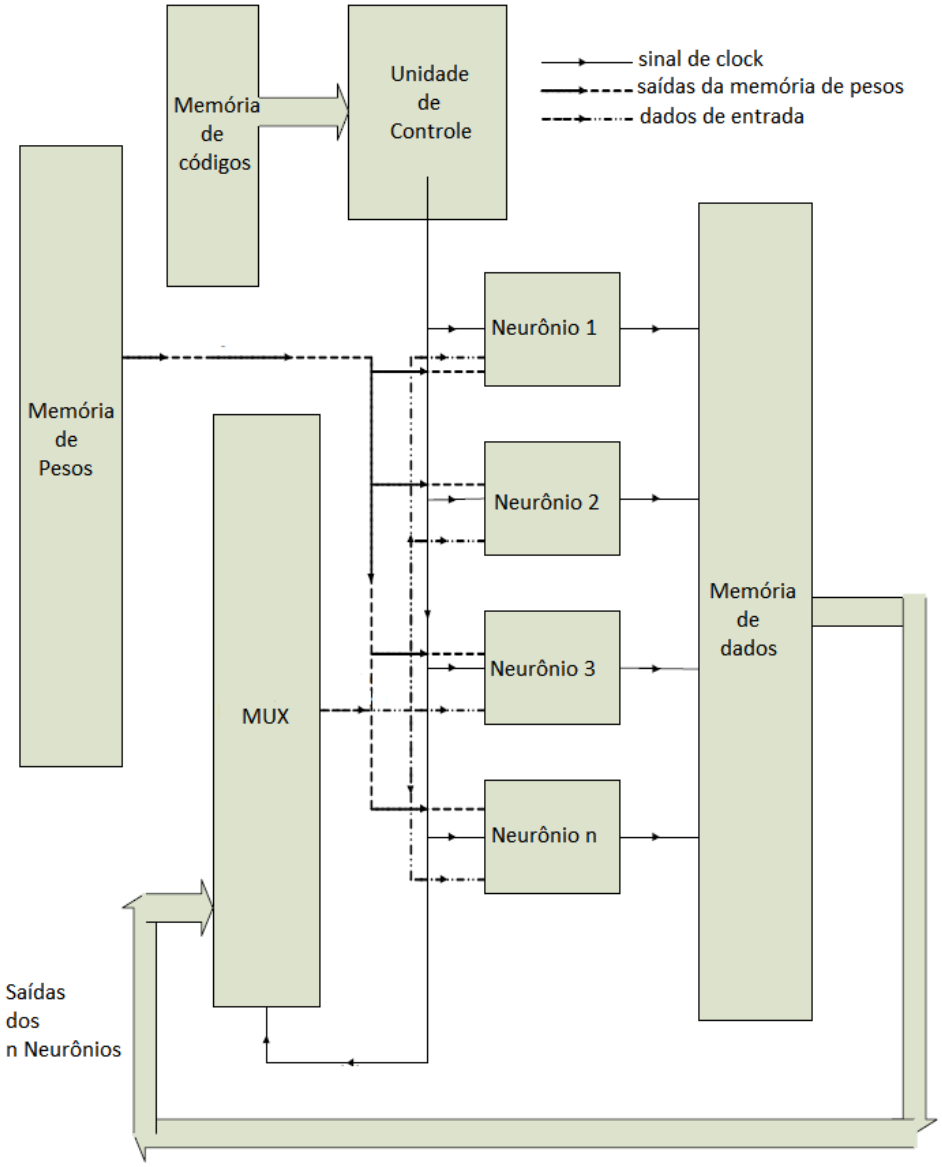

Figura 15 – RNA Reconfigurável com multiplexação de camada - abordagem eficiente em tempo de execução (12)

manter uma alta precisão (62). Esta abordagem também explora o conceito de multiplexação e virtualização de camadas, ou seja, somente uma camada física é implementada e reusada durante a execução da Rede Neural Artificial. A estrutura geral da parte em hardware da arquitetura pode ser vista na Figura 16.

Esta arquitetura é formada por dois grandes sistemas, o LCS (*Load and Control System*), responsável pelas memórias de entradas, de pesos e da função de ativação, além de ser responsável por controlar a execução da RNA. E o ANNCH (*Artificial Neural Network Computing Hardware*), que é responsável pela execução dos neurônios da camada. Nesta implementação, apesar dos neurônios operarem de forma paralela dentro da camada, sua operação interna é sequencial, ou seja, somente um multiplicador é responsável por executar todos os produtos necessários na soma ponderada.

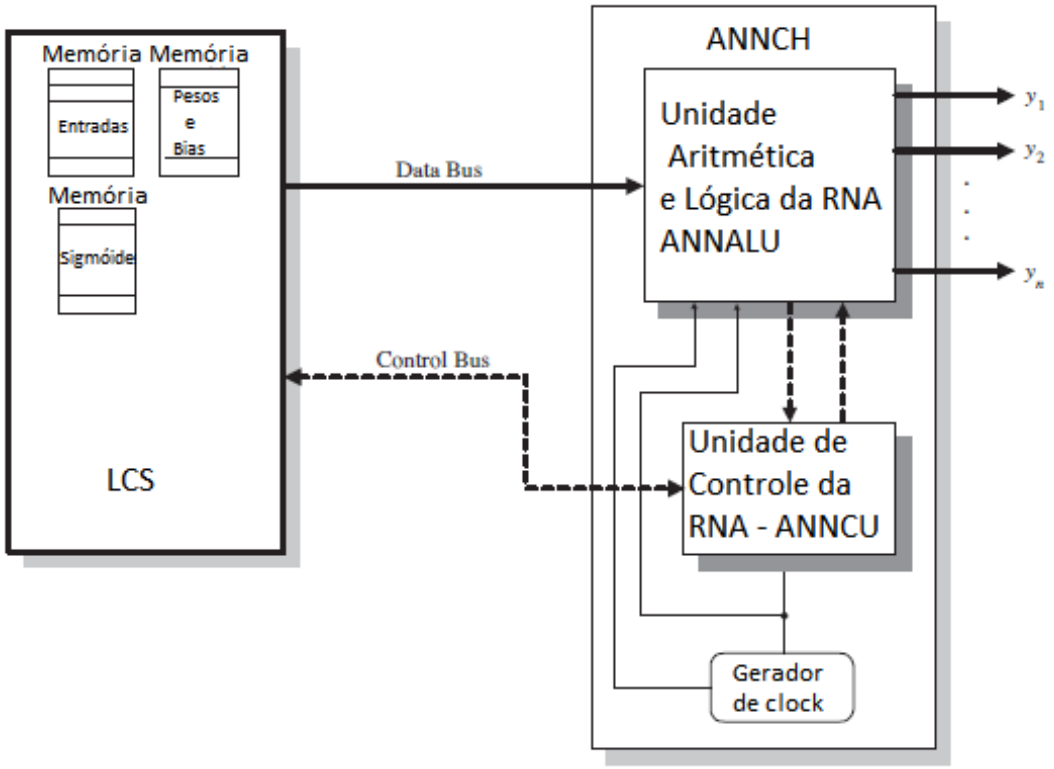

Figura 16 – RNA Reconfigurável baseada em frações (62)

Esta implementação também contém um sistema em software responsável por controlar a reconfiguração da RNA, como os parâmetros que definem a quantidade de entrada, a quantidade de camadas, de neurônios por camada e se há ou não *bias* presente em algum neurônio da camada. Apesar da diminuição na área obtida através da multiplexação da camada, esta abordagem tem uma maior complexidade devido à notação numérica escolhida, o que traz um aumento na área. Também, neste trabalho, um controle em software é necessário para a execução da RNA, o que pode aumentar o custo do sistema.

2.3.3 Abordagem 3 - Multiplexação de Camadas com RDP

Esta abordagem para Redes Neurais Artificiais Reconfiguráveis é obtida ao se usar tanto a multiplexação de camada quanto uma técnica possível em FPGAs chamada de Reconfiguração Parcial Dinâmica (RDP) (45). A RDP atualiza as partes que serão usadas durante a execução de um sistema a tempo de execução, ou seja, o hardware da FPGA muda enquanto o sistema opera. Na arquitetura de RNA aqui discutida, além de somente uma camada física do neurônio ser implementada, a inserção ou remoção de novos neurônios nessa camada é feita através de RDP. Ou seja, a quantidade de neurônios implementada pela RNA varia durante a execução. Nesta arquitetura há um sistema em software responsável por controlar a execução geral da RNA.

A parte implementada em FPGA é dividida em um controlador e a camada. O controlador é responsável por receber as instruções do sistema em software e a camada

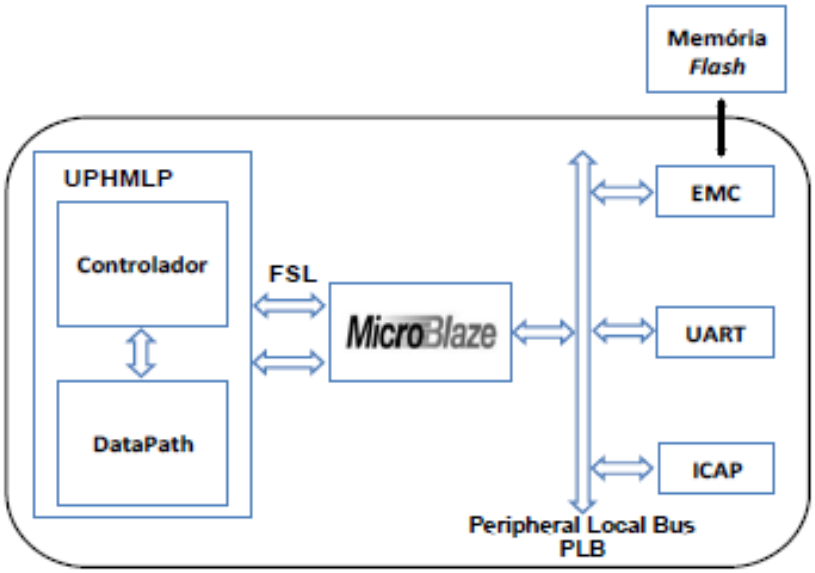

Figura 17 – RNA Reconfigurável com RDP (45)

é um sistema adaptativo responsável por executar os neurônios da RNA. A camada inicialmente é formada por 5 neurônios mas pode aumentar o número de neurônios a fim de conseguir executar maiores arquiteturas de Redes Neurais Artificiais. O controlador gerencia a necessidade de aumentar ou diminuir o número de neurônios da camada. Nesta abordagem, além da arquitetura da RNA e quantidade de neurônios, a função de ativação executada pelos neurônios pode ser escolhida entre a Tangente Hiperbólica e a Sigmóide Logística. Uma desvantagem desta implementação é a necessidade de usar a FPGA, já que somente ela permite RDP.

Em geral, a habilidade da RNA de mudar a quantidade de entradas, quantidade de camadas, quantidade de neurônios por camada e a função de ativação executada por cada neurônio permite a criação de hardware de propósito geral ou adaptável.

2.4 Implementação de Funções de Ativação em Sistemas Digitais

A Função de Ativação (FA) é parte importante no desenvolvimento de Redes Neurais Artificiais em hardware. Tomando as 5 funções de ativação mais usadas em redes MLP, e já mostradas na Seção 2.1.1, nota-se que três delas são funções lineares e duas são funções não lineares. As funções lineares são de fácil implementação em FPGAs e não necessitam de grandes análises. O detalhamento de sua implementação está na Seção 3.4.2.2.

Já a implementação de funções não lineares é uma das partes mais complexas no desenvolvimento de Redes Neurais Artificiais em FPGAs (66, 67, 68). Isto devido ao fato

de que, ao exemplo da função Sigmóide Logística e da Tangente Hiperbólica, estas funções são formadas por séries exponenciais infinitas (38, 66, 28), levando a uma grande área de hardware necessária para sua implementação (16, 26, 67, 68). Para reduzir o gasto em área das implementações das funções, aproximações foram propostas na literatura (26, 67, 68). Estas aproximações vêm a custo de exatidão das funções e deve-se, então, analisar a relação entre a área da aproximação, sua performance e sua exatidão, a fim de se ter melhores implementações (26).

A perda na exatidão das aproximações de Funções de Ativação impacta a execução da Rede Neural Artificial tanto quanto a quantidade de bits usada por seus dados (67). Redes Neurais Artificiais com FAs de baixa exatidão podem trazer grandes desvios em seus resultados, necessitando de mais neurônios para a execução de uma mesma tarefa (67). Também, Redes Neurais Artificiais com FAs mais precisas tem aprendizado mais rápido e maior capacidade de generalização (66).

Melhores precisões das aproximações vêm ao custo de maiores áreas de implementação e diminuição da velocidade de operação (66, 69), tornando a busca por uma aproximação com menores áreas, altas velocidades e altas precisões uma peça chave no desenvolvimento de Redes Neurais Artificiais em hardware. A escolha de qual aproximação será usada deve ser feita levando-se em conta o objetivo da Rede Neural Artificial e da tecnologia em que esta será implementada (66).

Por serem muito parecidas, as funções Tangente Hiperbólica e Sigmóide Logística serão tratadas somente como função sigmóide (na forma de "S"), facilitando a busca e análise de aproximações e implementações destas funções para FPGAs.

Várias aproximações de funções sigmóide são apresentadas na literatura. Essas aproximações podem ser divididas em algumas categorias para facilitar sua análise.

2.4.1 Aproximações baseadas em LUT

Neste tipo de aproximação, o intervalo de entrada é dividido em subintervalos e cada um deles é aproximado por um valor e guardado na LUT (*Look-Up Table*), que nada mais que é que uma pequena memória disponível na FPGA (66). Embora simples de ser implementada e capaz de atingir altas velocidades de operação, a aproximação baseada em LUT requer grande área do hardware quando se necessita de altas precisões (26, 38, 39, 28, 70, 71). Também, este tipo de aproximação se torna impraticável em Redes Neurais Artificiais grandes e totalmente paralelas, já que seria necessária uma LUT para cada neurônio (70). Para contornar esse problema, somente uma LUT poderia ser compartilhada entre todos os neurônios, trazendo um maior tempo de execução e perda no paralelismo da Rede Neural Artificial (42, 72).

2.4.2 Aproximações de Ordem Superior

Esta aproximação usa equações de segundo grau ou grau superior para tentar simular o comportamento da função sigmóide (26, 70). Em geral, são necessários dois multiplicadores ou mais para a implementação, aumentando tanto a área do circuito como o tempo de execução. Uma vantagem é a alta exatidão atingida pela aproximação (26). Uma maneira de diminuir a quantidade de multiplicadores necessários é compartilhar o multiplicador e executar a aproximação de forma sequencial, o que traz ainda maiores tempos de execução, fazendo esta aproximação impraticável para RNAs de tempo real (42).

2.4.3 Aproximações por Partes - Linear

Este tipo de aproximação é uma das maneiras mais comuns de se implementar funções de ativação em Redes Neurais Artificiais (71). Em geral, nesta aproximação a função é dividida em intervalos e cada um deles é aproximado através de uma função linear (66). Assim, é necessário ao menos um multiplicador para a implementação, o que aumenta sua área em hardware (39, 66). Uma maneira de evitar esse aumento é escolher os segmentos de maneira a evitar a necessidade de multiplicadores, através de deslocadores, por exemplo, levando a estruturas menores (71). De modo geral, a complexidade do sistema proposto é baixa (42).

A quantidade de segmentos usados para aproximar a função pode variar, mudando assim a exatidão que a aproximação atinge (26). A escolha dos segmentos deve ser feita de modo a minimizar o erro, tempo de processamento e área (71). De modo geral, quanto mais segmentos são usados, melhores são os valores de erros médio e máximo (67) e, em geral, estes erros também são baixos (42). Também, a velocidade de operação é baixa, e o consumo de área, quando o multiplicador não é necessário, também é baixo (71).

Alguns exemplos de implementações de Aproximação por Partes Linear podem ser vistos a seguir:

- *A-Law-Based*: Nesta abordagem, a curva da função sigmóide é dividida em 7 segmentos e o gradiente de cada uma das retas que aproximam esses segmentos é expressado como uma potência de 2, podendo assim, substituir as multiplicações por deslocamentos (42, 73).
- *Allipi e Storti-Gajani*: Nesta aproximação, os pontos de separação da curva são escolhidos como inteiros e as saídas são valores em potência de 2. A aproximação também só usa adição e potência de 2, permitindo o uso de deslocadores, fazendo-a adequada para implementações digitais (42, 72).

- *PLAN*: A aproximação PLAN (ou *Piecewise Linear Approximation of a Nonlinear Function*) usa portas lógicas para transformar diretamente cada intervalo da entrada na saída. Esta aproximação usa a simetria das funções sigmóide, ou seja, o cálculo é feito somente no valor absoluto e depois é ajustado caso a entrada seja negativa. Inicialmente, uma equação com índices em potência de 2 é escolhida para aproximar cada intervalo. Em seguida, estas equações são simplificadas de modo a serem somente deslocamentos e somas. Então, a operação é analisada como uma conversão direta de bits (42, 70). Esta mudança reduz a área da implementação e aumenta a velocidade de processamento (21).

- *CRI*: Também chamada de *Centred Recursive Interpolation*, essa aproximação usa um algoritmo recursivo para aproximar a função sigmóide. Esta técnica pode ser usada com segmentos lineares ou não. Neste caso, o método melhora a exatidão da aproximação através de um método recursivo de maneira que o número de segmentos lineares aumenta exponencialmente a cada rodada. A sigmóide é inicialmente dividida em três segmentos e, em seguida, o algoritmo CRI é usado para melhorar a aproximação. Como este método é recursivo, sua execução é lenta (42, 74, 67).

2.4.4 Aproximações por Partes - Não Linear

Este tipo de aproximação é similar ao Por Partes Linear, mas os segmentos em que a função sigmóide é separada são aproximados por funções de segunda ordem ou de ordens superiores (66). Uma desvantagem clara é a necessidade de multiplicadores (42). Esta técnica atinge excelentes precisões, a custo da velocidade de operação, já que operações de multiplicação são lentas (71).

Algumas soluções usam funções de segundo grau com adaptações para reduzir o número de multiplicadores necessários para um (75, 76), diminuindo a área usada no circuito.

2.4.5 Aproximações Combinacionais

A aproximação Combinacional mapeia os bits da saída diretamente através dos bits de entrada. Este método tem a clara vantagem de só usar circuitos combinacionais e não usar nenhum multiplicador ou operador aritmético (66).

Esta abordagem se mostra satisfatória quando tanto a entrada quanto a saída têm poucos bits. A relação entre a entrada e a saída é analisada na forma de uma função booleana que é, então, implementada através da soma de produtos, através de portas E e portas OU. A relação é feita de forma a minimizar o erro da aproximação (42).

2.4.6 Aproximações Híbridas

Métodos híbridos combinam os métodos acima já citados (66). Eles tipicamente atingem boas performances com pequena área e pouco tempo de execução, já que eles usam as vantagens dos métodos já vistos (71).

A Tabela 3 resume as características de cada tipo de implementação.

Tabela 3 – Comparação entre aproximações de Funções de Ativação

Aproximação	Área	Complexidade de Design	Velocidade	Presença de Multiplicador	exatidão
LUT	Alta	Simples	Alta	Não	A custo de área
Ordem Superior	Alta	Complexo	Baixa	Sim	Alta
Por Partes Linear	Baixa	Simples	Baixa	Não	Alta
Por Partes Não Linear	Alta	Simples	Baixa	Sim	Alta
Combinacional	Baixa	Simples	Alta	Não	Alta
Híbrida	Baixa	Complexo	Alta	-	Alta

Como o trabalho aqui desenvolvido procura não depender da arquitetura da FPGA, as técnicas de aproximação que usam multiplicadores foram descartadas. A aproximação através de LUT se mostra difícil para grandes sistemas, já que a área pode ser tornar muito grande. Sobram assim, as aproximações por Partes Linear e Combinacional. A Combinacional, como já foi dito, é usada quando tanto a entrada quanto a saída têm poucos bits, o que pode não ocorrer em alguns sistemas de RNAs. Assim, a técnica por Partes Linear foi escolhida. As propostas da literatura desta técnica foram comparadas em termos de área, tempo de execução e erros atingidos (42). Os resultados são mostrados na Tabela 4.

Tabela 4 – Comparação entre aproximações Por Partes - Linear (42)

Aproximação	Erro Máximo	Erro Médio	Área (Elementos Lógicos)	Frequência Máxima
PLAN	1,89%	0,59%	39	75,8 MHz
Alippi e Storti-Gajani	1,89%	0,87%	36	64,2 MHz
A-law based	4,90%	2,47%	36	58,6 MHz
CRI	2,06%	0,85%	65	8,7 MHz

Na Tabela 4 pode-se observar que a aproximação do tipo PLAN tem o menor erro médio, enquanto atinge a maior frequência de operação, enquanto a quantidade de elementos lógicos usados é semelhante à usada em outras aproximações, fazendo com que seja a melhor opção para este trabalho. Mais detalhes de sua implementação serão dados na Seção 3.4.2.2.

2.5 Multiplicadores em Sistemas Digitais

O multiplicador é um dos pontos-chave no desenvolvimento de RNAs, isto porque ele é o bloco que se repete mais vezes na Rede Neural Artificial, já que cada entrada necessita de um multiplicador para que o produto entre seu valor e o seu peso associado seja calculado.

Tomando um desenvolvimento em um FPGA, seria natural usar o multiplicador nela disponível. Porém, o uso do multiplicador embarcado faz com que a Rede Neural Artificial desenvolvida fique dependente do modelo da FPGA: mudanças na FPGA podem trazer mudanças nos resultados da RNA. Além disso, se a Rede Neural Artificial tiver intenção de ser implementada em Circuitos Integrados (CIs), o multiplicador disponível na FPGA não poderá ser usado, já que seu modelo é proprietário. Para contornar esse problema e permitir futuras implementações em CIs, optou-se neste trabalho por buscar topologias de multiplicadores digitais para serem usados nos neurônios.

Vários multiplicadores já foram propostos na literatura, entre eles vale ressaltar os multiplicadores Array, Booth, Wallace Tree e Vedic (38, 77, 78, 79, 80, 81). Os detalhes de cada uma destas técnicas podem ser vistos a seguir.

2.5.1 Multiplicador Array

O multiplicador Array baseia seu funcionamento em deslocamentos e adições (77) e é um dos multiplicadores mais conhecidos e simples (82). Um exemplo se seu funcionamento pode ser visto na Figura 18 a seguir:

				A3 x B3	A2 B2	A1 B1	A0 B0	
			c B1A3	B0A3 B1A2	B0A2 B1A1	B0A1 B1A0	B0A0	
		c B2A3	soma B2A2	soma B2A1	soma B2A0	soma		Somas Parciais
	c B3A3	soma B3A2	soma B3A1	soma B3A0	soma			
c								
S7	S6	S5	S4	S3	S2	S1	S0	

Figura 18 – Funcionamento do multiplicador Array

Como revela a Figura 18, o funcionamento do Array é baseado em multiplicar o primeiro bit do multiplicador por todos os bits do multiplicando, obtendo produtos parciais. Esses produtos são então deslocados e somados com o resultado dos produtos parciais do segundo bit do multiplicador por todos os bits do multiplicando, assim por diante até se obter o resultado final.

O multiplicador Array pode ser obtido através de um circuito combinacional, onde todos os produtos dos bits ficam disponíveis ao mesmo tempo, tornando-o, a princípio, uma maneira rápida de executar a multiplicação (38, 83). Porém, o atraso na execução aumenta à medida que as palavras a serem multiplicadas aumentam seu número de bits (77), já que a quantidade de produtos parciais e adições aumenta significativamente (84).

Algumas técnicas já foram propostas na literatura para melhorar a velocidade de operação do multiplicador Array. Além disso, esta implementação traz grandes consumos de potência, devido à sua arquitetura regular (77).

2.5.2 Multiplicador Booth

O multiplicador Booth foi proposto a fim de diminuir a quantidade de produtos parciais gerados para adição quando dois números são multiplicados (38, 78), reduzindo assim, a área do circuito. Uma vantagem deste método é a possibilidade de se trabalhar diretamente com números em complemento de 2 (85).

Esta abordagem é baseada no fato de que uma sequência de 0's no multiplicador não requer adição, somente deslocamento. A sequência de 1's do bit 2^k ao bit 2^m pode ser tratada como $2^{k+1} - 2^m$ (86). O fluxo de funcionamento do multiplicador Booth é visto na Figura 19:

Este multiplicador é formado por um somador de n bits, um controlador de lógica e registradores. Inicialmente, o multiplicando é colocado no registrador B e o multiplicador no registrador Q. Os registradores Q_{-1}, de 1 bit e A, de n bits são zerados. O somador recebe dados do multiplicando (registrador B) e do registrador A. Seu resultado é colocado novamente no registrador A. Este somador tem um *flag* que indica se suas entradas devem ser somadas ou subtraídas. O mesmo sinal garante que o complemento de 2 do multiplicando seja tomado quando se trata de uma subtração (86). Assim, de acordo com um controle feito pelo Multiplicador (Registrador Q) e através de somas sucessivas entre o Multiplicando (Registrador B) e o Registrador A, pode-se chegar ao produto correto.

O bloco de controle de lógica analisa 2 bits: um deles é o bit menos significativo do multiplicador e o último deles vem do registrador Q_{-1}. Dependendo do valor destes bits, uma ação será tomada (86). A Tabela 5 que define essa ação é mostrada a seguir:

Tabela 5 – Codificação em 2 bits do Multiplicador Booth

Codificação dos 2 bits	Operação a ser feita
00	Não há soma
01	O multiplicando é somado
10	O multiplicando é subtraído
11	Não há soma

Após essa operação, os registradores são deslocados para a direita e um novo ciclo

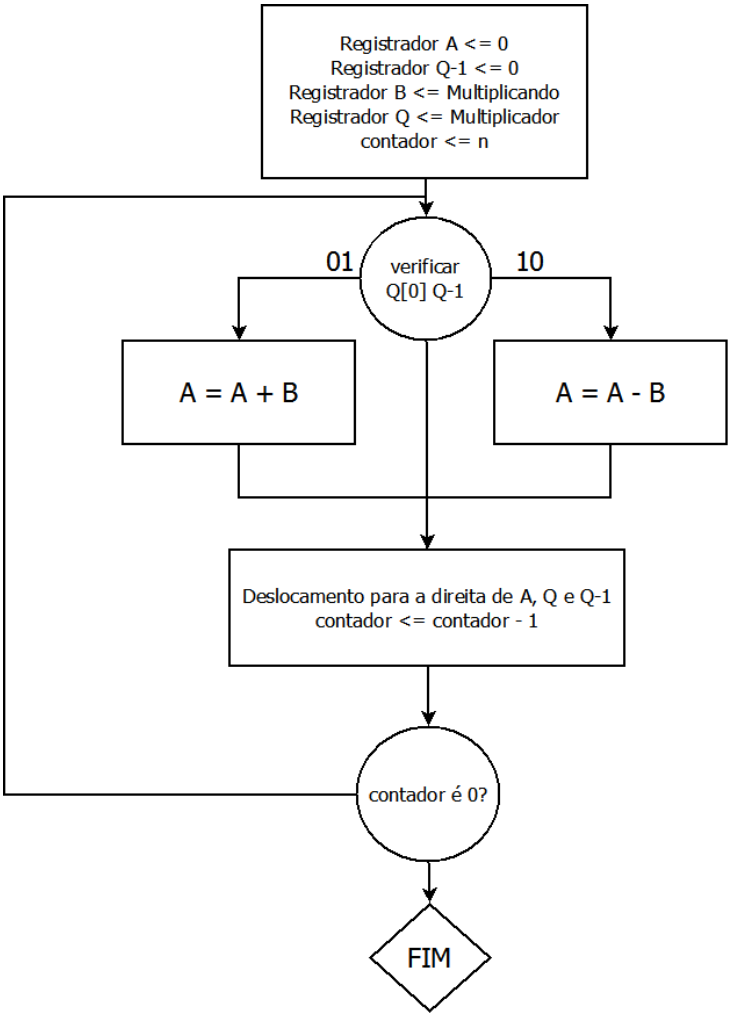

Figura 19 – Funcionamento do multiplicador Booth

da multiplicação se inicia. O deslocamento no registrador A é feito de maneira aritmética, para garantir que o sinal não se perca (86).

O multiplicador Booth foi modificado a fim de melhorar sua performance, ou seja, diminuir o tempo de execução (82). Na versão modificada, ao invés do controlador de lógica analisar 2 bits, são analisados 3 bits: 2 vindos do registrador Q e um do registrador Q_{-1} (82). A Tabela 6 de análise e operação a ser feita da nova abordagem é mostrada:

Tabela 6 – Codificação em 3 bits do Multiplicador Booth

Codificação dos 2 bits	Operação a ser feita
000	Não há soma
001	O multiplicando é somado
010	O multiplicando é somado
011	O multiplicando é multiplicado por 2 e então somado
100	O multiplicando é multiplicado por 2 e então subtraído
101	O multiplicando é subtraído
110	O multiplicando é subtraído
111	Não há soma

Vale notar que a multiplicação por 2 do multiplicando pode ser feita através de um deslocamento aritmético para a esquerda. Esta nova técnica reduz o número de adições e, assim, reduz o atraso na sua operação (38). Entretanto, o consumo de potência é alto (38, 83). Também, o processo envolve comparações, adições e subtrações, o que diminui a velocidade de operação à medida que o multiplicando e o multiplicador aumentam de tamanho (38).

2.5.3 Multiplicador Wallace Tree

O multiplicador Wallace Tree foi proposto a fim de ser uma alternativa de multiplicador rápido e paralelo, já que as somas sequenciais são diminuídas para reduzir a acumulação de produtos parciais (77). Esta diminuição de adições é obtida através da criação de uma árvore de soma, que recebe como entrada os produtos parciais gerados de forma paralela, e os propaga até que a soma final seja atingida (87).

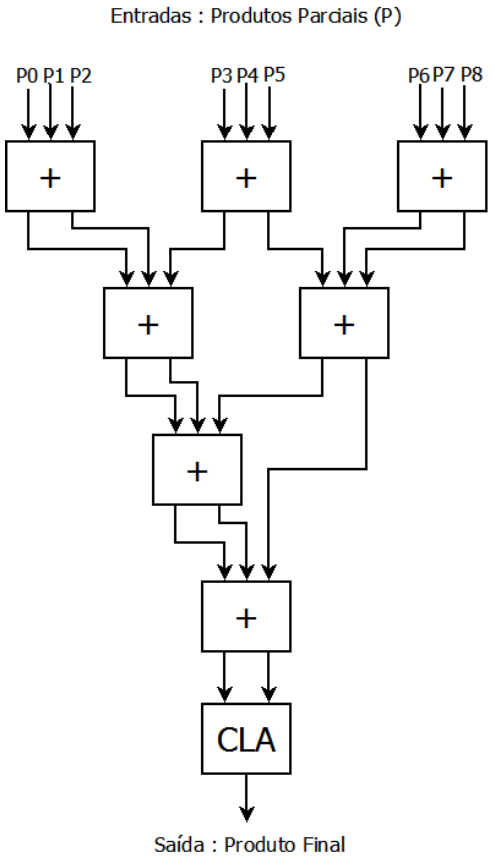

Figura 20 – Árvore de somadores de Wallace

O número de níveis necessários para se obter o produto final depende da quantidade de produtos parciais da multiplicação, que por sua vez é dependente da quantidade de bits das entradas (88).

A grande vantagem do multiplicador Wallace Tree vem da diminuição do atraso na execução, que pode ser comparado ao atraso de operações de adição. Também, o aumento

no atraso não é linear ao aumento de bits das palavras a serem multiplicadas, entretanto, a complexidade do multiplicador é maior (84).

Algumas soluções híbridas procuram usar as vantagens de duas ou mais técnicas para a implementação de multiplicadores. Uma das formas mais conhecidas é o multiplicador Booth codificado com Wallace Tree. Nesta abordagem, as entradas são codificadas segundo o algoritmo de Booth, diminuindo o número de produtos parciais e a árvore de Wallace é usada para somar estes produtos parciais. Esta abordagem traz uma diminuição geral no tempo de execução, melhorando a performance do multiplicador (77).

2.5.4 Multiplicador Vedic

A aritmética Vedic é baseada em 16 sutras, que trata de várias áreas no campo da matemática (38). Os sutras são conhecidos como técnicas que melhoram os atrasos nas implementações das operações e aumentam sua velocidade (89). O método de operação usado por eles é baseado na maneira que a mente humana calcula (90).

O sutra que trata da multiplicação de forma generalizada é o terceiro, conhecido como Urdhva Tiryakbhyam, que significa "vertical e cruzado" (38). A maneira como este sutra funciona é mostrada a seguir:

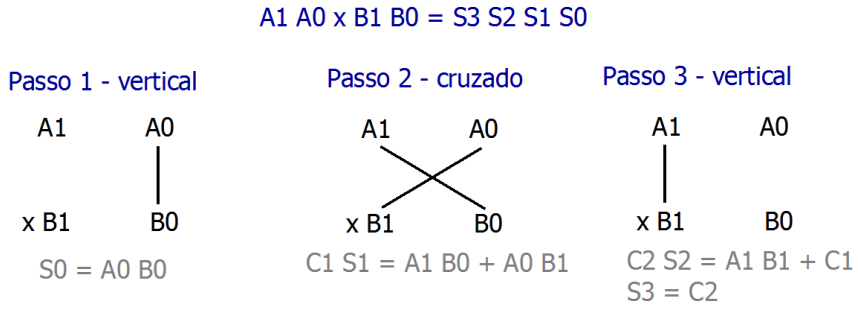

Figura 21 – Funcionamento do Multiplicador Vedic

Como o nome do sutra diz, essa técnica de multiplicação é baseada no processamento vertical e cruzado. Tomando como exemplo dois números de dois bits a serem multiplicados (A1A0 e B1B0). O resultado esperado é um valor de 4 bits S3S2S1S0. O primeiro passo é multiplicar A0 por B0 e definir o bit menos significativo da saída, S0 (multiplicação vertical). Em seguida, há uma multiplicação cruzada: A1 é multiplicado por B0 e A0 por B1. Esses produtos parciais são então somados e geram o bit S1 da saída e um carry C1. O último passo também é uma multiplicação vertical do bit A1 pelo B1. Esse produto é somado ao carry C1 e define os bits de saída S2 e S3. Para o caso do exemplo, ou seja, entradas com dois bits, são necessárias 4 multiplicações e, como se trata da notação binária, as multiplicações necessárias são feitas somente com portas lógicas, como mostrado na Figura 22.

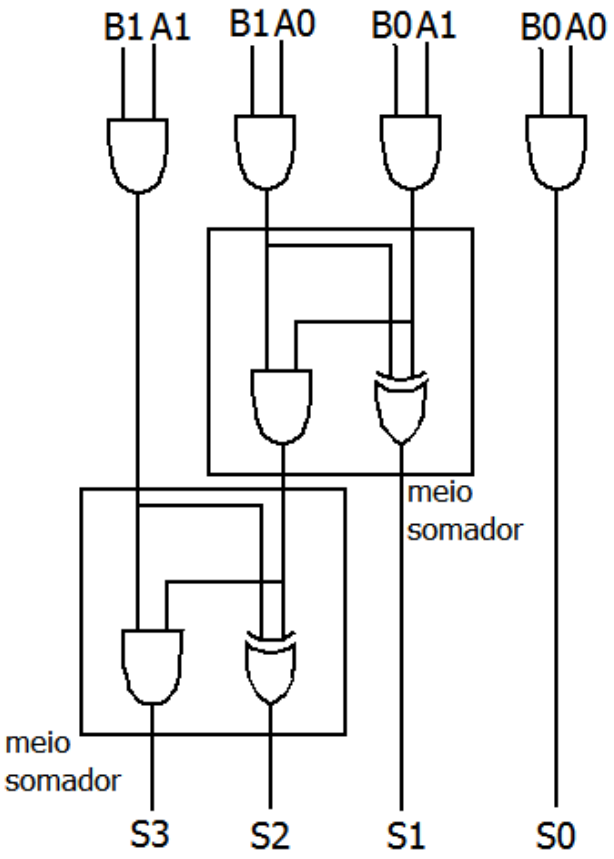

Figura 22 – Multiplicador Vedic de 2 bits

O multiplicador Vedic de 2 bits é formado por portas AND, responsáveis por multiplicar os bits de entrada e meio somadores, responsáveis por somar os produtos parciais.

Se ao invés de sinais de entrada de 2 bits, fossem usados sinais de 4 bits, o processo seria o mesmo, porém as entradas seriam divididas ao meio e o carry produzido pela primeira multiplicação deveria ser levado em consideração. A Figura 23 mostra este caso.

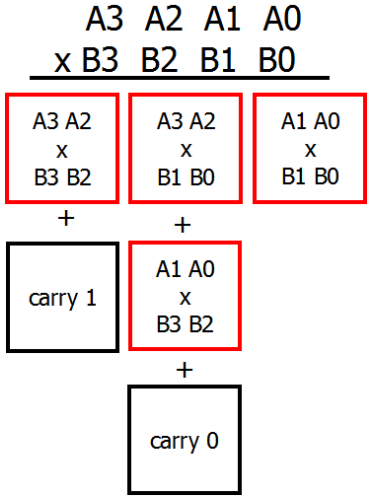

Figura 23 – Multiplicação Vedic de 4 bits

Pode-se notar que, para multiplicar números de 4 bits são necessários 4 multiplicações de 2 bits e 3 adições. Assim, o princípio do multiplicador Vedic é usar a estratégia dividir para conquistar (38), ou seja, a multiplicação de entradas de N bits é feita através de multiplicações de N/2 bits e somadores de N bits. A estrutura geral para um multiplicador de N bits pode ser vista na Figura 24.

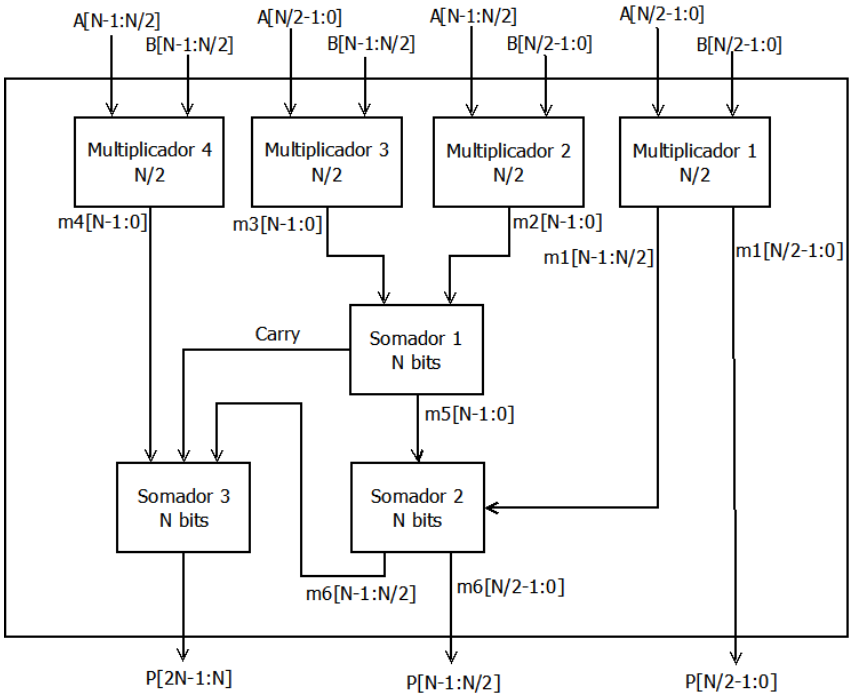

Figura 24 – Multiplicador Vedic de N bits

Esta estrutura se repete com entradas de N/2 bits, N/4 bits e assim por diante, até se chegar em palavras de 2 bits, que podem ser facilmente implementadas com portas lógicas, como mostrado anteriormente (Figura 22).

Com este tipo de estrutura, se o número de bits das palavras de entrada aumenta, a área e o atraso da implementação aumentam a uma menor taxa, se comparado com outras estruturas (90, 91). Pode-se notar que os produtos parciais são feitos de forma paralela, ajudando a diminuir o tempo de execução (38). Como a estrutura é regular, ela é simples de ser aumentada (86). Também, esta estrutura tem menores consumos de energia (82).

Em geral, implementações de multiplicadores do tipo Vedic trazem menores consumos de energia, menores atrasos e maiores velocidades de operação que outras implementações de multiplicadores. Porém, estas melhorias vêm a custo de área. Entretanto, esta diferença em área não é muito acentuada, ou seja, a área da implementação Vedic não fica muito diferente de outras implementações de multiplicadores. O menor consumo de energia é uma característica importante se a RNA for implementada em sistemas im-

plantados ou embarcados, sendo assim, para este trabalho foi escolhido trabalhar com o multiplicador Vedic.

3 Arquitetura Proposta

A arquitetura aqui implementada é baseada na criação de uma só camada de neurônios que é reutilizada durante toda a execução da rede (12, 21, 40, 45, 62), diminuindo a área da implementação. Na abordagem proposta, algumas características da rede podem ser reconfiguradas durante a execução:

- Número de entradas;
- Número de neurônios por camada;
- Número de camadas;
- Presença de *bias* nos neurônios da camada;
- Função de ativação executada pelos neurônios da camada (Degrau Bipolar, Rampa Limitada, Tangente Hiperbólica ou Sigmóide Logística).

A escolha de que tipo de arquitetura será executada e quais características esta arquitetura terá é feita pelo usuário por meio de poucas e simples instruções, de modo que a atualização é feita em tempo de execução, sem a necessidade de uma nova compilação do código, mantendo fixo o tamanho da implementação.

Em geral, a proposta deste trabalho é desenvolver uma camada de neurônios que pode ser reutilizada, permitindo reconfiguração da arquitetura da RNA a tempo de execução. Nesta arquitetura o usuário define como deverá ser a rede e toda a reconfiguração interna é feita de maneira autônoma.

Algumas decisões precisaram ser tomadas antes da implementação da Rede Neural Artificial, como a quantidade de neurônios na camada física, a quantidade máxima de camadas possível e a quantidade de bits das palavras da RNA. Tomando a quantidade de camadas para aplicações com RNAs do tipo MLP, sabe-se que em redes de três ou mais camadas a quantidade de problemas representáveis e solucionáveis é muito grande (92). Devido a isso, foi definido que a quantidade máxima de camadas de neurônios para esta implementação seria 4. Para que 3 camadas sejam suficientes para resolver uma grande variedade de problemas, a quantidade de neurônios em cada camada também deve ser alta (65). Por isto foi escolhido implementar uma camada física com 20 neurônios de 20 entradas. Assim, a maior rede possível nesta implementação tem 20 entradas e 4 camadas com 20 neurônios cada.

Para permitir ainda uma maior flexibilidade na topologia da rede, cada neurônio pode executar uma entre 4 funções de ativação: Degrau Bipolar, Rampa Limitada, Tan-

gente Hiperbólica ou Sigmóide Logística. Estas funções são as mais comuns em redes do tipo MLP e permitem a solução de uma grande variedade de problemas (9, 25).

Outros pontos importantes no projeto de Redes Neurais Artificiais em hardware são a notação numérica usada e a quantidade de bits necessários para que não haja grande perda na resolução da Rede Neural Artificial. A notação usada para as entradas, saídas, pesos, *bias* e toda operação da RNA foi a de ponto fixo com complemento de 2, pois esta notação traz operações com circuitos menores e mais simples do que aquela em ponto flutuante (42, 47). Sobre a escolha da quantidade de bits das entradas e pesos da RNA, alguns estudos apontam que Redes Neurais Artificiais digitais com entradas de 8 bits e pesos de 16 bits tem respostas satisfatórias para uma grande quantidade de problemas e, por isso, estes valores foram escolhidos para esta implementação (16, 93).

Como a camada física de RNA vai ser usada por todas as camadas virtuais, a sua saída será realimentada na entrada, demandando que a quantidade de bits destes sinais seja igual, ou seja, a saída da Rede Neural Artificial também deve ter 8 bits. A saída dos neurônios é definida através de funções de ativação e as funções possíveis para esta implementação incluem a Sigmóide Logística e Tangente Hiperbólica. Estas duas funções têm variação limitada na saída (42) e, levando isso em consideração, foi decidido que os sinais de entrada teriam 3 bits para a parte inteira e 5 bits para a parte decimal; seguindo este princípio, os pesos têm 6 bits de parte inteira e 10 bits na parte decimal.

Assim que os pesos e entradas são multiplicados, gera-se um sinal de 32 bits, para que não haja perdas devido à exatidão. Este sinal é interno à RNA e tem 12 bits de parte inteira de 20 de parte decimal.

O resumo do intervalo representado pelos sinais e o erro máximo para cada sinal são mostrados na Tabela 7.

Tabela 7 – Faixa dos sinais usados

Quantidade de Bits	Notação	Erro máximo (e)	Menor Valor	Maior Valor
i+d	i.d	2^{-d}	-2^{i}	$2^{i} - e$
8	3.5	0,03125	-4	3,96875
16	6.10	0,0009765625	-32	31,9990234375
32	12.20	0,00000095367431640625	-2.048	2047,99999904632

Toda a Rede Neural Artificial foi implementada no software Quartus II Web Edition®, da empresa Altera®. A FPGA usada para as simulações pertence ao modelo Cyclone IV EP4CE115F29C7. Os códigos foram descritos na linguagem de descrição de hardware Verilog e simulados no software ModelSim - Altera®. O leitor pode encontrar todos os códigos usados em um CD anexo.

O modelo geral do design proposto é visto na Figura 25.

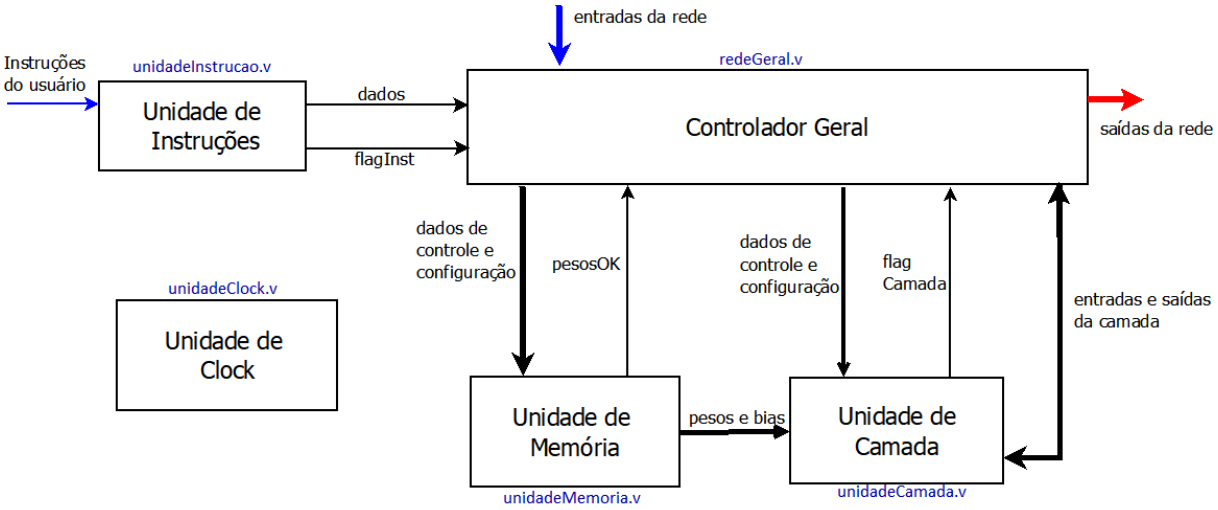

Figura 25 – Estrutura Geral da RNA

A estrutura é formada por 5 grandes blocos: a Unidade de Clock, responsável por dividir a frequência do *clock* de entrada a fim de atingir um valor necessário na Unidade de Camada e distribuir o *clock* por todo o sistema. A Unidade de Instruções é responsável por receber as instruções usadas na configuração da rede; seus dados são enviados para o Controlador Geral, que gerencia toda a execução da RNA. A Unidade de Memória contém as memórias com os pesos e *bias* para cada neurônio. Por fim, a Unidade de Camada é composta por 20 neurônios responsáveis pela execução da rede. Os sinais de *clock* e *reset* não foram mostrados para facilitar a visualização.

Mais detalhes de cada bloco serão detalhados nas seções a seguir.

3.1 Unidade de Clock

A Rede Neural Artificial Reconfigurável aqui proposta usa o *clock* disponível na FPGA. Porém, o multiplicador usado pelos neurônios necessita de um *clock* com menor frequência, sendo detalhado na seção 3.4.2.1. Para atingir a frequência necessária, um bloco que divide a frequência inicial por 3 é implementado. A estrutura deste bloco pode ser vista na Figura 26.

Como visto na Figura 26, este bloco recebe o *clock* da FPGA como entrada (*clk*) e também um sinal de *reset* (*rst*). Como saída tem-se o sinal de *clock* com a frequência alterada (*clkMAC*).

A alteração da frequência é feita através de *flip-flops* do tipo D e portas lógicas *E* e *OU*. O funcionamento vai ser detalhado através de um exemplo, visto na Figura 27.

Para este teste, o sinal de entrada do *clock* foi definido com um período de 20ns, porém, este valor não é o valor máximo para a arquitetura final da RNA, como será visto adiante.

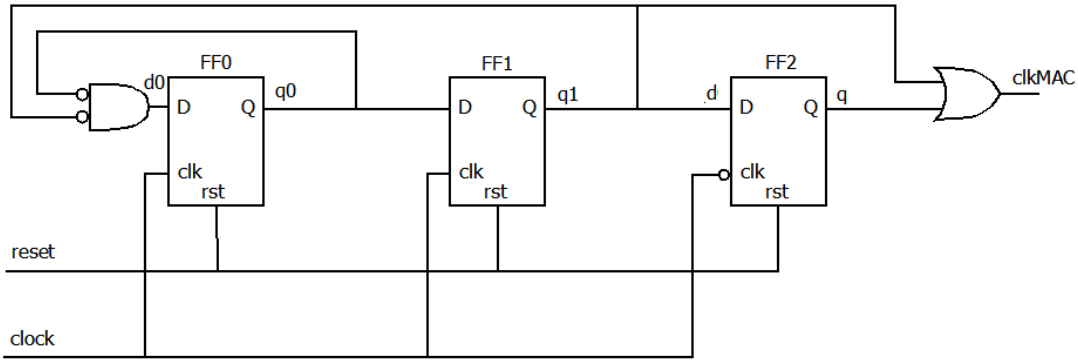

Figura 26 – Unidade de Clock

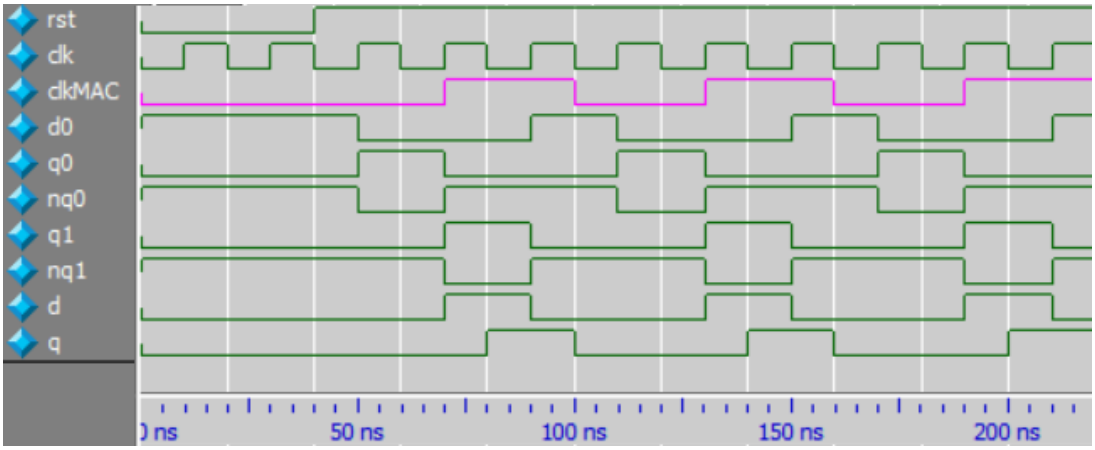

Figura 27 – Carta de Tempo da Unidade de Clock

Para facilitar a análise, os períodos de *clock* foram separados em *grids*, permitindo uma melhor visualização. Durante o primeiro ciclo de *clk*, o sinal de *reset* está em 0, indicando que as saídas de todos os *flip-flops* (*q0*, *q1* e *q*) devem ser levadas a 0. Os sinais *nq0* e *nq1* são o inverso das saídas *q0* e *q1*, ou seja, eles têm valor 1 durante o *reset*. Já o sinal *d0* é obtido através de uma lógica *E* entre *nq0* e *nq1* e também tem valor 1 durante o *reset*. A partir do segundo ciclo de *clk* o *reset* vai a 1 e o funcionamento da Unidade de Clock começa. Os *flip-flops* 0 e 1 funcionam na borda positiva do *clock* de entrada e o *flip-flop 2*, na borda negativa.

Tomando o *flip-flop 0*, na borda positiva do segundo ciclo de *clk*, sua entrada *d0* é igual a 1, fazendo com que a saída *q0* passe a 1 e a *nq0* passe a 0. Como a entrada d0 depende do valor de *nq0*, com a mudança desse sinal para 0, *d0* também muda para 0. A saída do *flip-flop 0* é a entrada *d* do *flip-flop 1*. Na borda positiva deste mesmo ciclo, a saída *q1* continua em 0, já que *q0* é 0. O *flip-flop 2* funciona durante a borda negativa do *clock* de entrada e recebe como entrada a saída do *flip-flop 1*. Na borda de descida deste ciclo de *clk*, a saída *q* recebe o valor 0, já que *q1* também é 0. A saída *clkMAC* é definida através de um *OU* entre *q1* e *q*, e, no instante de descida do ciclo, recebe nível lógico 0. O funcionamento continua seguindo esses passos.

Nota-se que os sinais *q0*, *q1* e *q* são sinais periódicos deslocados entre si. Os sinais *q0* e *q1* são deslocados em 1 ciclo de *clock* e o *q* está deslocado meio ciclo de *q1*. Esta formação garante que o sinal de saída, obtido através da lógica *OU* entre *q1* e *q*, seja um sinal de período 3 vezes maior que o *clock* de entrada e com ciclo de trabalho de 50%.

Vale observar na simulação da Figura 27, que o valor de *clkMAC* só está correto a partir da borda de subida do 3º ciclo de *clock*, já que no início da simulação o sinal de *reset* interfere na saída. Porém, este fato não prejudica o funcionamento da RNA, pois o *clkMAC* somente é usado durante a soma ponderada das entradas, o que não ocorre imediatamente após o *reset*.

3.2 Unidade de Instruções

A Unidade de Instruções é responsável por receber as instruções que definem a topologia da rede vindas do usuário (*instrucao*), extrair e organizar as informações necessárias para que a execução da RNA possa iniciar. Este bloco também recebe como entrada um sinal *flag* indicando que a instrução deve ser recebida (*flagInst*), um sinal de *clock* (*clk*) e um sinal de *reset* (*rst*).

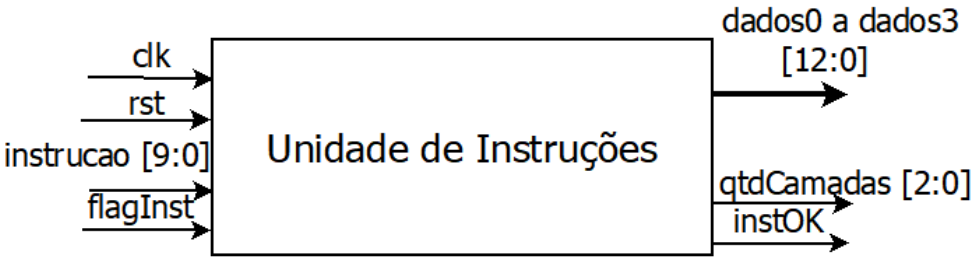

Figura 28 – Unidade de Instruções

Esta unidade realiza os passos iniciais do processamento da RNA e é executada toda vez que uma nova topologia for necessária. A indicação de uma nova topologia é dada pelo sinal *rst*.

A instrução é definida como um sinal de 10 bits, onde cada conjunto de bits indica uma característica da rede. É necessária uma instrução para cada camada que compõe a rede.

Tabela 8 – Características do sinal *instrucao*

Bits	Função	Possíveis Valores
i9 i8	Tipo de camada da RNA	00, 01 ou 11
i7 i6 i5 i4 i3	Quantidade de neurônios na camada	00000 a 10011
i2	Presença de *bias*	0 ou 1
i1 i0	Função de Ativação	00, 01, 10 ou 11

Na Tabela 8, cada parte da instrução é detalhada. Os bits i9 e i8 indicam o tipo de camada da RNA. Os valores possíveis para esses bits são 00, 01 e 11, indicando uma camada de entrada, oculta ou de saída, respectivamente. Os bits i3 a i7 indicam quantos neurônios a camada possui. Esta quantidade pode variar de 00000 a 10011(0 a 19), que representam a presença de 1 a 20 neurônios na camada. O bit i2 representa a presença de *bias* na camada: se este sinal é 1, há *bias* na camada; se é 0, não há *bias*. Os dois últimos bits i1 e i0 indicam o tipo da função de ativação implementado pelos neurônios da camada. Estes bits podem receber os valores 00, 01, 10 ou 11, indicando que a função de ativação usada na camada é Degrau Bipolar, Rampa Limitada, Tangente Hiperbólica ou Sigmóide Logística.

A cada nova topologia, um novo grupo de instruções deve ser colocado na Unidade de Instruções sequencialmente e o sinal *flagInst* deve ser acionado a cada uma delas. Este sinal indica que existe uma instrução na entrada do bloco pronta para ser analisada.

Estas instruções são então separadas e colocadas em registradores com as informações organizadas para a execução de cada camada de neurônios da rede. Por exemplo, nas instruções não existem bits que indicam quantas entradas há na camada. Esta informação é calculada de acordo com o número de neurônios na camada anterior, já que a quantidade de saídas de uma camada é igual à quantidade de entradas da camada seguinte.

No caso da primeira instrução, o campo que indica quantidade de neurônios da rede (i3 a i7) representa, na realidade, a quantidade de entradas da rede, já que a primeira camada de uma Rede Neural Artificial do tipo MLP não é formada por neurônios, e sim por unidades sem processamento. Estas informações são colocadas em registradores chamados *dados*. Como a implementação permite até 4 camadas de neurônios, existem até 4 registradores *dados*, cada um responsável por uma camada. A divisão deste registrador em bits é vista na Tabela 9.

Tabela 9 – Características do registrador *dados*

Bits	Função
i12 i11 i10 i9 i8	Quantidade de neurônios na camada
i7	Presença de *bias*
i6 i5	Função de Ativação
i4 i3 i2 i1 i0	Quantidade de entradas da camada

Cada um dos registradores *dados* é formado por um sinal de 13 bits e carrega as informações da camada em ordem. Estes registradores, ao final da execução da Unidade de Instruções, são enviados ao Controlador Geral para que a execução da RNA possa continuar. Além desta informação, a Unidade de Instruções também envia ao Controlador Geral um sinal indicando quantas camadas a rede especificada deve ter (*qtdCamadas*) e um *flag* indicando que a execução do bloco está finalizada (*instOK*).

O processamento deste bloco é dividido em duas partes, uma é responsável por controlar o recebimento das instruções e outra controla a análise destas.

Na etapa de recebimento de instruções,o fluxo da Figura 29 é executado.

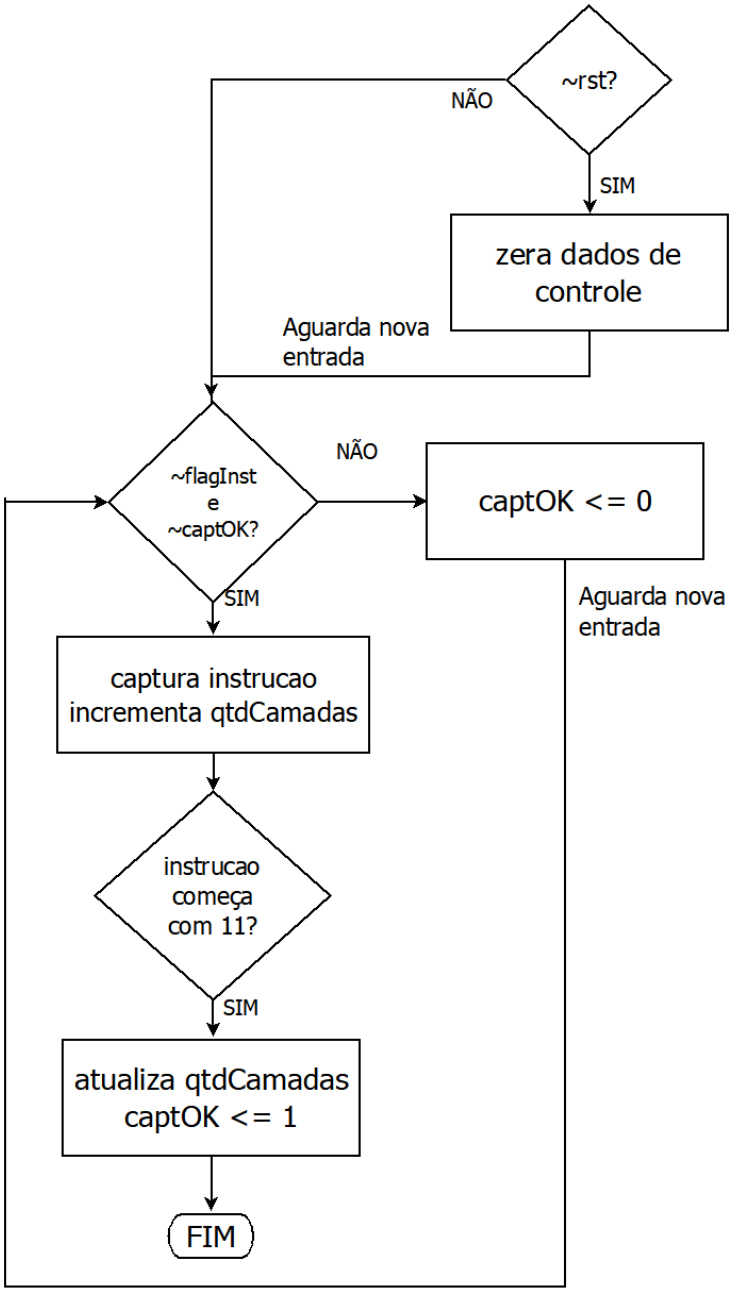

Figura 29 – Fluxo do recebimento de instruções

Este fluxo inicia com o sinal *rst*, que faz com que o *flag captOK* e o registrados *qtdCamadas* sejam zerados. O *captOK* indica que o recebimento das instruções foi finalizado e o *qtdCamadas* indica a quantidade de camadas que a RNA deverá executar. Fica-se então aguardando o sinal *flagInst*, que indica que há uma nova instrução pronta para ser capturada pelo bloco. Nesta etapa também é verificado se o sinal *captOK* é 0, já que se este sinal for 1, todas as instruções necessárias já foram recebidas. Isto garante que

nenhum sinal capaz de mudar a arquitetura da RNA seja recebido depois que ela já foi definida. Se ambos os sinais *captOK* e *flagInst* são 0, a instrução na entrada é capturada e o *qtdCamadas* é incrementado. Se esta instrução começa com 11, ou seja, se é uma instrução que define a camada de saída, o sinal *captOK* vai a 1 e a captura de instruções é finalizada. Caso contrário, o sinal *captOK* continua em 0 e uma nova instrução é aguardada.

Em paralelo ao recebimento das instruções, uma Máquina de Estados (MdE) funciona para o controle da análise das instruções. Esta máquina é responsável por separar as informações de cada instrução e colocá-las como saída desta Unidade. O diagrama de estados desta MdE pode ser visto na Figura 30.

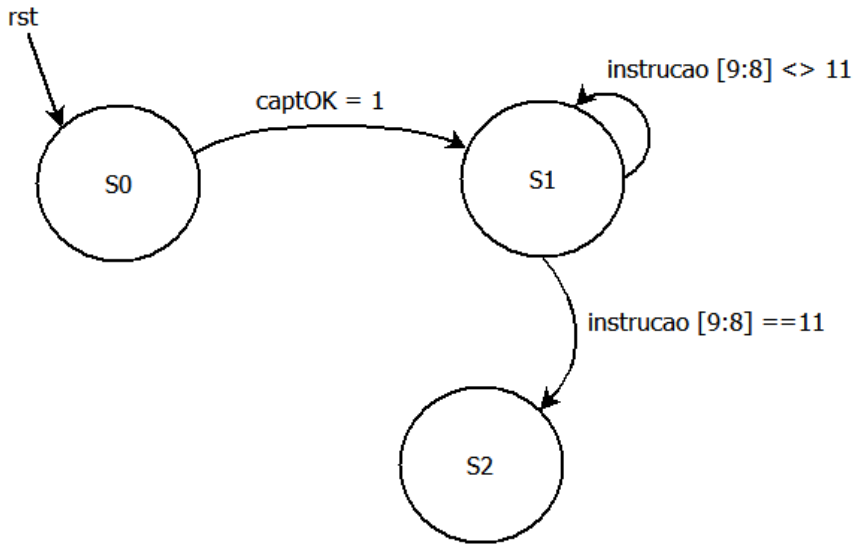

Figura 30 – MdE da Unidade de Instruções

Esta máquina de estados funciona na borda negativa do *clock*. O funcionamento da MdE é iniciado pelo sinal externo *rst*, indicado pelo usuário toda vez que uma nova topologia da rede for necessária. A partir deste sinal, a rede vai para o estado S0. O sinal *rst* é responsável por reiniciar a máquina de estados a qualquer momento da execução. O funcionamento de cada estado é mostrado a seguir:

rst: este sinal garante o início da busca de instruções. Quando ativo, os dados de saída (*dados*, *instOK*) e de controle são levados a 0 e a máquina de estados vai para S0.

S0: a MdE permanece neste estado enquanto a captura de instruções ainda não finalizou, ou seja, enquanto o sinal *captOK* está em 0. Assim que este sinal vai a 1, indicando que todas as instruções já foram recebidas, a MdE passa ao estado S1.

S1: cada uma das instruções recebidas é analisada e colocada nos registradores específicos *dados*, cada um correspondendo a uma camada da rede. Assim que esta análise é finalizada, ou seja, quando a instrução com bits i9 e i8 iguais a 11 for analisada, a máquina passa para o estado S2.

S2: este último estado disponibiliza os registradores *dados* para o Controlador Geral, além da quantidade de camadas e o *flag instOK*, indicando que a busca de instruções já foi finalizada e processamento da RNA pode prosseguir.

Um exemplo de simulação para uma Rede Neural Artificial de 3 camadas é mostrado na Figura 31.

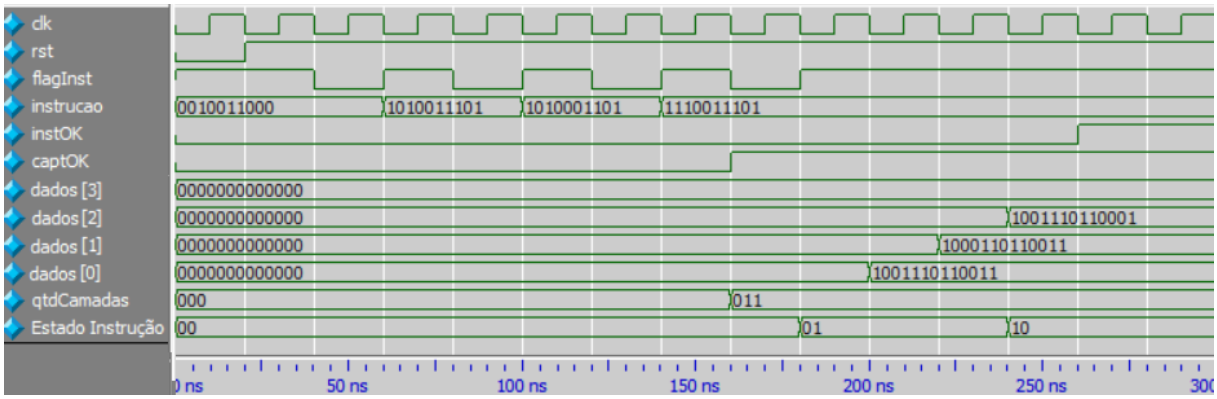

Figura 31 – Carta de Tempo da Unidade de Instruções

A etapa de Busca de Instruções tem seu tempo execução afetado pelo tempo que o usuário gasta com os sinais de *reset* e o tempo para inserir todas as execuções, além de também ser afetado pela quantidade de execuções, ou seja, pela quantidade de camadas que a RNA deverá executar. Assim como na simulação da Unidade de Clock, foi usado um *clock* de entrada com período de 20ns e cada ciclo de *clock* é separado por um *grid*.

Neste exemplo, o *reset* fica ativo por um ciclo de *clock*, cada instrução leva 2 ciclos de *clock* para ser colocada no sistema e são necessárias 4 instruções para definir a RNA. São gastos, portanto, 8 ciclos de *clock* para que todas as instruções necessárias sejam buscadas, analisadas e disponibilizadas para o Controlador Geral. Nota-se que, assim que o *captOK* se torna 1, o valor *qtdCamadas* é atualizado para a quantidade referente à RNA em execução. No caso, seu valor é o binário 11, indicando que a Rede Neural Artificial tem 1 camada de entrada e 3 camadas de neurônios. Para facilitar a análise da Unidade de Instruções, os sinais *captOK* e Estado Instrução foram mostrados na simulação. Na aplicação final estes sinais não são usados. Como a topologia escolhida pelo usuário tem 3 camadas de neurônios, somente os registradores *dados* de 0 a 2 são usados e o registrador *dados3* se mantém em 0.

3.3 Unidade de Memória

A Unidade de Memória é formada pelas memórias que guardam os pesos e *bias* usados na execução das camadas, além de um controlador, responsável por retirar as informações corretas da memória e enviá-las à Unidade de Camada, para que esta possa iniciar sua execução.

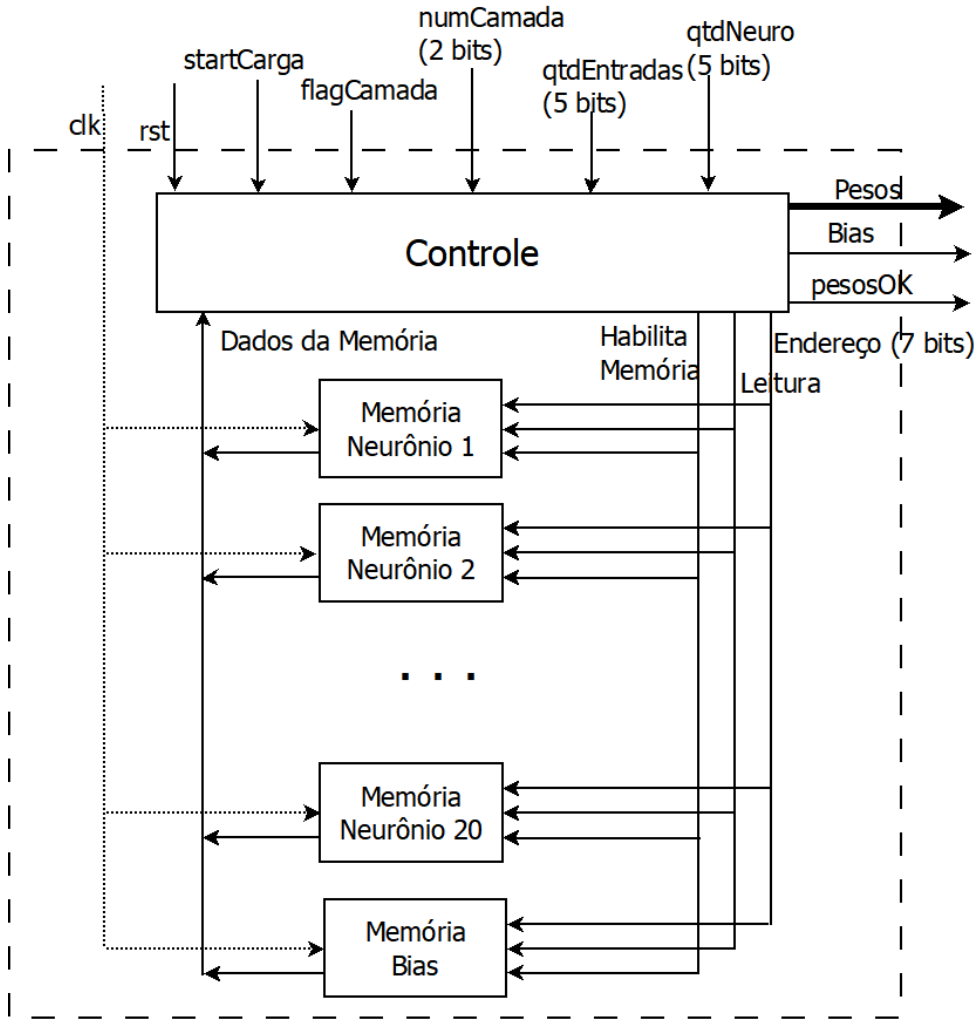

Figura 32 – Unidade de Memória

Além de um sinal de *clock* (*clk*) e um sinal de *reset* (*rst*), a Unidade de memória também recebe alguns sinais de entrada:

startCarga: indica que a busca de pesos deve iniciar. É usado na máquina de estados que controla este bloco.

flagCamada: indica que a camada que está sendo executada terminou sua execução. É usado juntamente com *startCarga* para indicar que uma nova busca de pesos deve iniciar.

numCamada (2 bits): indica qual a camada que está sendo executada. É usado no cálculo do endereço da memória a ser buscado.

qtdEntradas (5 bits): indica a quantidade de entradas da camada. Controla o número de vezes que a memória é acessada.

qtdNeuro (5 bits): indica a quantidade de neurônios na camada a ser executada. Usado juntamente com *qtdEntradas* como controle de quantas vezes a memória deve ser acessada.

Como saída, o bloco Unidade de Memória apresenta 400 valores de 16 bits que

representam os pesos (Pesos, na Figura 32) e 20 valores de 16 bits que representam os *bias* (*Bias*, na Figura 32), além do sinal *pesosOK* que indica se a execução deste bloco está finalizada.

A quantidade de pesos e *bias* na saída da Unidade de Memória é definida pelo número de pesos necessários na execução da maior camada possível da Rede Neural Artificial implementada. Como a maior camada possível é formada por 20 neurônios com 20 entradas e cada entrada de cada neurônio tem um peso correspondente, 400 pesos são necessários. Já o *bias* é definido como um para cada neurônio, como a rede tem 20 neurônios no máximo por camada, 20 valores são necessários.

A organização das memórias foi escolhida de maneira que o acesso aos pesos e *bias* necessários para cada entrada e cada neurônio fossem feitos paralelamente. Esta característica diminui o tempo de busca dos dados, reduzindo o tempo de execução total da rede.

No total foram construídas 21 memórias de 128 palavras, cada uma delas de 16 bits. Cada neurônio tem uma memória correspondente (ou seja, total de 20 memórias) e a última delas guarda o *bias* de todos os neurônios. Todas elas são acessadas paralelamente sempre que uma nova camada for executada. Nota-se que, apesar da memória ter possibilidade de 128 palavras, somente 80 palavras são realmente usadas, já que cada neurônio precisa de no máximo 20 pesos por camada e a RNA pode executar 4 camadas virtuais no máximo, totalizando 80 pesos. A memória foi construída com 128 palavras devido à forma que ela foi organizada. Apesar de uma maior área gasta, há uma diminuição no tempo de busca. Também, isto facilita uma possível expansão da Rede Neural Artificial.

Cada uma das memórias dos neurônios é organizada da mesma maneira, de forma que, através do número que indica qual camada a ser executada (*numCamada*) e da quantidade de entradas na camada (*qtdEntradas*), é possível calcular o endereço a ser acessado nestas memórias. Cada uma das memórias recebe um sinal de 7 bits indicando o endereço a ser acessado e dois sinais de controle, indicando que a memória está ativa e que a leitura da memória deve iniciar, além de sinais de *clock* e *reset*. A organização geral pode ser vista nas Figuras 33 e 34.

Na Figura 33 as memórias estão organizadas por camada. A notação *w3.2* indica o peso da terceira camada, referente à entrada 2, para o neurônio indicado pela memória.

Ao contrário dos pesos, que são separados por neurônios, os *bias* de todos os neurônios são armazenados juntos em uma única memória, como mostrado a seguir.

Apesar de ter a mesma quantidade de posições que a memória de pesos, a memória de *bias* é organizada de maneira diferente, pois ela guarda os *bias* de todos os neurônios, como visto na Figura 34. A notação *bias*1_2 indica o *bias* da camada 1 referente ao neurônio 2. Apesar disso, a memória de *bias* é acessada com o mesmo endereço das memórias

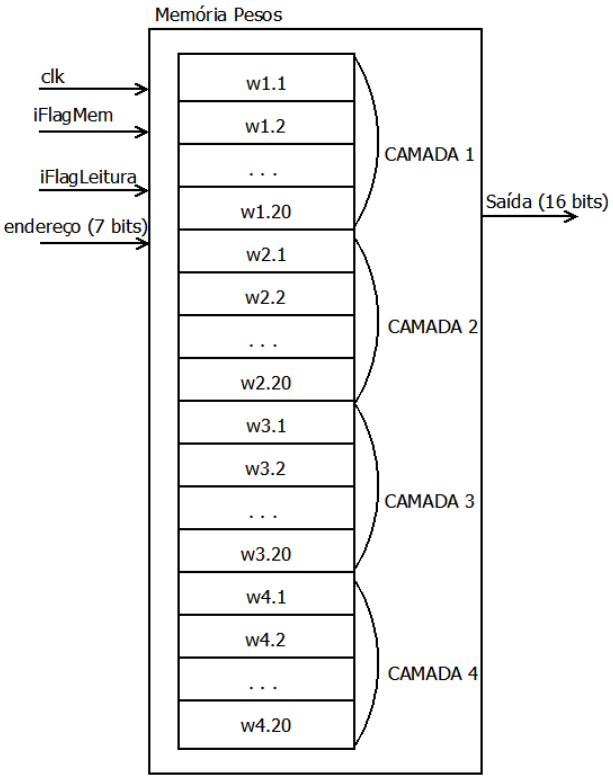

Figura 33 – Organização da Memória de Pesos

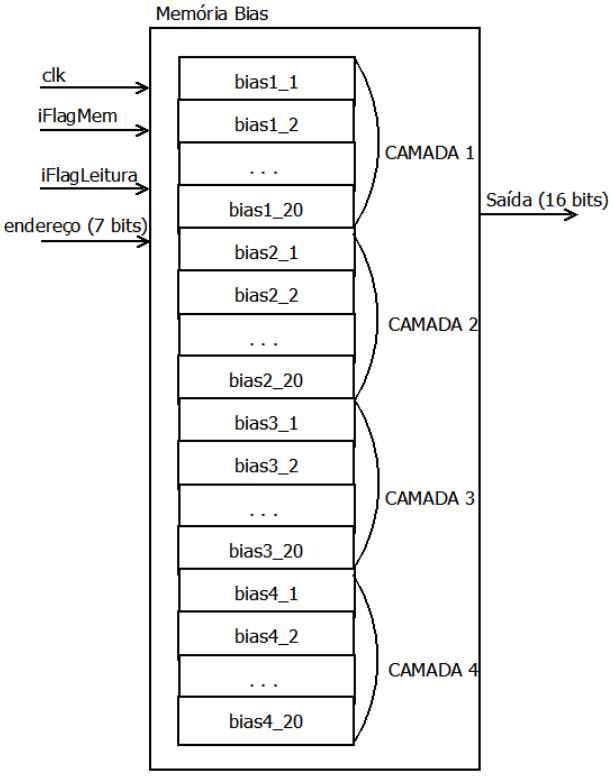

Figura 34 – Organização da Memória de *bias*

de pesos.

O endereço para acesso de todas as memórias é calculado usando a entrada *num-*

Camada e um contador (*cont*), limitado tanto pelo sinal *qtdNeuro* (para o caso do *bias*) quanto *qtdEntradas* (para o caso dos pesos). Este contador tem 5 bits. A maneira que o endereço é montado pode ser vista a seguir:

Tabela 10 – Cálculo do endereço das memórias

numCamada [1]	*numCamada [0]*	*cont[4]*	*cont[3]*	*cont[2]*	*cont[1]*	*cont[0]*

Por exemplo, se os pesos do 15º neurônio da 2ª camada estão sendo buscados, o contador será igual a 01110 (pois a contagem se inicia em 0) e a camada será igual a 01. Logo, o endereço a ser buscado é 0101110.

A busca pelos pesos e *bias* é controlada por uma máquina de estados, acionada toda vez que uma nova camada for executada. O diagrama de estados da MdE é visto na Figura 35.

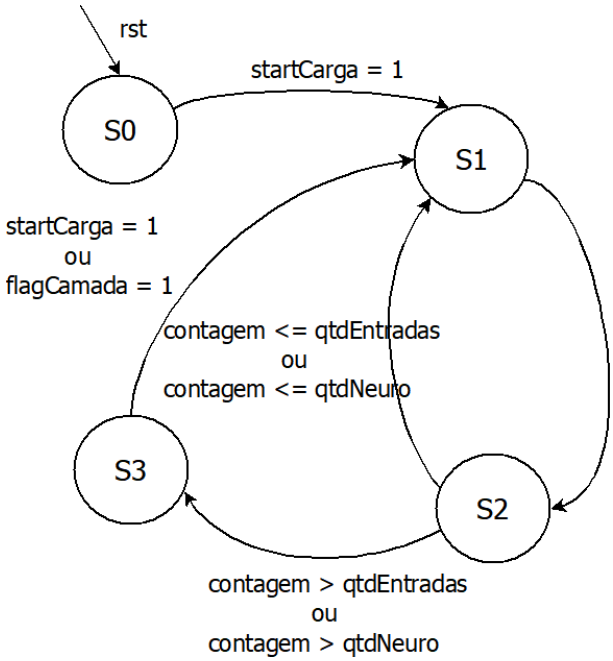

Figura 35 – MdE da Unidade de Memória

Esta máquina recebe instruções diretas do Controlador Geral, como comandos de quando a MdE deve executar, qual camada que está sendo executada, sua quantidade de neurônios e entradas. Assim, como a máquina de estados responsável pela busca de instruções, esta MdE é iniciada pelo sinal *rst*, onde os registradores das saídas são zerados. O funcionamento da máquina é detalhado a seguir:

S0: os flags de acesso à memória são colocados em 0 e o contador responsável pelo cálculo de endereços é zerado. Um sinal de controle enviado pelo Controlador Geral, chamado *startCarga*, é responsável por informar que a rede está pronta para iniciar a busca dos pesos: se este sinal for igual a 1, a MdE vai para o estado S1; caso contrário, a MdE permanece no estado S0.

S1: neste estado o endereço que será acessado na memória é calculado, conforme mostrado anteriormente. Com o endereço pronto, os flags de acesso à memória são ativados e a busca de pesos e *bias* é iniciada. A máquina de estados passa então para o estado S2.

S2: os flags de acesso às memórias são novamente colocados em 0 e a busca do primeiro grupo de pesos e *bias* é encerrada. Os dados buscados são colocados nos registradores de saída e o contador usado no cálculo do endereço é incrementado. Em seguida, é verificado se todos os pesos e *bias* já foram buscados. Essa verificação é feita através do contador: se seu valor for menor que a quantidade de neurônios da camada ou menor que a quantidade de entradas da camada, então a busca ainda não foi finalizada e a MdE volta ao estado S1. Caso contrário, a MdE vai para o estado S3.

S3: este estado é responsável por finalizar a busca dos pesos e *bias*. O *flag pesosOK* que indica que a busca está pronta é colocado em 1 e o contador é zerado. A máquina de estados permanece nesse estado até que uma nova busca seja necessária e o sinal *startCarga* seja colocado em 1, passado assim para o estado S1. Os pesos e *bias* necessários para a execução são enviados diretamente para a Unidade de Camada e o *flag pesosOK* é enviado ao Controlador Geral para que a execução da Rede Neural Artificial continue.

Pode-se observar que a máquina só vai para o estado S0 quando há um sinal de *reset* e este sinal só é ativado quando uma nova topologia de rede for necessária. Assim, somente a primeira execução de uma nova topologia de RNA passa pelo estado S0. Esta característica permite diminuir o número de ciclos de *clock* usados na execução da Rede Neural Artificial.

Para verificar o funcionamento da Unidade de Memória, a busca de pesos para uma camada com 4 neurônios e 3 entradas é mostrada na Figura 36.

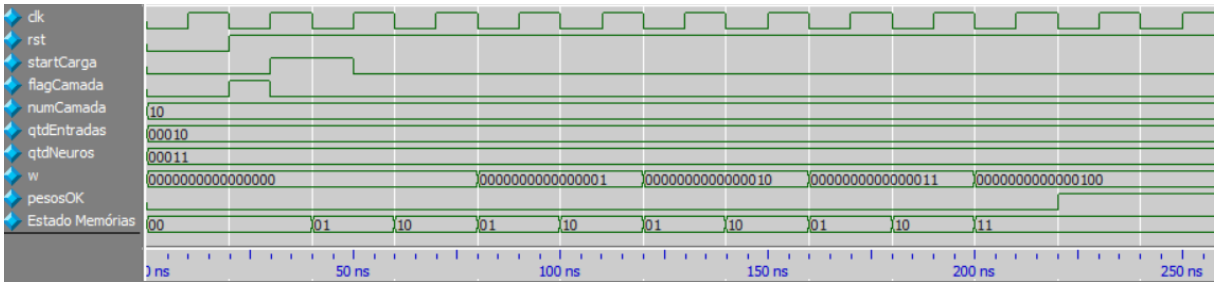

Figura 36 – Carta de Tempo da Unidade de Memória

Assim como nas simulações anteriores o *clock* tem período de 20 ns e cada período se encaixa em um intervalo de *grid*. A saída desta unidade é composta de 400 valores de 16 bits representando os pesos e 20 valores de 16 bits representando os *bias*. Porém, para facilitar a visualização somente um sinal é mostrado. O momento em que este sinal atualiza seus valores representa o momento que todas as outras saídas atualizariam. Também para facilitar a análise, o estado em que a MdE se encontra é mostrado.

Esta etapa do funcionamento da RNA gasta 11 ciclos de *clock* para ser finalizada e este valor está relacionado com a quantidade de entradas e de neurônios da camada em execução. Esta unidade inicia sua execução com o sinal *startCarga* e o estado que controla sua execução oscila entre os estados S1 e S2 até que todos os pesos e *bias* necessários sejam buscados. Como a execução desta etapa não sofre influência do usuário, pode-se obter uma fórmula que indique quantos ciclos de *clock* são necessários para sua finalização.

Inicialmente 1 ciclo de *clock* é gasto para que o sinal *startCarga* vá a nível lógico 0. Em seguida, os pesos devem ser buscados para cada entrada e os *bias* para cada neurônio. Como todos os acessos às memórias são feitos em paralelo, a quantidade de vezes que as memórias serão acessadas é igual ao número de entradas ou ao número de neurônios, dependendo de qual valor é maior. Para cada acesso à memória há a necessidade de mais um ciclo de *clock* para receber este valor e colocá-lo na ordem correta do registrador de saída. A finalização desta etapa se dá com dois ciclos de *clock*, um para que o sinal *pesosOK* vá a 1 e outro para que o Controlador Geral passe para o próximo estado (esta última parte não é mostrada na carta de tempo da Figura 36). Levando em conta a análise aqui feita, a quantidade de ciclos de *clock* gastos é igual a:

$$\text{ciclos busca pesos } l_i = 3 + 2.max\{n_i, e_i\} \tag{3.1}$$

Onde:

l_i: camada de número i;

n_i: quantidade de neurônios da camada i;

e_i: quantidade de entradas da camada i.

Porém, o estado S0 só será atingido uma vez, durante a execução da primeira camada da RNA. Nas demais camadas, a busca dos pesos inicia no estado S1. Assim, a fórmula que define a quantidade de ciclos usados na busca de instruções pode ser atualizada para:

$$\text{ciclos busca pesos } l_i = \begin{cases} 3 + 2.max\{n_i, e_i\}, & \text{se } i = 1 \\ 2 + 2.max\{n_i, e_i\}, & \text{se } i \neq 1 \end{cases} \tag{3.2}$$

Tomando a camada aqui testada, tem-se $l_i = l_1, n_1 = 3$ e $e_1 = 4$. Assim, a quantidade de ciclos de *clock* esperada é $3 + max.\{3, 4\} = 11$, que é igual ao valor observado na simulação.

3.4 Unidade de Camada

A Unidade de Camada é a responsável pelo processamento aritmético da Rede Neural Artificial. Esta unidade consiste de 20 neurônios operando em paralelo e um bloco

que controla sua saída.

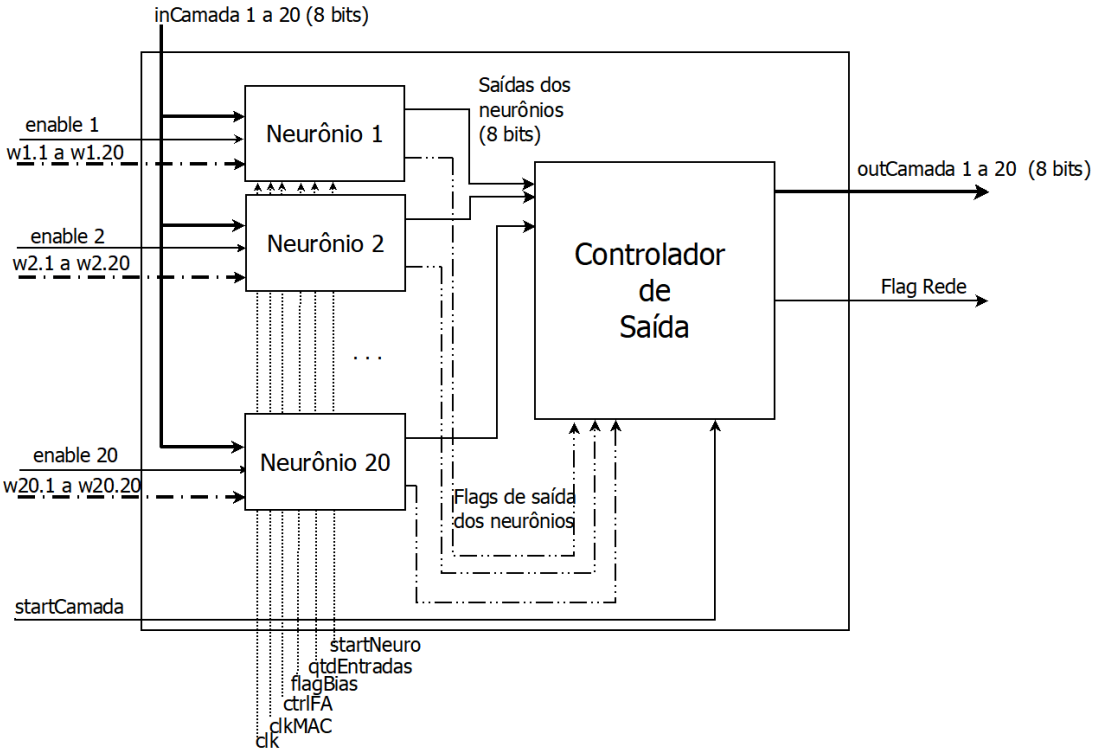

Figura 37 – Unidade de Camada

A Unidade de Camada recebe da Unidade de Memória os pesos e *bias* necessários para sua execução, indicados na Figura 37 como *w*. Da Unidade de Clock, esta unidade recebe dois sinais de *clock*, um usado no controle das saídas (*clk*) e outro usado na execução da multiplicação (*clkMAC*). Do Controlador Geral, recebe as entradas, que são iguais para todos os neurônios, além dos sinais de controle necessários para a execução:

flagBias: indica se há ou não *bias* nos neurônios da camada;

ctrlFA: indica qual função de ativação deve ser usada;

startCamada: indica que a camada está sendo executada. Vai a 1 assim que a execução da camada inicia e permanece assim até que a execução da camada termine. Este sinal age juntamente com o sinal *startNeuro*;

startNeuro: indica quando o neurônio deve iniciar sua execução;

qtdEntradas: usado para controle da máquina de estados que gerencia as multiplicações;

enable de 1 a 20: cada um destes sinais é enviado a um neurônio e indica qual deles está ativo na camada.

A camada pode ser dividida em dois blocos: Controlador de Saída e Neurônio, analisados a seguir.

3.4.1 Controlador de Saída

O Controlador de Saída recebe sinais dos neurônios que indicam que suas saídas estão disponíveis, além das próprias saídas dos neurônios (Figura 38).

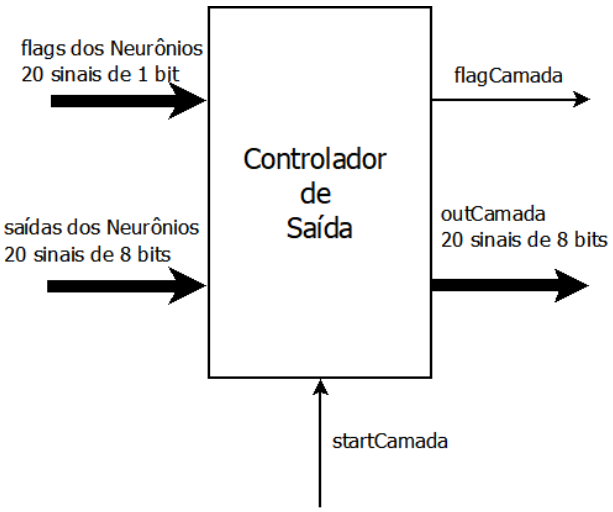

Figura 38 – Controlador de Saída da Camada

Este bloco também recebe o sinal *startCamada* que indica que o sistema está pronto para receber as saídas da camada. Como saída, há um *flag* que indica que as saídas estão disponíveis (*flagCamada*) além das saídas de cada um dos neurônios (*outCamada*). Estes sinais são enviados ao Controlador Geral que decide se as saídas vão para a entrada de uma nova camada ou se são as saídas externas da rede.

O Controlador de Saída tem como função principal garantir que as saídas dos neurônios sejam disponibilizadas de maneira simultânea para o Controlador Geral e também indicar a este bloco a finalização da Unidade de Camada, permitindo assim que a execução da Rede Neural Artificial continue.

3.4.2 Neurônio

Este é o bloco mais importante da Rede Neural Artificial já que é responsável por toda a execução aritmética da rede. Sua função é multiplicar as entradas pelos seus respectivos pesos, somar estes produtos, somar o *bias*, se estiver presente, e executar uma função de ativação responsável por definir a saída do neurônio. O neurônio é formado pelo bloco Multiplicador-Acumulador (MAC), responsável pela multiplicação e soma, e pelo bloco Função de Ativação (FA), responsável por executar as funções de ativação definidas pelo Controlador Geral e escolher qual delas define a saída do neurônio.

O Neurônio recebe da Unidade de Camada as entradas, pesos e *bias* usados na sua execução, além dos sinais de *clock* (*clk* e *clkMAC*). Também são recebidos os sinais de controle *flagBias*, *enable*, *ctrlFA*, *startNeuro*, *startCamada* e *qtdEntradas*, já detalhados acima.

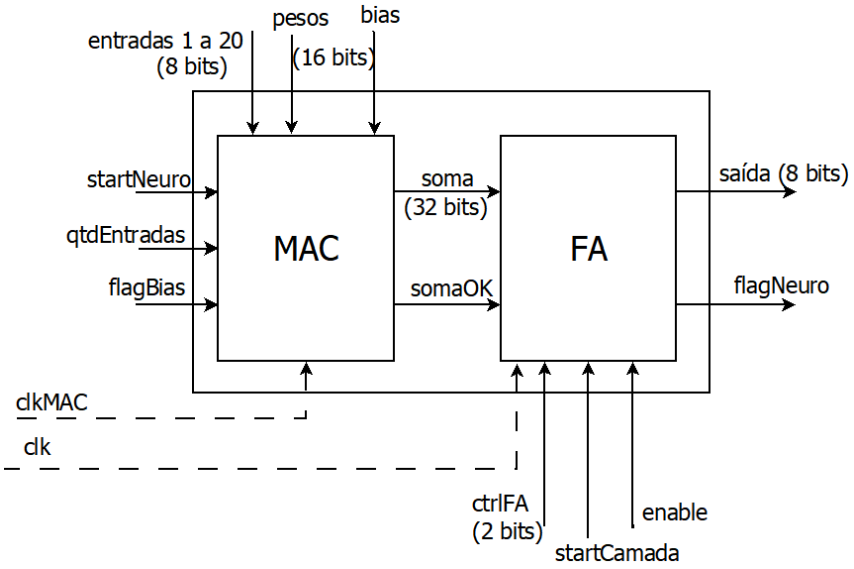

Figura 39 – Neurônio

Os sinais de entrada, pesos e *bias* são enviados diretamente ao bloco MAC, além do sinal *clkMAC* e do sinal de *flagBias*. Também, os sinais de controle *startNeuro* e *qtdEntradas* são enviados a este bloco. Os demais sinais de controle são enviados ao bloco FA.

Os blocos MAC e FA serão detalhados a seguir. Em seguida, a carta de tempo da execução do neurônio será mostrada.

3.4.2.1 MAC

O MAC ou Multiplicador-Acumulador é o bloco responsável por multiplicar as entradas pelos pesos, somar os produtos parciais e o *bias* para que o resultado seja enviado à função de ativação, que em seguida definirá a saída do neurônio. Assim, este é um bloco formado por, basicamente, multiplicadores, somadores e um controlador.

Na implementação de um neurônio totalmente paralelo, haveria um multiplicador para cada entrada, porém, em Redes Neurais Artificiais digitais em hardware, deve-se ter atenção quanto ao tamanho do bloco multiplicador, já que este tem, em geral, a maior área dentre os blocos que compõem o circuito. Para contornar esse problema, foi decidido usar somente 2 multiplicadores de maneira paralela. Esta abordagem diminui a área usada para o módulo MAC, mas aumenta o tempo de execução do circuito, já que as multiplicações não serão mais feitas de maneira totalmente paralela. Em casos em que a aplicação que a RNA opera não necessita de execução em tempo real, esse aumento no tempo de execução não é limitante.

O modelo do MAC implementado é mostrado na Figura 40.

O Controlador MAC mostrado na Figura 40 é uma máquina de estados responsável

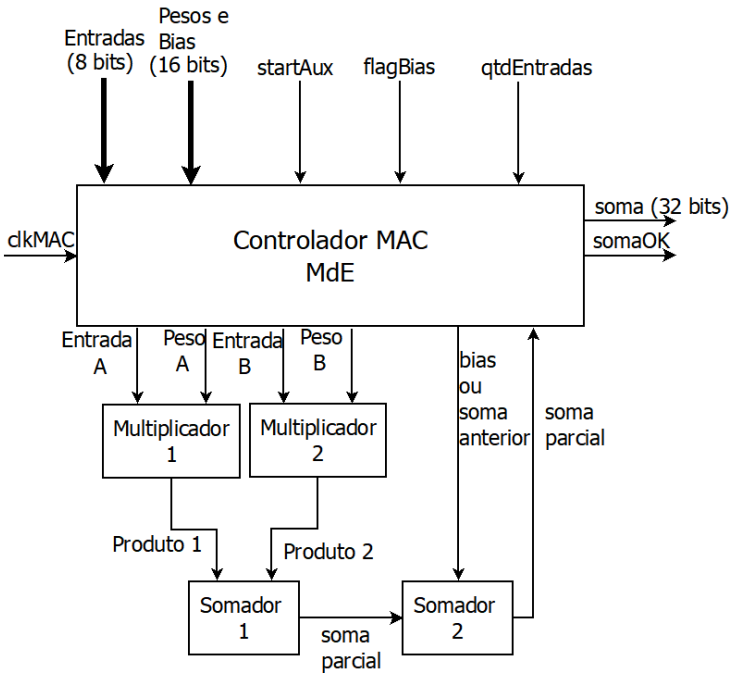

Figura 40 – MAC

por controlar as entradas sequenciais dos multiplicadores e a soma dos resultados parciais, além de disponibilizar a saída para o neurônio. Seu diagrama de estados é detalhado na Figura 41.

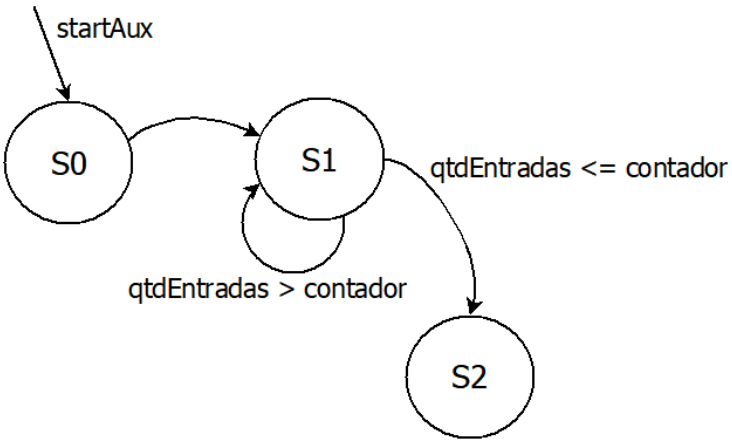

Figura 41 – MdE do MAC

A função geral da MdE é garantir que todas as entradas do neurônio serão multiplicadas pelos seus pesos e somadas de forma correta. A descrição de cada estado e do *startAux* é vista a seguir:

startAux: é o sinal responsável por iniciar o processamento do MAC. Este sinal é definido como uma função lógica entre os sinais *startNeuro* (que indica o início da execução do neurônio) e *startCamada* (que indica a execução da camada). A função lógica entre esses sinais garante que o MAC vai para o estado S0 sempre que o neurônio começar a executar ou quando a camada terminar sua execução. Nesta etapa, os registradores usados

para o cálculo são zerados. Se o *flag* de *bias* estiver ativo, um registrador responsável por armazenar o resultado parcial das somas é carregado com o valor do *bias*; senão, este registrador é carregado com o valor 0. Também um contador que guarda qual entrada está sendo executada é zerado. A máquina de estados é então levada ao estado S0.

S0: neste estado os dois multiplicadores recebem as duas primeiras entradas e seus respectivos pesos para que a primeira multiplicação seja feita. O sinal do contador é então incrementado por 2. A máquina fica neste estado durante um ciclo do *clkMAC* e depois passa para o estado S1.

S1: o estado S1 é responsável por receber o resultado da soma das duas multiplicações anteriores e adicioná-lo com o valor do registrador do *bias*. Para facilitar o entendimento do funcionamento deste estado, seu fluxo de operação é mostrado na Figura 42. Este estado inicia sua operação colocando os valores de duas entradas da camada (sinais x) e dos pesos respectivos (sinais w) nas entradas dos Multiplicadores (entradas A e B). Vale notar que é usado um sinal de contagem para fazer essa atribuição. Em seguida, o valor da soma dos produtos anteriores é recebido e somado ao valor parcial da saída. Uma verificação é então feita: se o valor da quantidade de entradas da camada for maior que o valor do contador diminuído de 1, esta última é incrementada por 2 e a MdE volta ao início do estado S1; caso contrário, a MdE vai para o estado S2, a fim de finalizar sua execução. Eata verificação garante que todas as entradas da camada serão multiplicadas e que todos os produtos serão recebidos.

S2: este é o estado final da MdE e é responsável por disponibilizar a saída (*soma*) e ativar um *flag* (*somaOK*) que indica ao neurônio que o MAC já terminou sua execução.

A máquina de estados que controla o MAC deve garantir que cada multiplicação ocorra corretamente dentro de um estado, ou seja, dentro de um ciclo de *clkMAC*. Como a multiplicação é uma das operações mais lentas dentro de uma Rede Neural Artificial, escolheu-se usar um *clock* específico para o MAC, mais lento que o usado pelo restante do sistema. Assim, garante-se a correta execução da multiplicação, sem comprometer o restante da Rede Neural Artificial. Este *clkMAC* tem um período 3 vezes maior que o *clk* e é disponibilizado pela Unidade de Clock.

Além da MdE, o MAC também é formado pelo bloco SomaOverflow, que faz a soma dos resultados parciais e também o controle de *overflow*, e pelo bloco Multiplicador.

O bloco SomaOverflow faz uma verificação de *overflow*, já que as entradas estão em complemento de dois. Seu código é mostrado na Figura 43.

O bloco somador foi definido como uma *function*, para que pudesse ser usado dentro de um bloco *always* e facilitasse a leitura do código. Este bloco recebe duas entradas de 32 bits (Linha 2) e tem como saída um sinal também de 32 bits (Linha 1). Na linha 6 os dois sinais de entradas são somados. Então é feita uma verificação: se as duas entradas são

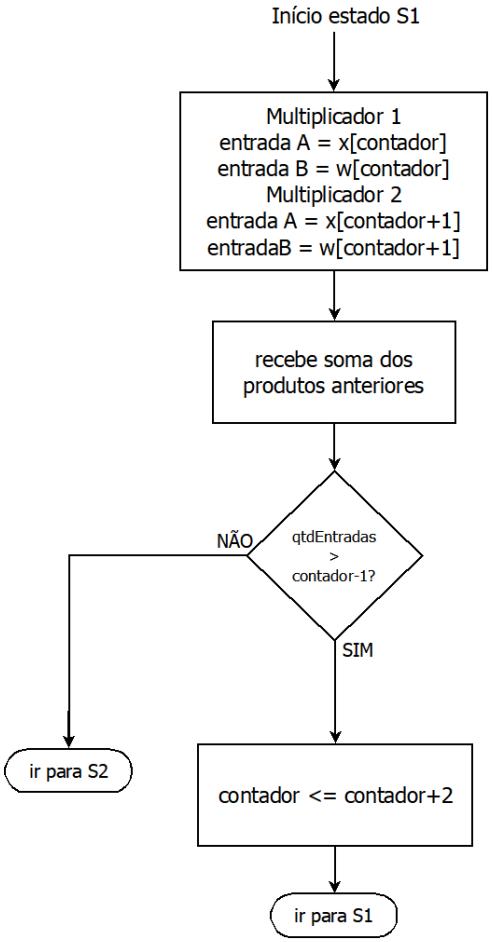

Figura 42 – Fluxo do estado S1 do MAC

```
1  function [31:0] SomaOverflow;
2     input [31:0] iA, iB;
3
4     reg [31:0] aux;
5     begin
6        aux = iA + iB;
7
8        if (iA[31] == 1'b1 & iB[31] == 1'b1 & aux[31] == 1'b0)
9           aux = 32'b10000000000000000000000000000000;
10       else if (iA[31] == 1'b0 & iB[31] == 1'b0 & aux[31] == 1'b1)
11          aux = 32'b01111111111111111111111111111111;
12
13       SomaOverflow = aux;
14    end
15 endfunction
```

Figura 43 – Código da Soma com Overflow

negativas (tem bit 31 igual a 1) e a soma é positiva (bit 31 igual a 0), então a saída recebe o maior valor negativo possível (Linhas 8 e 9); caso as duas entradas forem positivas (bit 31 igual a 0) e a soma for negativa (bit 31 igual a 1), então a saída recebe o maior valor positivo. Isto garante o controle de *overflow* durante o uso do MAC. As entradas deste bloco são os produtos vindos dos multiplicadores.

O bloco mais importante do MAC é o multiplicador. A FPGA disponibiliza blocos multiplicadores em seu hardware, mas para evitar a dependência do modelo da FPGA e permitir uma futura implementação em CI, uma topologia de multiplicador diferente da disponível pela FPGA foi utilizada. Esta abordagem traz uma maior área na implementação, porém este gasto de área é evitado ao se usar somente 2 multiplicadores por neurônio, cada um usado 10 vezes no máximo, dependendo da quantidade de entradas, como dito anteriormente. O modelo de multiplicador escolhido foi o Vedic, pois se mostrou eficiente em termos de potência dissipada e tempo de execução (Vide Revisão Bibliográfica).

O multiplicador do tipo Vedic trabalha somente com números positivos, exigindo uma adaptação para este trabalho, já que foi usada uma notação em complemento de dois. O fluxo do modelo adotado para garantir o funcionamento correto mesmo com números negativos é mostrado na Figura 44.

Como os pesos são de 16 bits, as entradas de 8 bits são estendidas para 16 bits, levando em conta a notação em ponto fixo. O modelo pode ser visto a seguir:

$$entrada(8\ bits) = i7\ i6\ i5,\ i4\ i3\ i2\ i1\ i0$$

$$entrada(16\ bits) = i7\ i7\ i7\ i7\ i6\ i5,\ i4\ i3\ i2\ i1\ i0\ 0\ 0\ 0\ 0\ 0$$

O ajuste é feito de modo que o valor de 16 bits coincida com o valor de 8 bits. Assim, são colocados 5 bits 0 no final da entrada de 8 bits, fazendo com que esta tenha 10 bits na parte decimal. Para a parte inteira, é importante que o valor do sinal não se perca, assim, são colocados 3 bits de valor igual ao bit 7 da entrada, garantindo que o sinal de 16 bits tenha 6 bits na parte inteira e que este valor tenha o mesmo sinal que a entrada de 8 bits.

Após este ajuste, se as entradas ou pesos forem negativos, é feito um complemento de dois. Os sinais já ajustados são colocados na entrada do multiplicador para que a operação possa ser executada. Assim que a multiplicação for concluída, é feito o complemento de dois da saída, se necessário. Para verificar se é necessário o complemento de dois da saída, é a função XOR é aplicada entre os bits de sinal das entradas. Este pequeno ajuste garante um correto funcionamento mesmo para sinais em complemento de dois (90).

Vários modelos de multiplicadores Vedic já foram propostos na literatura (80, 90, 91, 94). O modelo escolhido melhora o tempo de execução ao transformar o multiplicador de 4 bits na menor célula do sistema, sendo implementando somente com blocos lógicos e meio somadores (80).

No trabalho em que essa implementação se baseou (80), o multiplicador proposto tem entradas de 8 bits. Como, para a implementação da Rede Neural Artificial é necessário um multiplicador com entradas de 16 bits, o projeto inicial foi estendido. A estrutura regular do multiplicador Vedic permite que esta extensão seja simples de ser implementada.

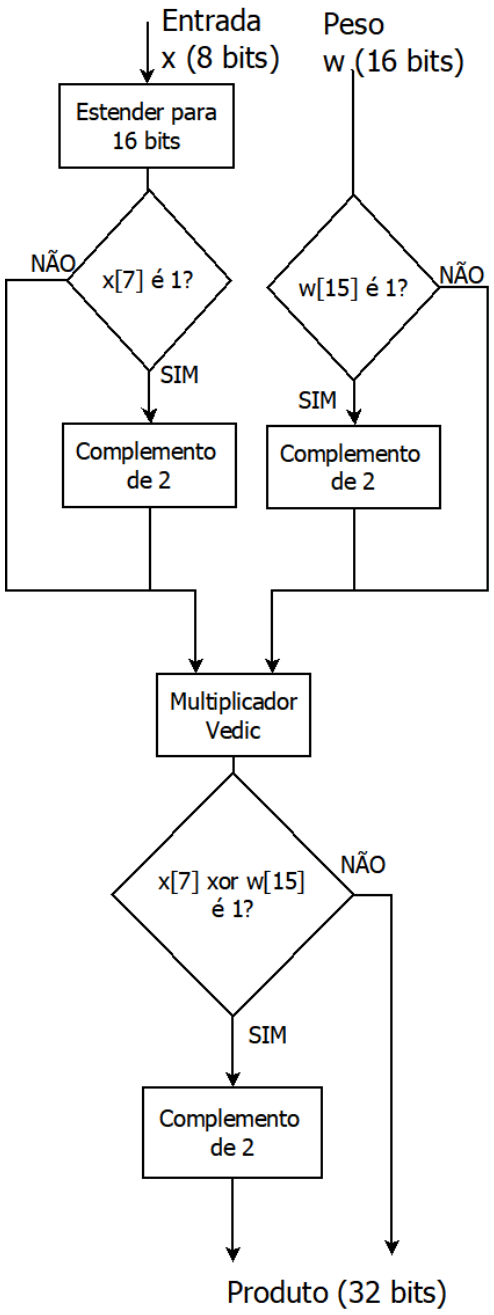

Figura 44 – Multiplicador com Complemento de 2

Como dito, o multiplicador de 4 bits é o menor bloco do multiplicador implementado. Sua estrutura é mostrada nas Figuras 45 e 46. Nestas figuras, os blocos *HA* indicam somadores completos.

Somente portas lógicas e somadores de 1 bit são usados para calcular o produto entre duas entradas de 4 bits. Este multiplicador é baseado no somador do tipo *Carry Skip*. Cada bit da saída é analisado separadamente. Se algum estágio produz um *carry* de 2 bits, então o estágio imediato é pulado e o *carry* é passado para os estágios seguintes, aumentando a velocidade de processamento.

O multiplicador de 4 bits é então usado para criar o multiplicador de 8 bits, como

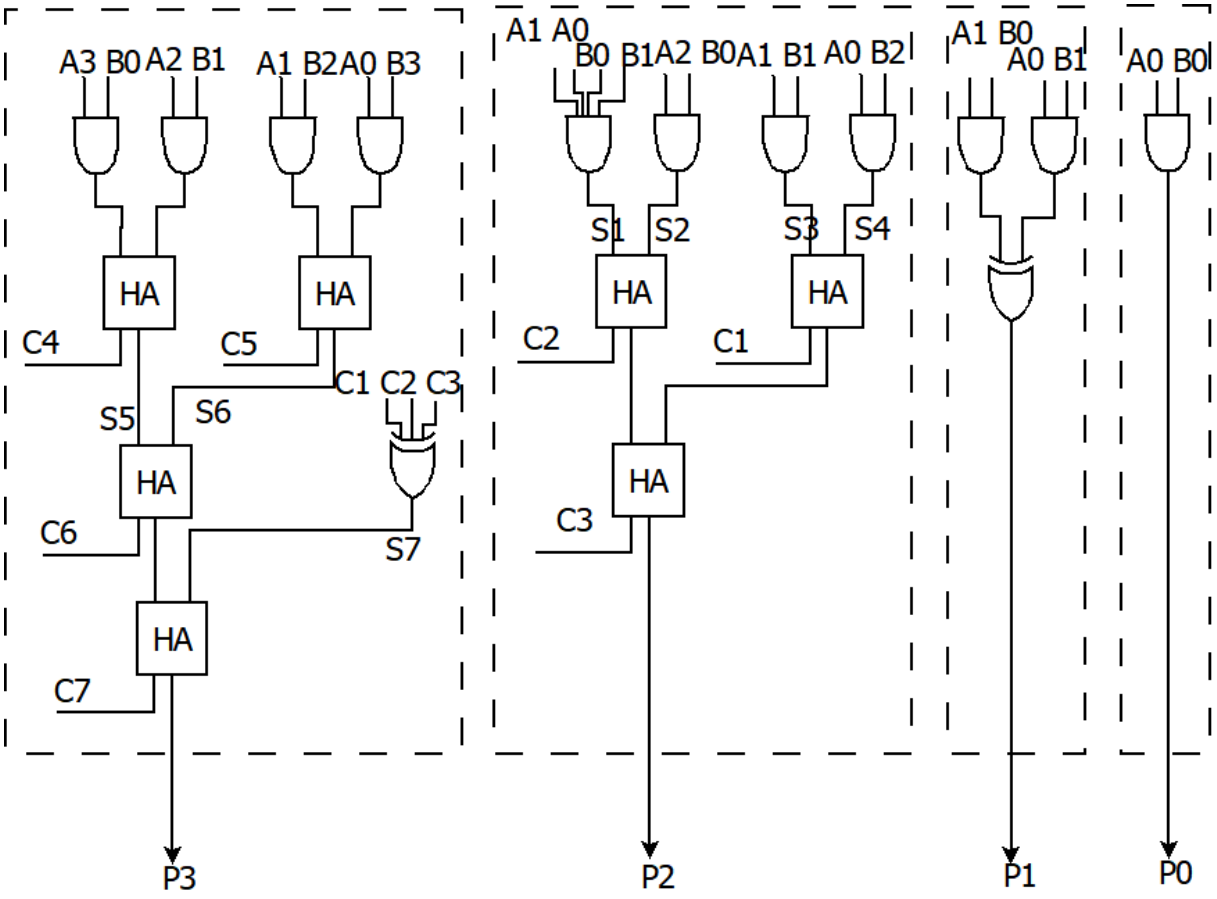

Figura 45 – Multiplicador de 4 bits - saídas 0 a 3

mostrado na Figura 47.

Nota-se que o multiplicador de 8 bits é formado por 4 multiplicadores de 4 bits em conjunto de 3 somadores de 8 bits. Os somadores também foram definidos de maneira diferente do somador convencional (80). Cada um deles tem uma estrutura própria, para garantir um menor atraso da execução.

Os 4 bits menos significativos da saída (P[3:0]) são definidos pelos 4 bits menos significativos do Multiplicador 1 (m1[3:0]). O resultado dos Multiplicadores 2 e 3 (m2[7:0] e m3[7:0]) são colocados na entrada do Somador 1. Este bloco é definido como um meio somador seguido por 7 somadores completos (Figura 48)

A saída do Somador 1 (m5[7:0]) é enviada ao Somador 2, juntamente com os 4 bits mais significativos do resultado do Multiplicador 1 (m1[7:4]). O Somador 2 é mostrado na Figura 49.

Nem todos os somadores que formam a cadeia do Somador 2 precisam ser completos, já que este bloco soma sinais de 8 e 4 bits. Somente a parte da cadeia que recebe duas entradas precisa ser formada por somadores completos; o restante da cadeia é formado por meio somadores e uma porta XOR.

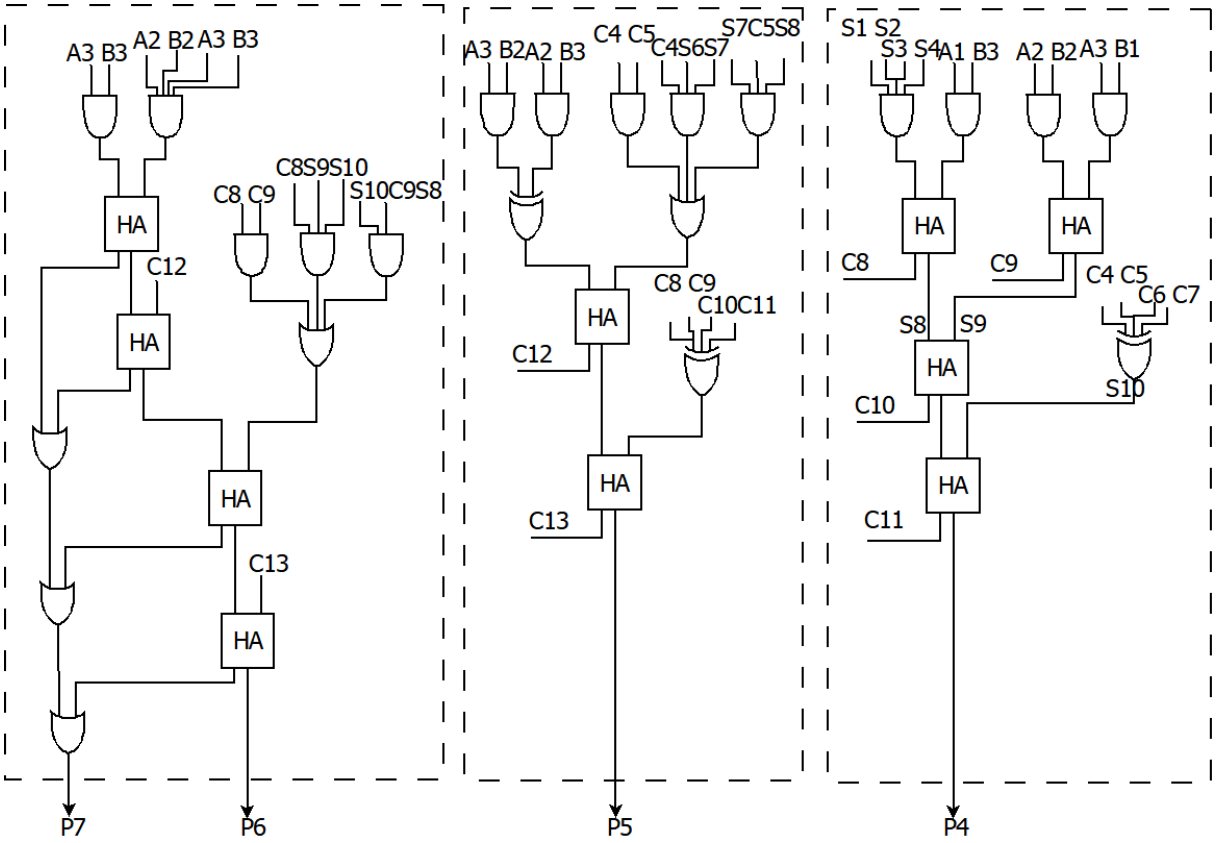

Figura 46 – Multiplicador de 4 bits - saídas 4 a 7

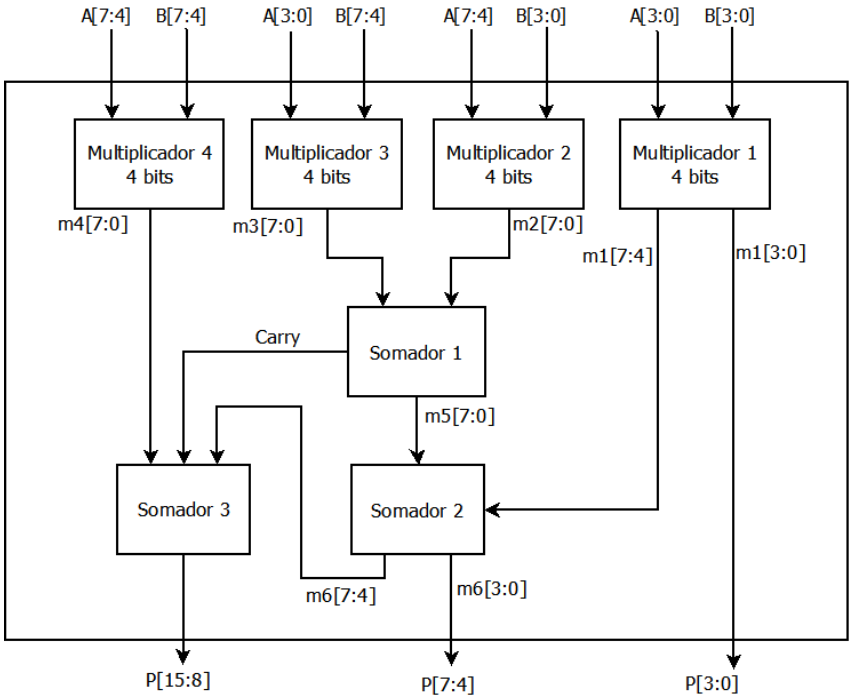

Figura 47 – Multiplicador com entradas de 8 bits

A parte menos significativa do resultado do Somador 2 (m6[3:0]) define os bits P[7:4] da saída do Multiplicador de 8 bits. A parte mais significativa do resultado do Somador 2 (m6[7:4]) é concatenada ao *carry* do Somador 1 e colocado na entrada do

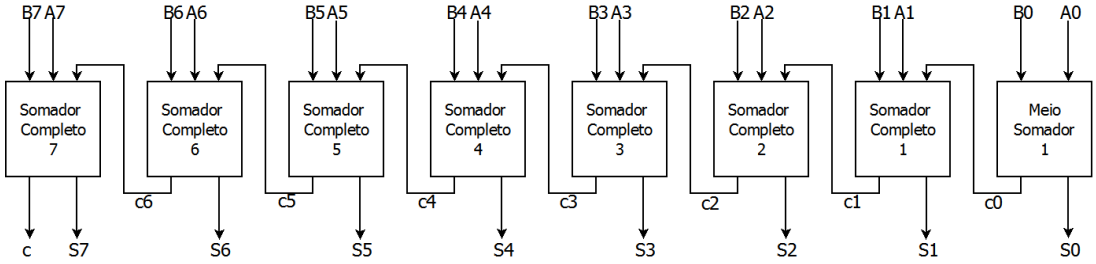

Figura 48 – Somador 1

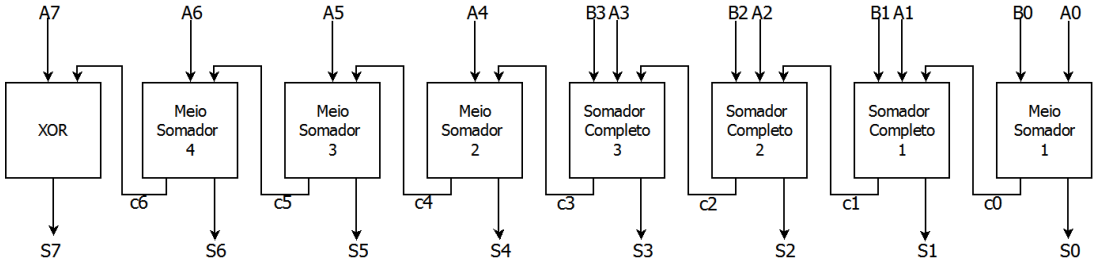

Figura 49 – Somador 2

Somador 3, junto com o resultado do Multiplicador 4 (m4[7:0]). O Somador 3 é então projetado seguindo o mesmo princípio do Somador 2 (Figura 50).

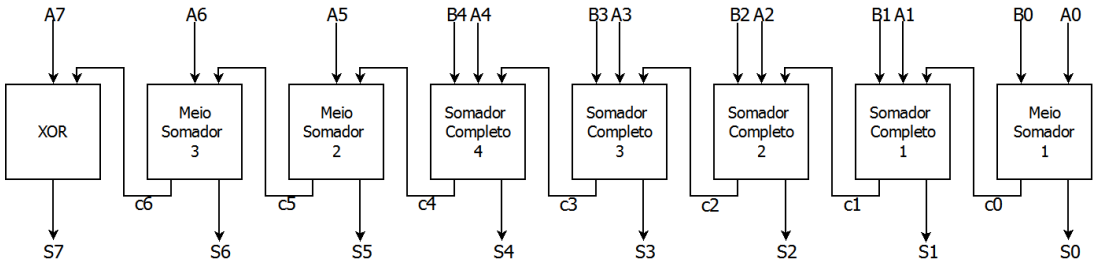

Figura 50 – Somador 3

O Somador 3 é formado por uma cadeia de um meio somador e quatro somadores completos e o restante da cadeia tem meio somadores e uma porta XOR. Novamente, somente a parte da cadeia que recebe duas entradas necessita ser formada por somadores completos. A saída do Somador 3 forma então os 8 bits mais significativos da saída do multiplicador (P[15:0]).

Seguindo a estrutura do multiplicador Vedic de 8 bits, um multiplicador de 16 bits pode ser formado. Este multiplicador pode ser visto na Figura 51.

Esta abordagem permite aumentar o número de bits do multiplicador de maneira simples, com o uso de blocos já definidos anteriormente. O funcionamento do multiplicador de 16 bits é semelhante ao de 8 bits. A diferença está no tamanho das palavras que os somadores têm que lidar, além do uso de multiplicadores de 8 bits para o estágio inicial. Nota-se que o aumento da quantidade de bits nas entradas aumenta o tempo de execução do multiplicador, já que os somadores usados se baseiam em uma cadeia de estruturas e,

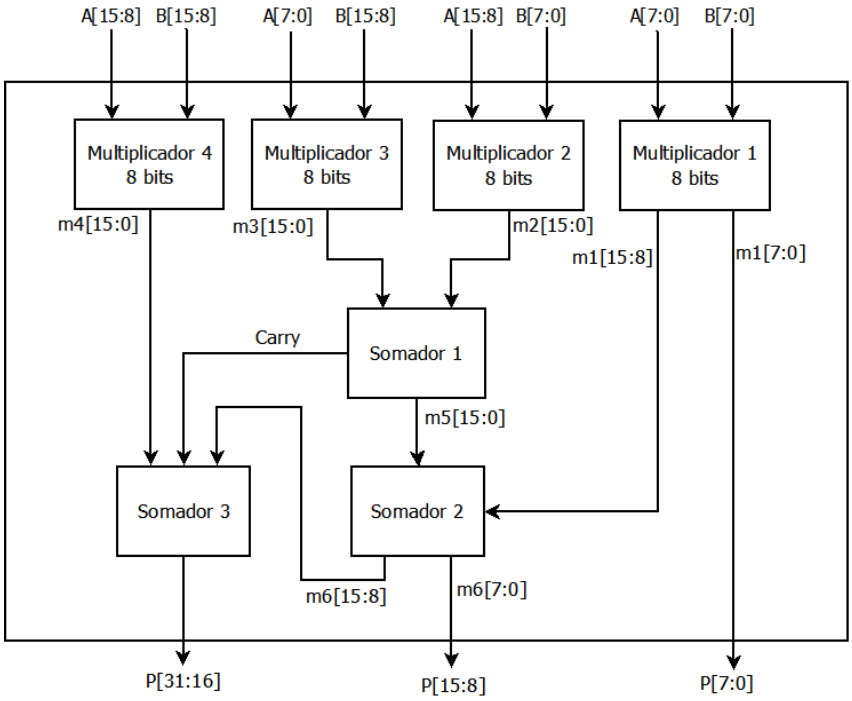

Figura 51 – Multiplicador com entradas de 16 bits

quanto maior a cadeia, maior o atraso da execução. Assim, melhorar os somadores é uma forma de melhorar os atrasos do multiplicador (91).

A área total gasta por um multiplicador de 16 bits é de 788 elementos lógicos. Nenhum bloco interno da FPGA foi necessário, permitindo assim o uso de qualquer modelo de FPGA e, principalmente, permitindo futuras implementações em CIs.

O atraso máximo para a execução do multiplicador é de 35,805 ns, definindo assim o mínimo *clock* para a MdE que executa o multiplicador.

O multiplicador de 16 bits é então colocado na estrutura que garante a execução de dados em complemento de 2, mostrada na Figura 44. Vale lembrar que, no caso da Rede Neural Artificial aqui implementada, uma das entradas do multiplicador tem somente 8 bits. Isto faz com que o software Quartus otimize o circuito, cortando as partes que não são usadas. Assim, a área total do multiplicador de 16 com complemento de 2 é 589 elementos lógicos, que é menor que a área do multiplicador sozinho. O atraso máximo para esta estrutura é 36,484 ns, maior que para o multiplicador sozinho. Isto ocorre, pois, um fluxo maior deve ser percorrido pelos dados de entrada até que o produto correto esteja disponível.

Para mostrar o correto funcionamento do MAC, um teste é analisado. Neste teste, é simulado o caso de 20 entradas a serem multiplicadas. O valor das entradas e os pesos são vistos na Tabela 11.

O *bias* neste teste é 0. Ao somar os produtos entre entradas e pesos o valor da soma final esperada no MAC é igual a 10. A Figura 52 mostra a carta de tempo para este

Tabela 11 – Entradas para o teste do MAC

Entrada	x	w
0	1	-1
1	-1	-2
2	1	-1
3	-1	-2
4	1	-1
5	-1	-2
6	1	-1
7	-1	-2
8	1	-1
9	-1	-2
10	1	-1
11	-1	-2
12	1	-1
13	-1	-2
14	1	-1
15	-1	-2
16	1	-1
17	-1	-2
18	1	-1
19	-1	-2

exemplo. Para facilitar a visualização, os sinais de entrada não são mostrados.

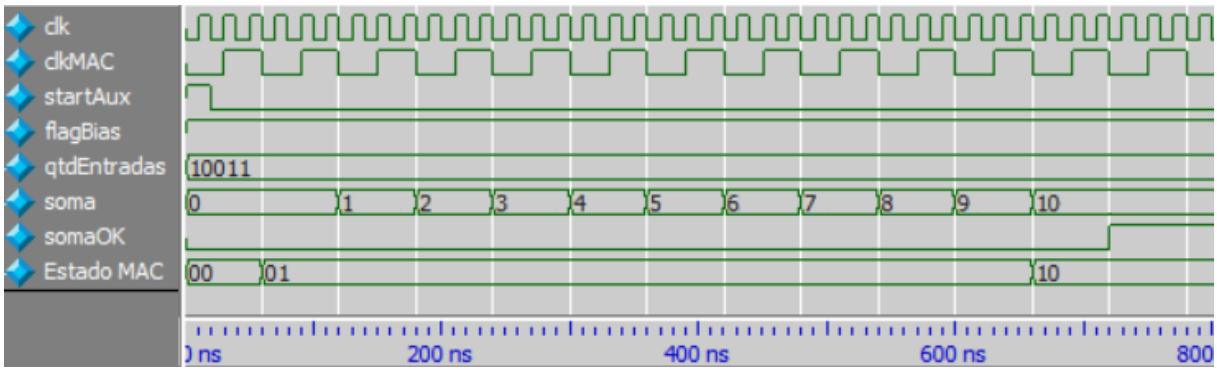

Figura 52 – Carta de Tempo MAC

Na execução da soma ponderada, o *clkMAC* é responsável por controlar o funcionamento da máquina de estados. O *clkMAC* é necessário pois a multiplicação tem um maior tempo de execução e o produto deve estar disponível em um ciclo de *clock*. Para não afetar todo o funcionamento da RNA com um *clock* mais lento, escolheu-se usar os dois sinais. O *clock* original tem período de 20ns e, portanto, o *clkMAC* tem período de 60ns.

Na carta de tempo, pode-se notar que assim que o sinal *startAux* vai a 1 a MdE vai para o estado S0. Na borda de descida do *clkMAC* seguinte, a máquina passa ao

estado S1 e se mantém nesse estado por 10 ciclos de *clkMAC*, um para cada dupla de entradas. Nota-se que a cada borda de descida o valor da saída é atualizado, mostrando que as multiplicações e somas estão sendo feitas. Quando todas as entradas já foram multiplicadas e somadas, a MdE parte para o estado S2 e, no ciclo seguinte, o *flag* de saída vai a 1, indicando o fim da execução do MAC. O resultado obtido nessa simulação é 10, valor correspondente ao esperado.

A quantidade de estados que esta MdE deve passar até que o resultado da soma ponderada esteja disponível depende da quantidade de entradas do neurônio. Cada entrada deve ser multiplicada e somada para gerar o resultado final. Cada multiplicação leva 1 ciclo de *clkMAC* para ficar pronta e é necessário 1 ciclo para iniciar o funcionamento do MAC e mais 1 ciclo de *clkMAC* para finalizar a soma ponderada. Vale lembrar que as multiplicações são calculadas duas a duas, diminuindo o tempo de execução deste bloco pela metade. Também, *clkMAC* é 3 vezes maior que o *clk*. Assim:

$$\text{ciclos execução camada } l_i = 3\left(2 + arr\left(\frac{e_i}{2}\right)\right) \tag{3.3}$$

Onde a função $arr\left(\frac{e_i}{2}\right)$ arredonda para cima o resultado da divisão da quantidade de entradas e_i por dois.

Tomando o exemplo aqui mostrado, a quantidade de ciclos necessárias para esta camada é $3\left(2 + arr\left(\frac{20}{2}\right)\right) = 36$ ciclos de *clk*, resultado igual ao observado na simulação.

Porém, quando a camada a ser executada não é a primeira, é necessário um ciclo de *clock* para que a MdE do MAC volte ao estado S0, aumentando o número de ciclos de *clock*. Assim:

$$\text{ciclos execução camada } l_i = \begin{cases} 3\left(2 + arr\left(\frac{e_i}{2}\right)\right), & i = 1 \\ 1 + 3\left(2 + arr\left(\frac{e_i}{2}\right)\right), & i \neq 1 \end{cases} \tag{3.4}$$

Assim que a saída do MAC fica disponível, ela e o *flag* são enviados ao bloco função de ativação, onde a saída do neurônio é calculada e finalizada.

É importante frisar que este valor calculado indica a quantidade mínima de ciclos de *clock* usados para o cálculo do MAC. Este valor pode aumentar em 1 ou 2 ciclos de *clock* devido ao uso de dois *clocks* diferentes no sistema. Isto pode levar a uma certa "perda de sincronia" entre o *clk* e o *clkMAC*. Por exemplo, pode ocorrer que, durante a execução total da RNA, o início do MAC não se dê exatamente na borda de descida do *clkMAC*, levando a uma diferença de até 2 ciclos de *clk* para que a MdE do MAC possa realmente começar sua execução.

3.4.2.2 FA

Este módulo é responsável por executar as quatro funções de ativação possíveis para o neurônio, além de disponibilizar a saída, de acordo com o sinal de controle que escolhe qual função deve defini-la.

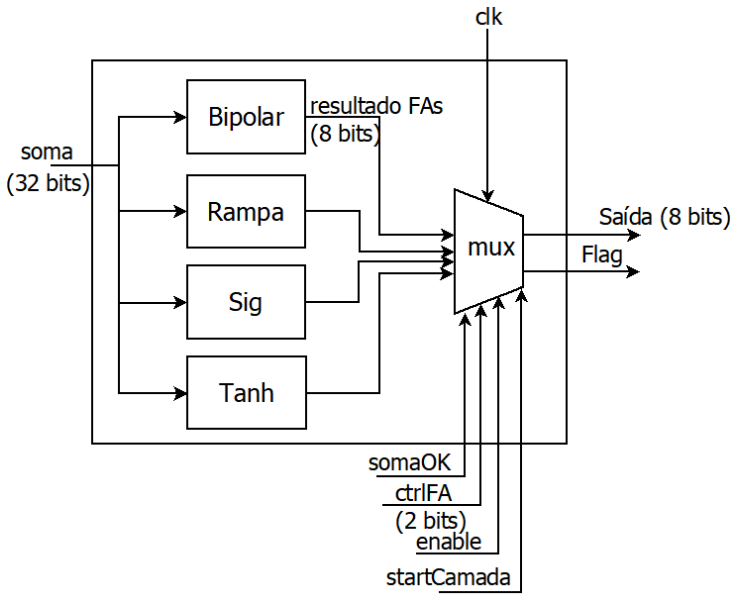

Figura 53 – FAs

Como visto na Figura 53, este bloco recebe como entrada a saída do MAC com 32 bits (*soma*) e um *flag* indicando que o MAC já terminou sua execução (*somaOK*), além dos sinais de controle *ctrlFA*, *startCamada* e *enable* e um sinal de *clock*. Este módulo é formado por 4 tipos de funções de ativação diferentes (função Degrau Bipolar, função Rampa Limitada, função Sigmóide Logística e função Tangente Hiperbólica) e um multiplexador (*mux*), responsável por escolher e disponibilizar a saída do neurônio.

Todas as funções são definidas com 32 bits de entrada (vindos do MAC) e 8 bits de saída. A definição da quantidade de bits de saída é feita levando em consideração o fato de que elas serão realimentadas na entrada da camada, ou seja, o número de bits da saída deve ser igual ao número de bits da entrada da RNA. Os resultados das simulações das funções de ativação foram enviados ao MATLAB para que os gráficos pudessem ser plotados.

As duas primeiras funções são lineares, sendo de fácil implementação. O código para a função Degrau Bipolar pode ser visto na Figura 54.

Esta função é implementada através de um simples comparador e atribuições. Como os sinais são representados em complemento de 2, basta verificar se o bit mais significativo é 0 ou 1. Se for 0 (linha 10), a saída recebe o valor 1 (linha 11); se for 1 (linha 8), a saída recebe o valor -1 (linha 9).

```
1  module degrau_bipolar(soma, saidaFA);
2
3     input signed [31:0] soma;
4     output reg [7:0] saidaFA;
5
6     always@(soma)
7     begin
8        if (soma[31]) // se menor que 0, saída é -1, se maior que 0 é 1.
9           saidaFA = 8'b11100000; //-1
10       else
11          saidaFA = 8'b00100000; // 1
12    end
13 endmodule
```

Figura 54 – Código da Função Degrau Bipolar

A curva desta função pode ser vista na Figura 55:

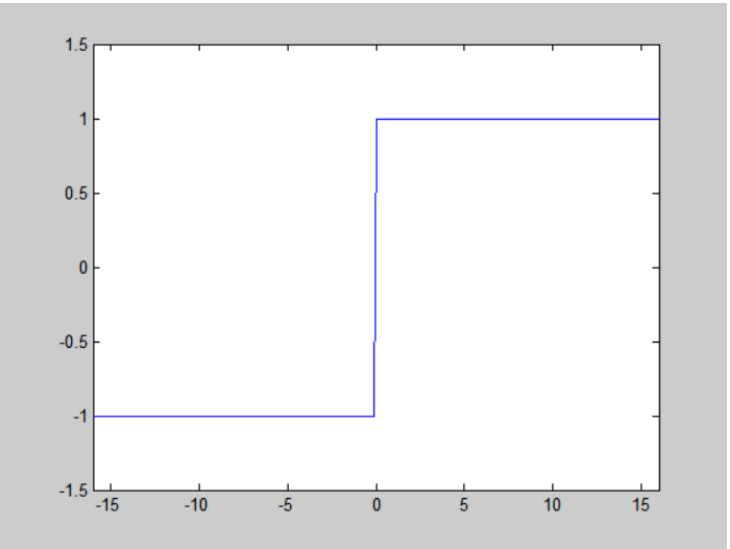

Figura 55 – Função Degrau Bipolar Implementada

A segunda função implementada é a função Rampa, que é limitada entre -4 e 3,96875, valores limite da representação de números em complemento de 2 com ponto fixo de 8 bits escolhida para este trabalho. Assim, se a entrada estiver entre -4 e 3,96875, a saída é igual à entrada. Caso contrário, a saída recebe -4 ou 3,96875. O código implementado pode ser visto na Figura 56.

Este bloco, assim como o bloco do Degrau Bipolar, é formado basicamente por comparações e atribuições. Na linha 9 a entrada é comparada com -4. Se ela for menor ou igual a esse valor, a saída recebe o valor -4 (linha 10). Na linha 12, a entrada é comparada com 3,96875: se for maior ou igual a esse valor, a saída recebe 3,96875, que é o valor de limite superior da representação (linha 13). Se a entrada estiver entre esses dois limites, a saída deve receber o valor da entrada. Para isso, um ajuste deve ser feito para garantir uma saída correta: a partir do local da vírgula no sinal de 32 bits, conta-se 3 bits para a esquerda, que representam a parte inteira da saída e 5 bits para a direita, que

```
1  module rampa(soma, saidaFA);
2
3     input signed [31:0] soma;
4     output reg [7:0] saidaFA;
5
6     always@(soma)
7     begin
8        // menor que -4
9        if (iA <= $signed(32'b111111111100_00000000000000000000))
10          oS = 8'b10000000;
11       // maior que 3,96875
12       else if (iA >= $signed(32'b000000000100_00000000000000000000))
13          oS = 8'b01111111;
14       // entre -4 e 3,96875
15       else
16          oS = iA[22:15];
17    end
18 endmodule
```

Figura 56 – Código da Função Rampa Limitada

representam a parte decimal. Assim, o sinal de saída recebe os bits 15 a 22 da entrada, que representam os valores dentro do intervalo para palavras de 8 bits (Linha 16). Neste tipo de implementação há uma perda na exatidão, pois a quantidade de bits da saída da função de ativação é menor que a quantidade de bits da entrada.

A curva característica que resulta da simulação deste bloco pode ser vista na Figura 57.

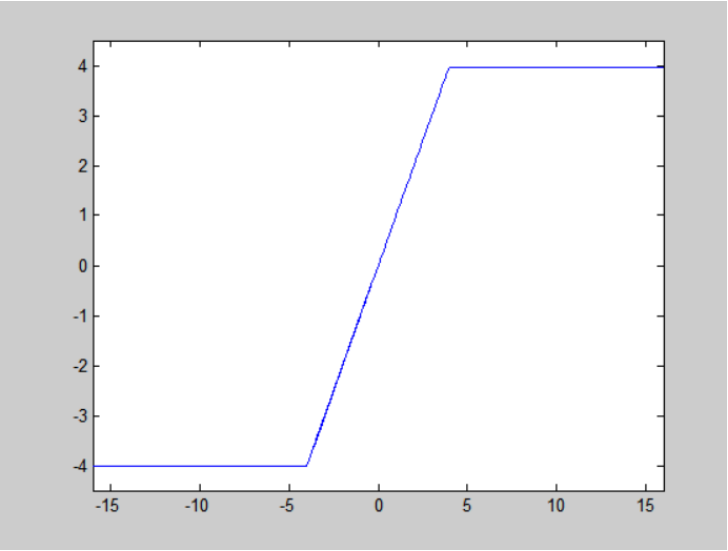

Figura 57 – Função Rampa Limitada Implementada

Por outro lado, as funções Sigmóide Logística e Tangente Hiperbólica, que são funções não lineares, são de difícil implementação em sistemas digitais. Na Seção 2.4 foram mostradas algumas implementações de funções não lineares. A aproximação do tipo PLAN (70) se mostrou superior a outras aproximações pois não usa blocos proprietários

da FPGA, tem menores erros e maior frequência de operação.

A mesma abordagem de aproximação foi usada para as funções Sigmóide Logística e Tangente Hiperbólica e, portanto, o processo de desenvolvimento será mostrado simultaneamente. Somente a parte com entradas positivas das funções é considerada, já que ambas são simétricas. O resultado das funções para entradas negativas é calculado através de complemento de 2 e subtrações.

O primeiro passo para a aproximação é dividir a curva em 4 segmentos. Esta divisão é feita levando em conta retas que melhor se ajustam à curva. Este ajuste também é pensado de forma que o coeficiente angular da reta possa ser substituído por deslocamentos, ou seja, o coeficiente angular deve ser representado por um número na forma 2^n. O coeficiente linear também deve ser representado por um número em ponto fixo, complemento de 2 com até 32 bits.

As relações que definem a parte positiva da aproximação da função Sigmóide Logística são mostradas na Equação 3.5.

$$y = \begin{cases} 0,25x + 0,5, & 0 \leq x < 1 \\ 0,125x + 0,625, & 1 \leq x < 2,5 \\ 0,03125x + 0,859375, & 2,5 \leq x < 4,5 \\ 1, & x \geq 4,5 \end{cases} \tag{3.5}$$

A parte com entradas negativas é obtida através da relação

$$y(-x) = 1 - y(x) \tag{3.6}$$

O resultado dessa adaptação pode ser visto na Figura 58.

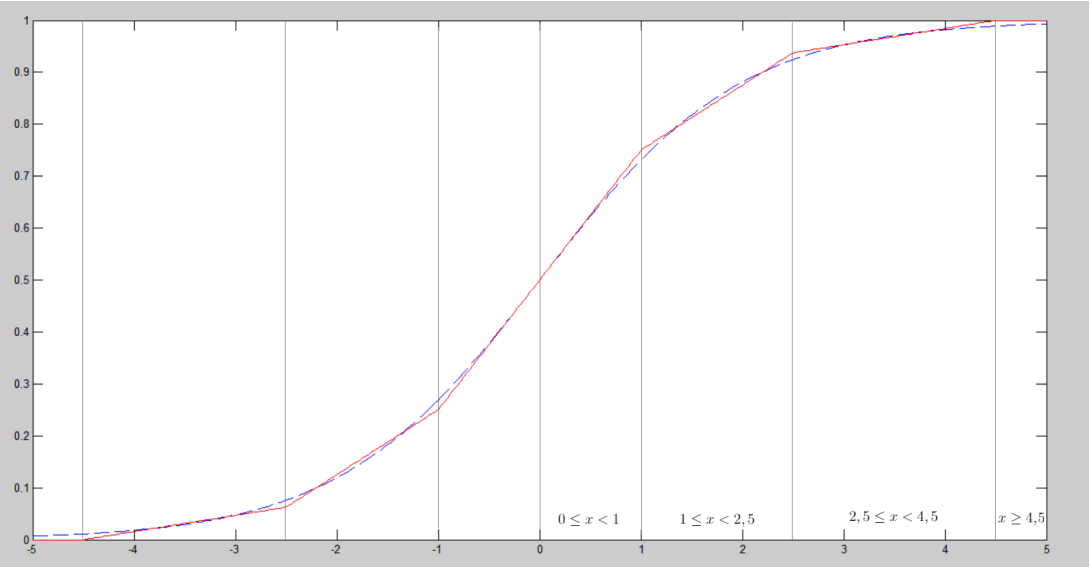

Figura 58 – Aproximação linear esperada da Função Sigmóide Logística

O erro médio esperado para a aproximação da função Sigmóide Logística é de 8,462511E-006 e pode ser visto na Figura 59.

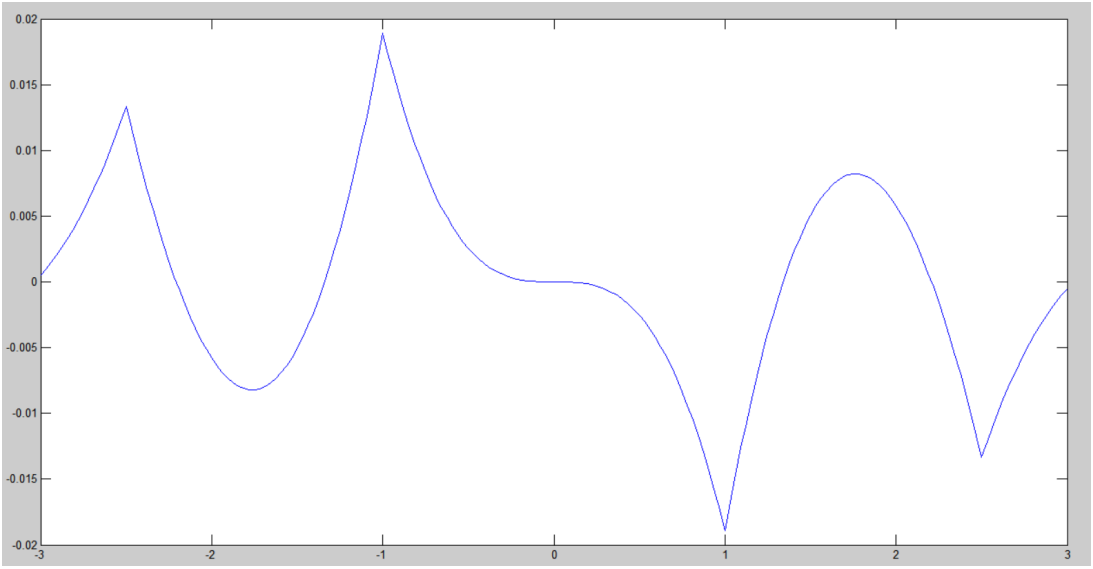

Figura 59 – Erros da aproximação da Função Sigmóide Logística

Pelo gráfico da Figura 59 pode-se observar que os maiores valores de erro são esperados em torno de -1 ou 1.

Já para a função Tangente Hiperbólica as equações são:

$$y = \begin{cases} x, & 0 \leq x < 0,5 \\ 0,5x + 0,25, & 0,5 \leq x < 1 \\ 0,25x + 0,5, & 1 \leq x < 2 \\ 1, & x \geq 2 \end{cases} \tag{3.7}$$

Se as entradas forem negativas, o resultado é obtido através da relação

$$y(-x) = -y(x) \tag{3.8}$$

A Figura 60 mostra o resultado desta aproximação.

O erro médio esperado para esta aproximação é 2,619923E-006 e seu gráfico pode ser visto na Figura 61.

Para esta aproximação são esperados maiores valores de erro em torno de -2 e 2.

Com as equações para a aproximação definidas, pode-se começar a implementação. Vale observar que ambas aproximações se encaixam bem à curva, então é de se esperar que os valores para os erros sejam pequenos.

Para as aproximações da função sigmóide, cada segmento da aproximação é analisado e implementado separadamente. Esta implementação é feita através de blocos lógicos por meio de uma transformação direta, ou seja, os resultados possíveis são analisados e colocados diretamente na saída. Como exemplo, o primeiro intervalo da função Sigmóide Logística é analisado. A equação para este intervalo é 0,25x+0,5 e ele é válido para entradas entre 0 e 1.

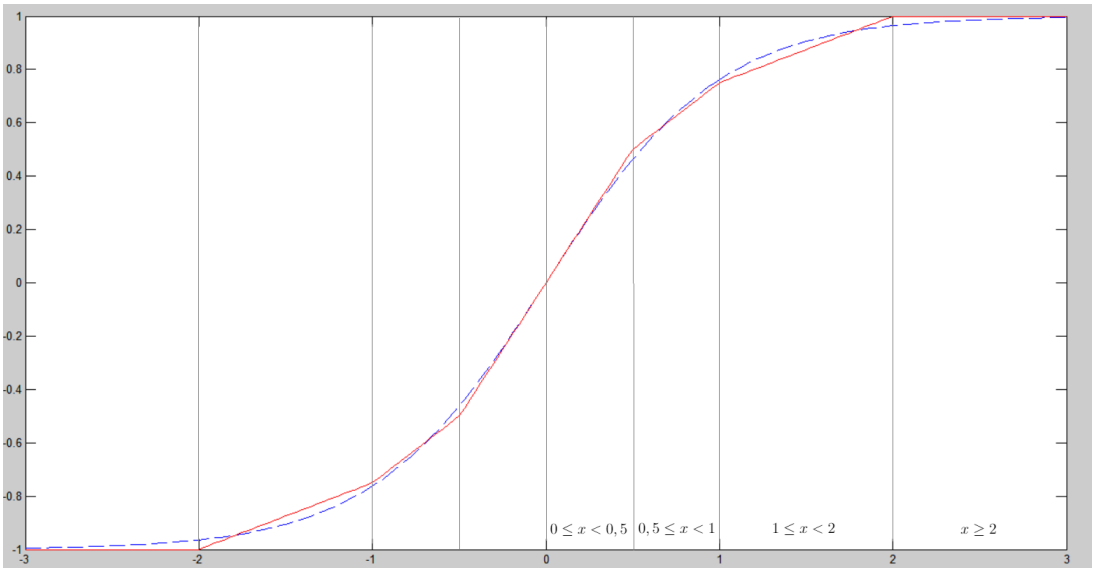

Figura 60 – Aproximação linear esperada da Função Tangente Hiperbólica

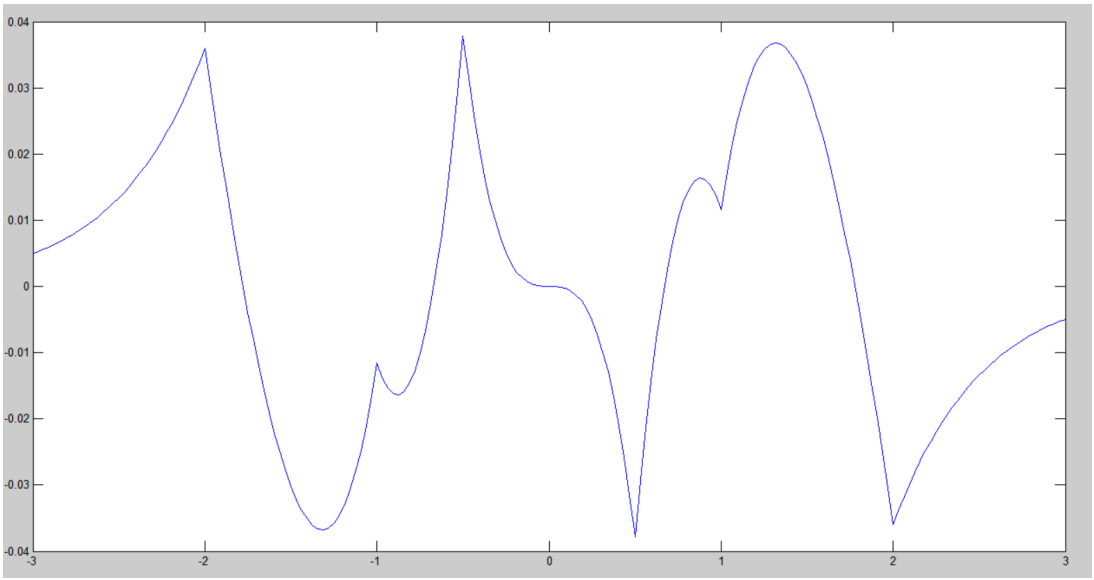

Figura 61 – Erros da aproximação da Função Tangente Hiperbólica

O sinal de entrada de 32 bits é definido como:

$$x_{31}x_{30}x_{29}x_{28}x_{27}x_{26}x_{25}x_{24}x_{23}x_{22}x_{21}x_{20},\ x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2x_1x_0$$

O menor valor possível para este intervalo é 0 e o limite superior é o valor imediatamente menor que 1, que nesta notação é representado como 000000000000, 11111111111111111111, ou seja, os bits 20 a 31 da entrada são sempre iguais a 0 neste intervalo. O sinal de entrada é mostrado a seguir:

$$0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0,\ x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2x_1x_0$$

A multiplicação 0,25x pode ser substituída por dois deslocamentos para a direita.

Após dois deslocamentos para a direita, o sinal se torna:

$$0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0,\ 0\ 0 x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2$$

Em seguida deve-se somar 0,5 ao valor obtido acima. Como 0,5 é representado com 32 bits da forma $000000000000, 10000000000000000000$, a adição só afeta o bit imediatamente depois da vírgula. Assim o resultado é para esta equação é:

$$0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0,\ 1\ 0 x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2$$

Como a saída é representada por somente 8 bits, o resultado sinal fica:

$$0\ 0\ 0,\ 10 x_{19}x_{18}x_{17}$$

O restante das equações da Sigmóide Logística e da Tangente Hiperbólica são obtidos com análises semelhantes. O detalhamento de cada intervalo pode ser visto nos Anexos A e B.

Os resultados das aproximações para cada intervalo são mostrados nas Tabelas 12 e 13.

Tabela 12 – Transformação direta para a função Sigmóide Logística

Intervalo / Bit	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
$0 \leq x < 1$	0	0	0	1	0	x_{19}	x_{18}	x_{17}
$1 \leq x < 2,5$	0	0	0	1	1	$\bar{x_{20}}$	x_{19}	x_{18}
$2,5 \leq x < 4,5$	0	0	0	1	1	1	$\bar{x_{21}} + x_{20}$	$\bar{x_{20}}$
$x \geq 4,5$	0	0	1	0	0	0	0	0

Tabela 13 – Transformação direta para a função Tangente Hiperbólica

Intervalo / Bit	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
$0 \leq x < 0,5$	x_{22}	x_{21}	x_{20}	x_{19}	x_{18}	x_{17}	x_{16}	x_{15}
$0,5 \leq x < 1$	0	0	0	1	0	x_{18}	x_{17}	x_{16}
$1 \leq x < 2$	0	0	0	1	1	x_{19}	x_{18}	x_{17}
$x \geq 2$	0	0	1	0	0	0	0	0

Todas as equações mostradas na Tabela 13 podem ser implementadas com portas *NÃO* e *OU*, fazendo com que a implementação tenha uma pequena área.

O esquema da aproximação das funções é mostrado na Figura 62.

Vale lembrar que o bloco "Complemento de 2" na Figura 62 só faz o complemento se a entrada for negativa.

As funções Sigmóide Logística e Tangente Hiperbólica têm o mesmo modo de operação. A diferença está no cálculo da saída, que segue as Tabelas 12 e 13, e no ajuste para a saída, que é feita seguindo as Equações 3.6 e 3.8.

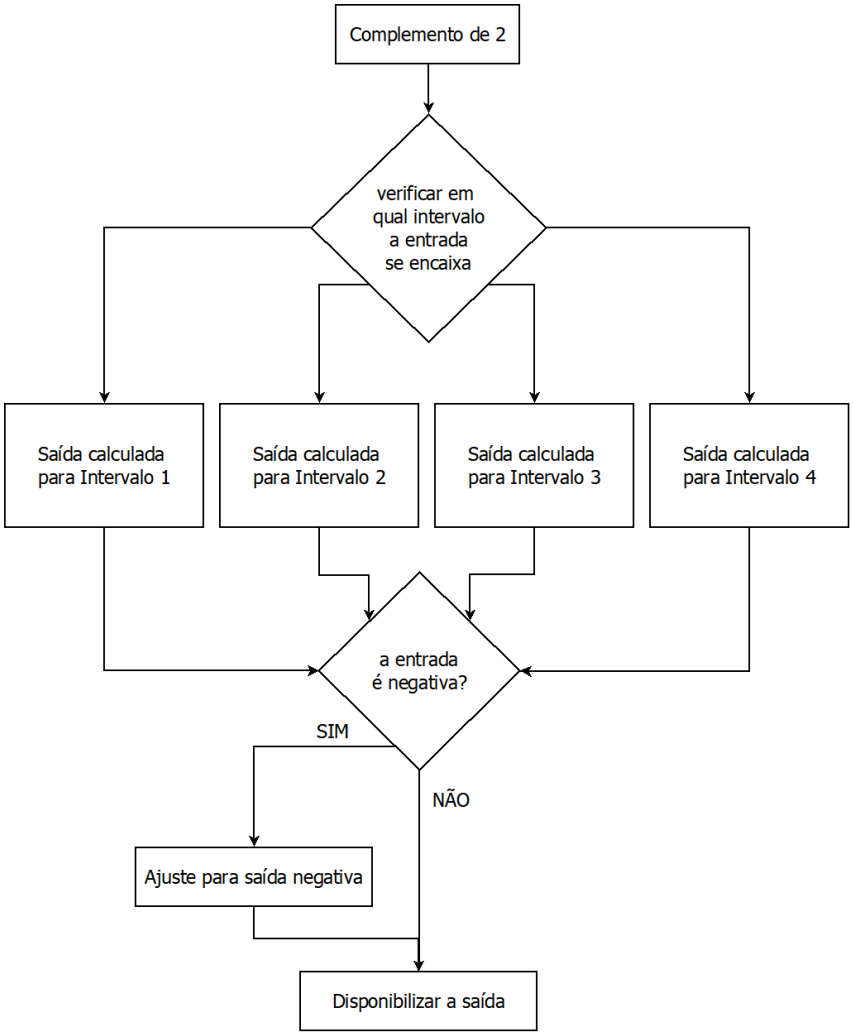

Figura 62 – Fluxo de Dados para a aproximação de Funções não lineares

A Figura 63 mostra o resultado da aproximação da Função Sigmóide Logística e os erros atingidos são mostrados na Tabela 14.

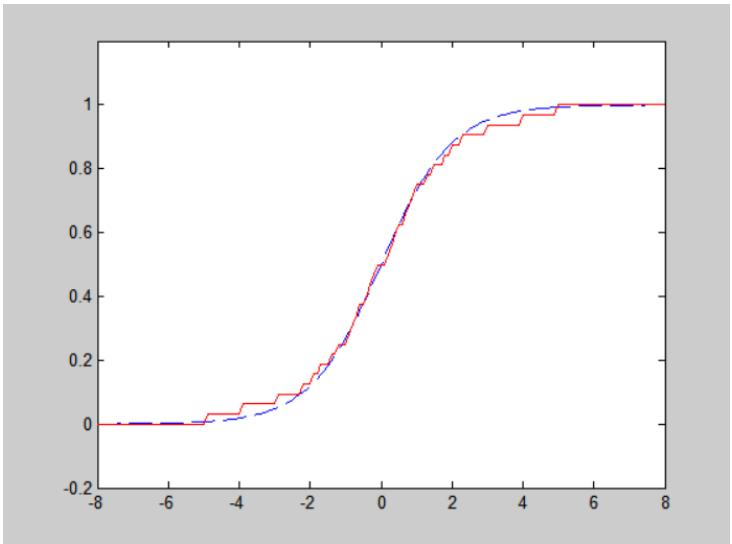

Figura 63 – Resultado da Simulação da Aproximação da Função Sigmóide Logística

Tabela 14 – Erros da aproximação da Sigmóide Logística

	Erro da aproximação em hardware	Erro da aproximação em software
Erro médio	0,0129	8,462511E-6
Erro máximo	0,0427	0,0189

Nota-se tanto pela Figura 63, quanto pela Tabela 14 que os erros atingidos pela aproximação em hardware foram maiores que a aproximação em software. Isto se deve ao fato de que a precisão usada pelo hardware é menor que a usada pelo software, o que leva a maiores erros de representação e uma aproximação com menor exatidão. Vale observar também que a aproximação se distancia mais da curva esperada nos pontos de inflexão, enquanto na parte central, os erros são mínimos.

Os resultados para a Função Tangente Hiperbólica são mostrados a seguir.

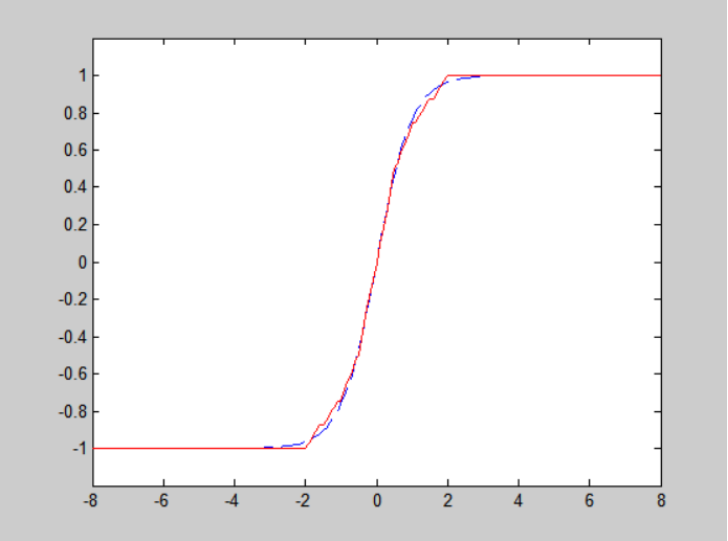

Figura 64 – Resultado da Simulação da Aproximação da Função Tangente Hiperbólica

Tabela 15 – Erros da aproximação da Tangente Hiperbólica

	Erro da aproximação em hardware	Erro da aproximação em software
Erro médio	0,0085	2,619923E-6
Erro máximo	0,0524	0,0379

Na aproximação da Tangente Hiperbólica, os resultados de hardware foram nelhores que na aproximação da função Sigmóide Logística, mas ainda assim há uma diferença entre os resultados em software e hardware devido a notação numérica usada. Vale notar pela Figura 64 que a parte central da aproximação é praticamente exata.

Em geral, os valores atingidos para os erros para as duas funções foram baixos e as aproximações das funções se adequaram bem à forma da curva esperada.

A última parte do bloco FA é o multiplexador que define qual das funções de ativação será levada para a saída do neurônio e pode ser visto na Figura 65.

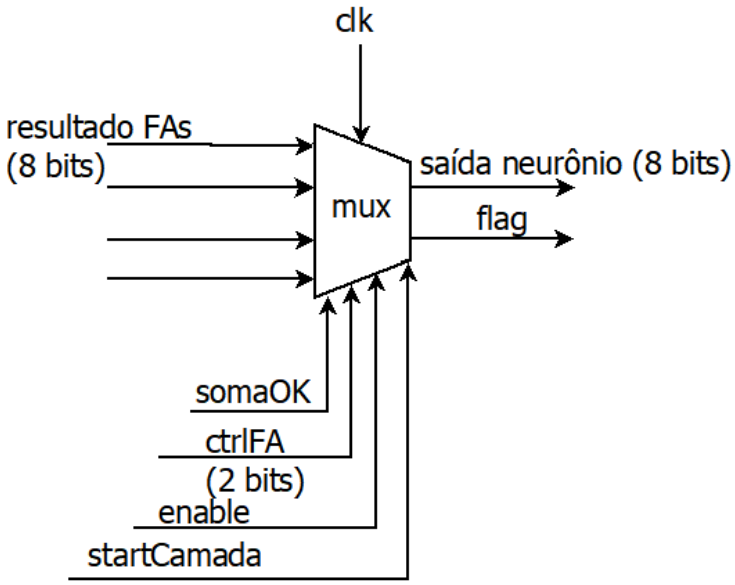

Figura 65 – Multiplexador do bloco FA

O fluxo executado por este multiplexador é mostrado na Figura 66.

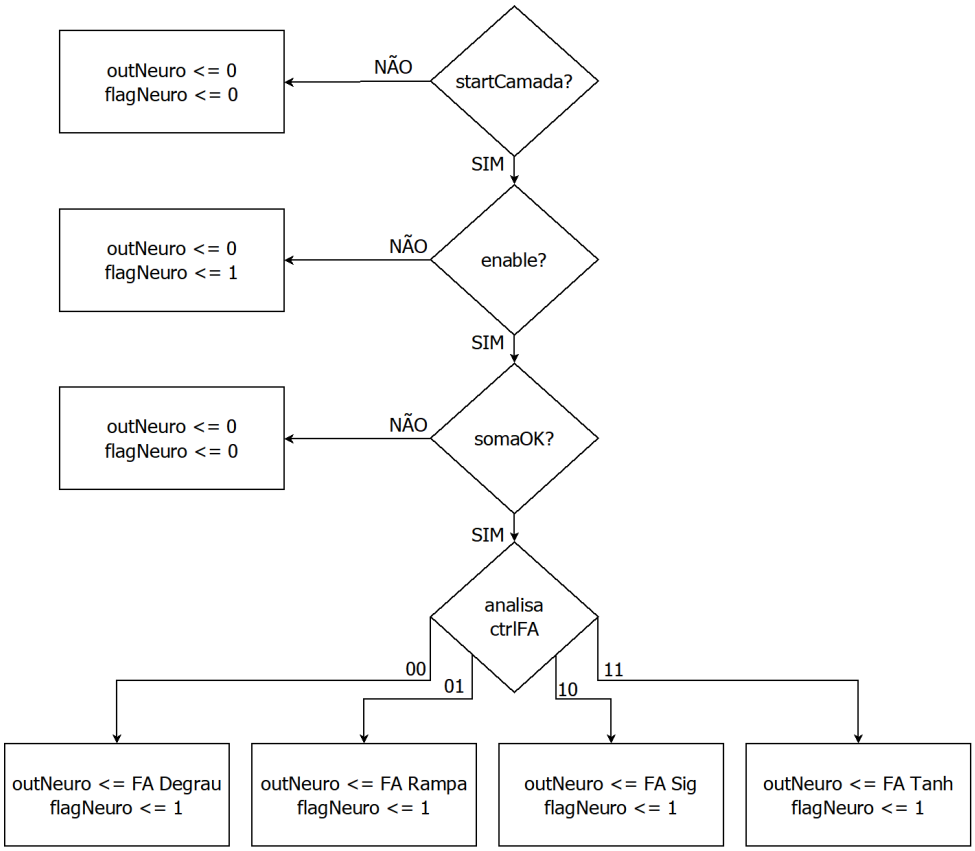

Figura 66 – Fluxo do multiplexador do bloco FA

Primeiramente é verificado se a execução da camada já iniciou, através do sinal *startCamada*: se sim, a execução do multiplexador continua; se não, tanto a saída quanto o *flag* que indica se a saída está pronta são colocados em 0. Em seguida é verificado se o

neurônio está ativo (sinal *enable*): se sim, a execução continua sem interrupções; se não, a saída do neurônio é colocada em 0 e o *flag* é colocado em 1, pois o neurônio não será usado e a camada precisa que todos os flags dos neurônios sejam iguais a 1 para encerrar sua execução.

Uma terceira verificação é feita: se a soma feita pelo MAC está pronta (sinal *somaOK*), o *mux* inicia a escolha de qual das funções de ativação irá para a saída do neurônio, através do sinal *ctrlFA*. Se a soma ainda não está pronta, a saída é colocada em 0.

3.4.2.3 Funcionamento do Neurônio

Uma vez que os funcionamentos dos blocos MAC e FA foram detalhados, pode-se avaliar o funcionamento do neurônio. Uma simulação teste de um neurônio com 2 entradas, sem *bias* e que implementada a função de ativação Tangente Hiperbólica pode ser visto na Figura 67.

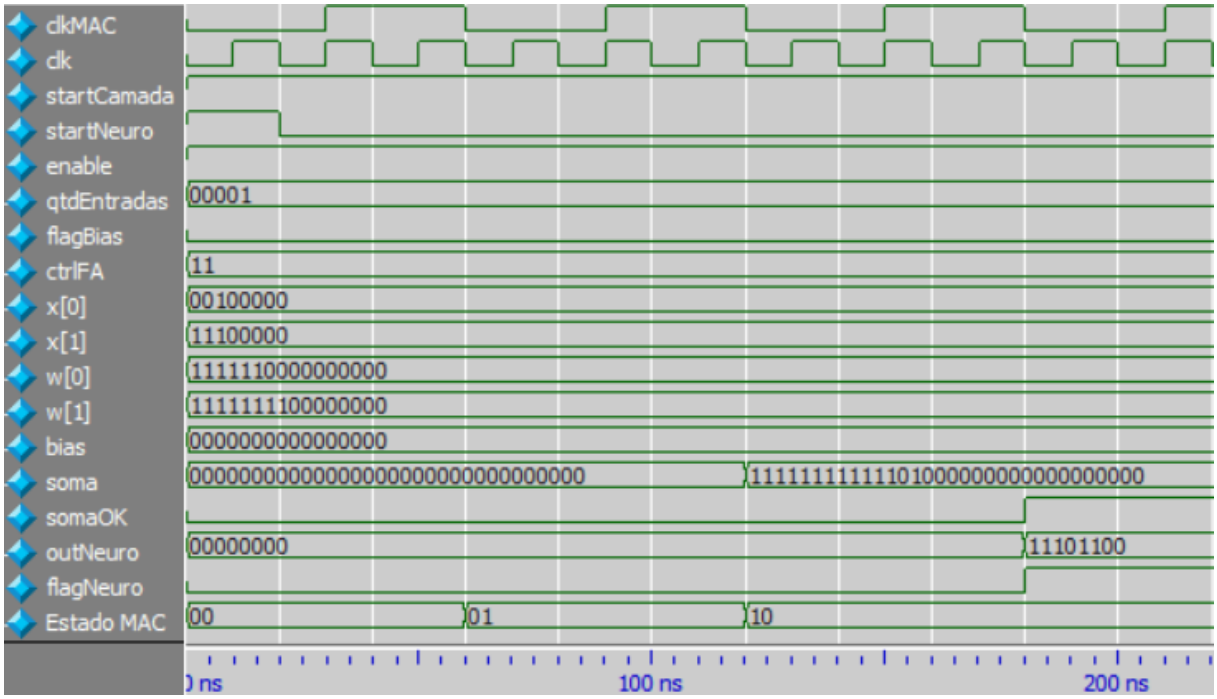

Figura 67 – Carta de Tempo do Neurônio

Neste teste, os valores das entradas foram escolhidos como 1 e -1 e os seus respectivos pesos são -1 e 0,25. Para facilitar a visualização, somente um par destes sinais foram mostrados, mas vale lembrar que o neurônio pode ter até 20 entradas e 20 pesos no total. A soma ponderada esperada é de -0,75 e a saída esperada da Tangente Hiperbólica é -0,6351. O *bias* e o *flagBias* são colocados em 0, já que o neurônio do teste não usa este valor. O sinal de controle *qtdEntradas* tem valor 01, o que indica 2 entradas. O *ctrlFA* é colocado em 11, indicando que o neurônio deve executar a Tangente Hiperbólica. O sinal *enable* é 1, indicando que o neurônio deve ser usado durante a execução da camada. O

sinal *startCamada* está sempre em 1 durante a execução, indicando que a camada que o neurônio se encontra está ativa e operando. O pulso no sinal *startNeuro* indica que a execução do neurônio deve iniciar.

A operação se inicia com o cálculo da soma ponderada pelo MAC. Pode-se observar na Figura 67 que a MdE que controla esse bloco gasta 3 ciclos de *clkMAC* para que a soma esteja disponibilizada. O valor esperado para esta simulação, seguindo a fórmula que determina a quantidade de ciclos de *clock* é $3(2 + arr(2/2)) = 9$, ou seja, o resultado atingido é igual ao esperado. Vale notar que o cálculo da FA é feito de forma simultânea ao funcionamento do MAC, não afetando o tempo de execução do neurônio.

Como esperado o valor da soma é -0,75. Assim que o *flag somaOK* vai para nível lógico 1, o bloco FA calcula a saída do neurônio e na borda de subida do ciclo de *clk* seguinte a saída fica disponível e o *flag* que indica a finalização do neurônio é levado a 1 (*flagNeuro*). O resultado obtido pela simulação foi -0,625, muito próximo do resultado esperado que era -0,6351.

Vale citar que a operação da camada se baseia na simulação paralela dos neurônios. Os *flags* e sinais de controle usados são os mesmos, com a exceção do sinal *enable*, pois há um para cada neurônio. Apesar das entradas serem as mesmas para todos os neurônios da camada, os pesos e *bias* de cada um deles é diferente. A camada é responsável por direcionar os pesos corretamente para cada neurônio.

3.5 Controlador Geral

O Controlador Geral é responsável por garantir que a rede seja executada de forma correta e sincronizada. Ele é formado por uma máquina de estados que manda sinais de controle por toda a rede e assegura a sua execução. Os sinais de controle usados por este bloco durante a execução da rede são detalhados a seguir.

Primeiro são mostrados os sinais que o Controlador Geral recebe do meio externo:

rst: indica que uma nova arquitetura deve ser executada pela RNA;

start: indica que novas entradas para a Rede estão disponíveis. A RNA deve usar a arquitetura atual para resolver essas entradas, sem necessidade de reconfiguração.

Os sinais a seguir são calculados pelo Controlador Geral e mandados para os outros blocos, a fim de garantir a execução:

startCarga: responsável por indicar o início da busca de pesos na Unidade de Memória;

startCamada: indica que a Camada está sendo executada. Garante o início da execução do neurônio;

startNeuro: dá início à execução do neurônio, mais especificamente do MAC;

flagRede: indica que a execução da Rede Neural Artificial foi finalizada e o resultado está disponível.

O Controlador Geral também recebe alguns sinais dos outros blocos e os usa para garantir o fluxo do funcionamento da Rede Neural Artificial. São eles:

instOK: indica que a busca de instruções já foi finalizada e que a RNA pode iniciar a sua execução;

pesosOK: indica que todos os pesos necessários para a execução da camada estão disponíveis;

flagCamada: indica que a execução da camada foi finalizada.

O último grupo de sinais que o Controlador gerencia são os que configuram a nova arquitetura para a Rede Neural Artificial:

ctrlFA: indica qual função de ativação deve ser executada na camada;

numCamada: indica qual camada está sendo executada;

qtdEntradas: revela a quantidade de entradas de uma dada camada;

qtdNeuro: define a quantidade de neurônios de uma camada;

qtdCamadas: indica a quantidade de camadas da RNA;

flagBias: indica se a camada deve ou não considerar o *bias*;

enable 1 a 20: indicam se dado neurônio da camada está sendo usado. Existe um *flag* deste para cada neurônio da camada.

Vale notar que os dados de configuração, com exceção da quantidade de camadas, variam a cada nova camada que inicia sua operação.

O esquema que indica como esses sinais estão conectados é mostrado na Figura 68. Para facilitar a visualização, os sinais de controle que saem do Controlador Geral foram agrupados. Os sinais de entrada da RNA estão destacados com a cor azul e os de saída com a cor vermelha. Os sinais internos estão em preto.

Uma máquina de estados é responsável por mandar os sinais do Controlador aos demais blocos na ordem correta e garantir o funcionamento da Rede. Ela pode ser vista na Figura 69.

Cada estado da MdE tem uma responsabilidade, conforme detalhado a seguir.

rst: Quando o sinal de *reset* é ativado, a topologia da rede será mudada, seguindo as instruções vindas do usuário. A ativação deste sinal garante que as entradas e saídas da camada sejam levadas a zero, também os *flag*s de saída da rede, *startCamada*, *startNeuro*,

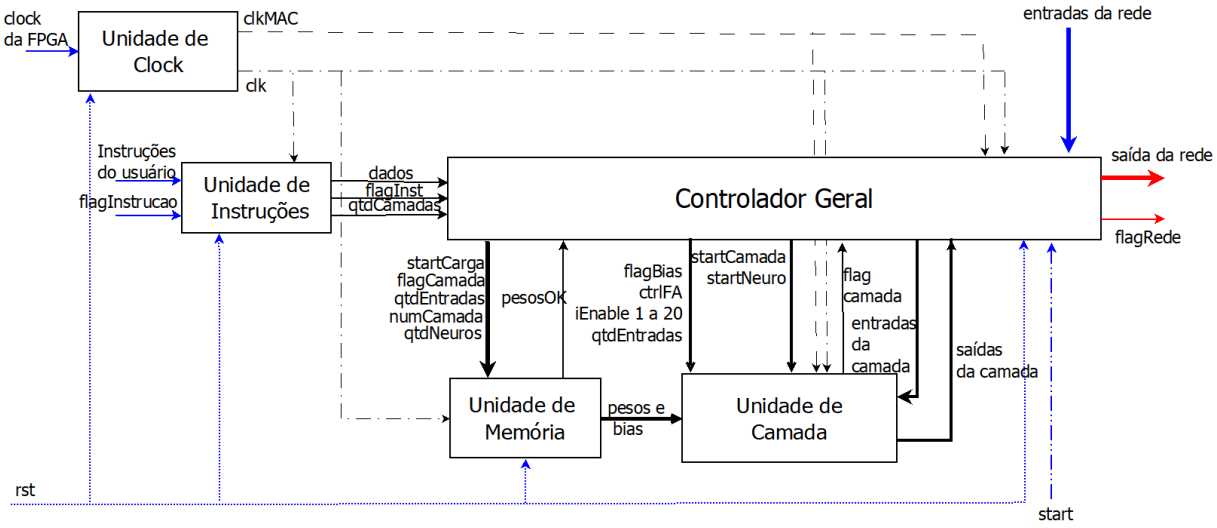

Figura 68 – Arquitetura Geral da RNA com sinais de Controle

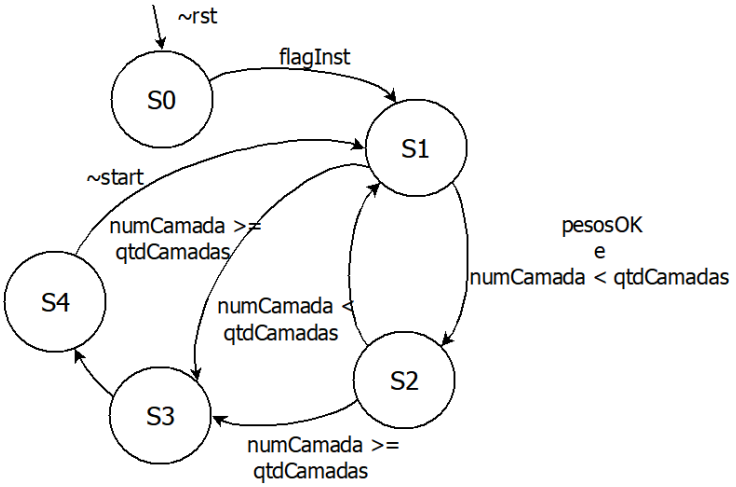

Figura 69 – Mde do Controlador Geral

flagBias, *enable* (de todos os neurônios), *ctrlFA*, *qtdEntradas* e *numCamada*. A busca de instruções também é iniciada por este sinal de *reset*. A MdE é então levada ao estado S0.

S0 - Busca de Instruções: Este estado é responsável pela busca e análise das instruções. Através da análise dos dados vindos das instruções, a RNA é preparada para a busca de pesos e *bias* e execução da camada. O fluxo deste estado é mostrado na Figura 70. A execução deste estado começa somente quando a busca de instruções tiver terminado, ou seja, quando *instOK* estiver em nível alto. Enquanto este sinal não está em 1 o sinal *startCarga* continua em 0. Assim que o sinal se torna 1, o *startCarga* também é levado a 1 e, como a primeira camada da RNA irá iniciar sua execução, o registrador que guarda as entradas da camada (*inCamada*) recebe as entradas da Rede Neural Artificial (*inRede*). Em seguida a MdE é levada ao estado S1. Vale notar que este estado só executa quando uma nova arquitetura for requerida pelo usuário através do sinal *rst*.

S1 – Busca de Pesos: Os dados necessários para a execução da camada são ajus-

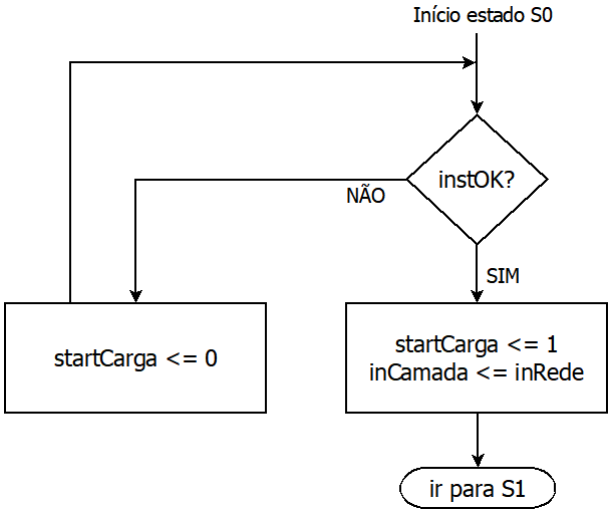

Figura 70 – Fluxo do Estado S0 do Controlador Geral

tados e a busca de pesos é executada. O fluxo executado é mostrado na Figura 71. A primeira ação deste estado é levar o sinal *startCarga* a 0, o que inicia a busca dos pesos. Em seguida, é verificado se todas as camadas da RNA já foram executadas. Se sim, ou seja, se *numCamada* for maior ou igual a *qtdCamadas*, a MdE passa para o estado S3. Caso contrário, as informações necessárias para a execução da camada são retiradas do registrador *dados* vindo da Unidade de Instruções. Os sinais *qtdEntradas*, *flagBias*, *ctrlFA* e *qtdNeuro* são obtidos diretamente de *dados*. Já os sinais *enable* para cada um dos neurônios são calculados a partir de *qtdNeuro*. Após os dados para a execução da camada estarem disponíveis, este estado espera o sinal *pesosOK*, ou seja, espera a busca de pesos e *bias* ser finalizada. Quando a busca é finalizada, os sinais *startNeuro* e *startCamada* que iniciam o funcionamento da camada são colocados em 1 e a MdE vai para o estado S2.

S2 – Execução da Camada: Este estado é responsável por executar a camada da Rede Neural Artificial. Seu fluxo é mostrado na Figura 72. Este estado começa colocando o sinal *startNeuro* como 0, o que inicia a execução dos neurônios da camada. O estado então espera a finalização da camada, indicada pelo *flagCamada*. Assim que este sinal vai a 1, o *startCamada* vai a 0, pois este sinal indica que a camada está em execução. Em seguida, é verificado se a camada que acabou de ser executada era a camada de saída da RNA: se sim, a MdE vai para o estado S3; se não, o sinal *numCamada*, que controla qual camada está sendo executada, é incrementado e o registrador que guarda as entradas da camada a ser executada recebe a saída da camada que acabou de finalizar. Novamente é verificado se a próxima camada a ser executada é a última. Se não for, o sinal que dá início à busca de pesos recebe nível lógico alto e a MdE vai para o estado S1 e a execução deste estado é finalizada.

S3 – Disponibilizar as saídas da RNA: Este estado só é acionado quando todas as camadas da RNA já terminaram sua execução. A saída da camada (*outCamada*) é então

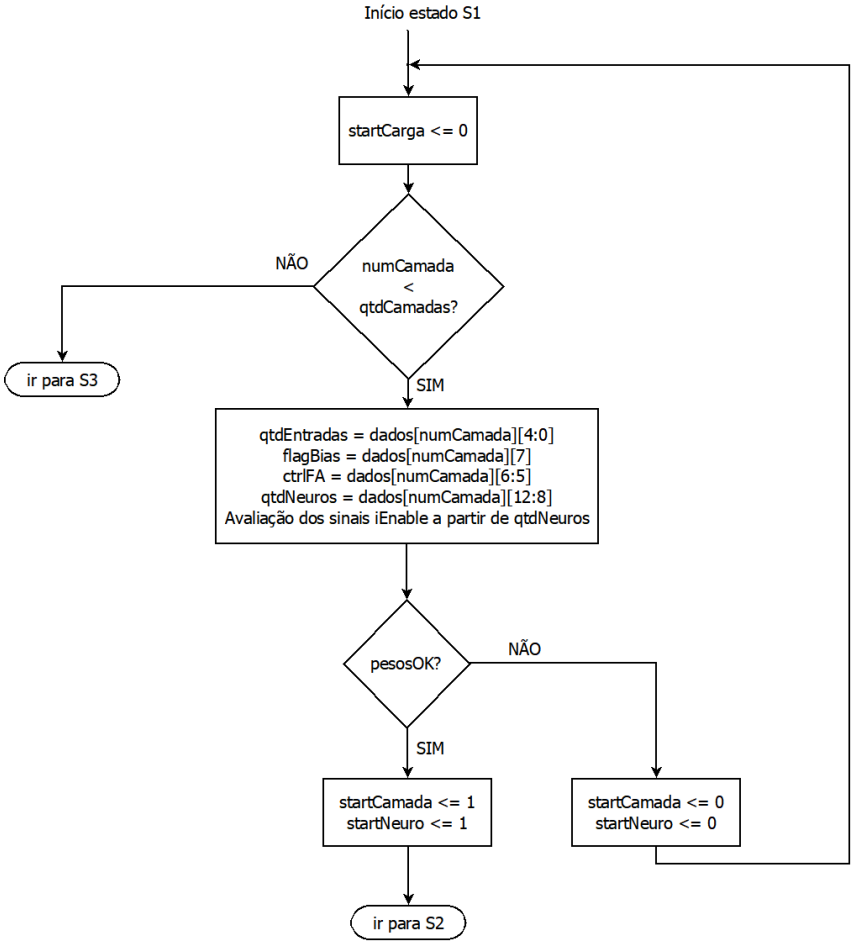

Figura 71 – Fluxo do Estado S1 do Controlador Geral

colocada na saída da Rede Neural Artificial (*outRede*) e essa é disponibilizada para o meio externo. A MdE vai então para o estado S4.

S4 – Finalizar a execução da Rede e aguardar novas entradas: este estado finaliza a execução da RNA e espera novas entradas. Seu fluxo é mostrado na Figura 73. No início do estado, o *flag* que indica que a execução da RNA foi finalizada é colocado em 1, finalizando a execução da Rede Neural Artificial. O estado então aguarda uma borda negativa no sinal de *start*, indicando que novas entradas estão disponíveis para serem calculadas. Assim que este sinal recebe nível 0, os sinais *flagRede* e *numCamada* são zerados e o sinal *startCarga* é colocado em 1. As entradas da camada (*inCamada*) recebem as entradas da RNA (*inRede*) e a MdE passa para o estado S1 para reiniciar sua execução.

Para verificar o fluxo dos sinais de controle da RNA, uma Rede Neural Artificial com 2 camadas e 2 entradas foi simulada. A primeira camada tem 3 neurônios e a segunda tem 1 neurônio. Como o objetivo é mostrar o fluxo de dados, os resultados não serão analisados, somente os sinais de controle. A primeira parte da carta de tempo resultante desta simulação é mostrada na Figura 74. Para facilitar a visualização os sinais de entrada

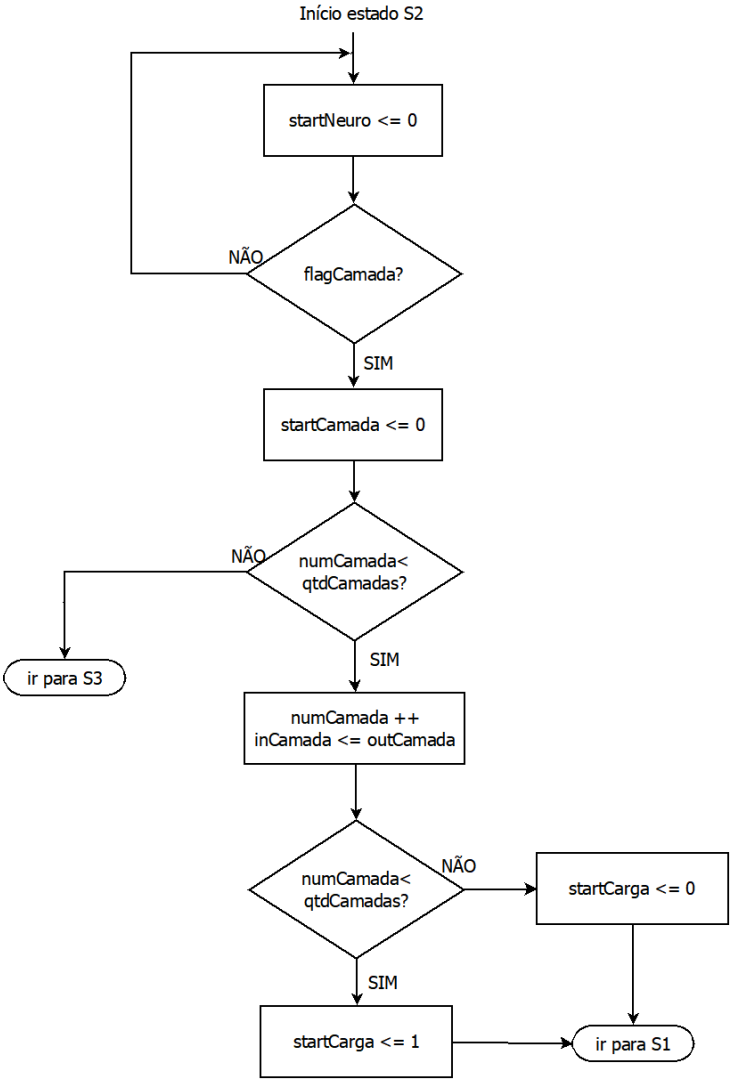

Figura 72 – Fluxo do Estado S2 do Controlador Geral

da RNA não foram mostrados. As marcações em vermelho mostram a passagem dos estados do Controlador Geral. O *clock* de entrada usado tem período de 20 ns, levando a um *clkMAC* de período de 60 ns.

Esta primeira parte da carta de tempo mostra a busca de instruções até a finalização da execução da primeira camada da RNA. Como já dito anteriormente, a etapa de Busca de Instruções tem seu tempo execução afetado pelo tempo que o usuário gasta para inserir todas as execuções. Neste exemplo, são gastos 11 ciclos de *clock* para que todas as instruções necessárias sejam buscadas, analisadas e disponibilizadas para o Controlador Geral. Assim que esta etapa é finalizada, a MdE do Controlador passa para o estado S1. Esta passagem de estado gasta mais um ciclo de *clock*. Na Figura 74 esta passagem está marcada pelo marcador a 220 ns. Vale notar também que, assim que a MdE que controla a busca das instruções passa para o estado S1, o valor da quantidade de camadas é atualizado para o valor binário 10. Este valor indica que a RNA a ser executada tem uma camada de entrada e duas camadas de neurônios.

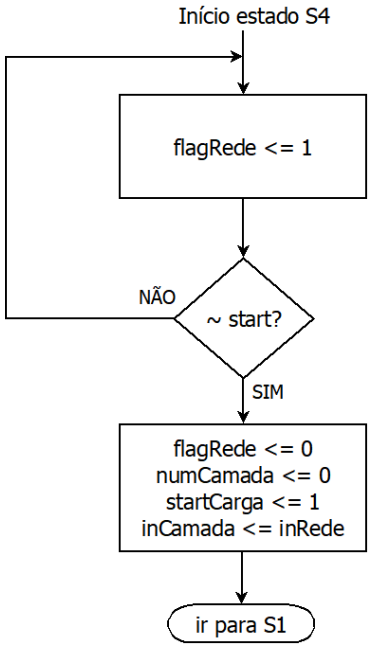

Figura 73 – Fluxo do Estado S4 do Controlador Geral

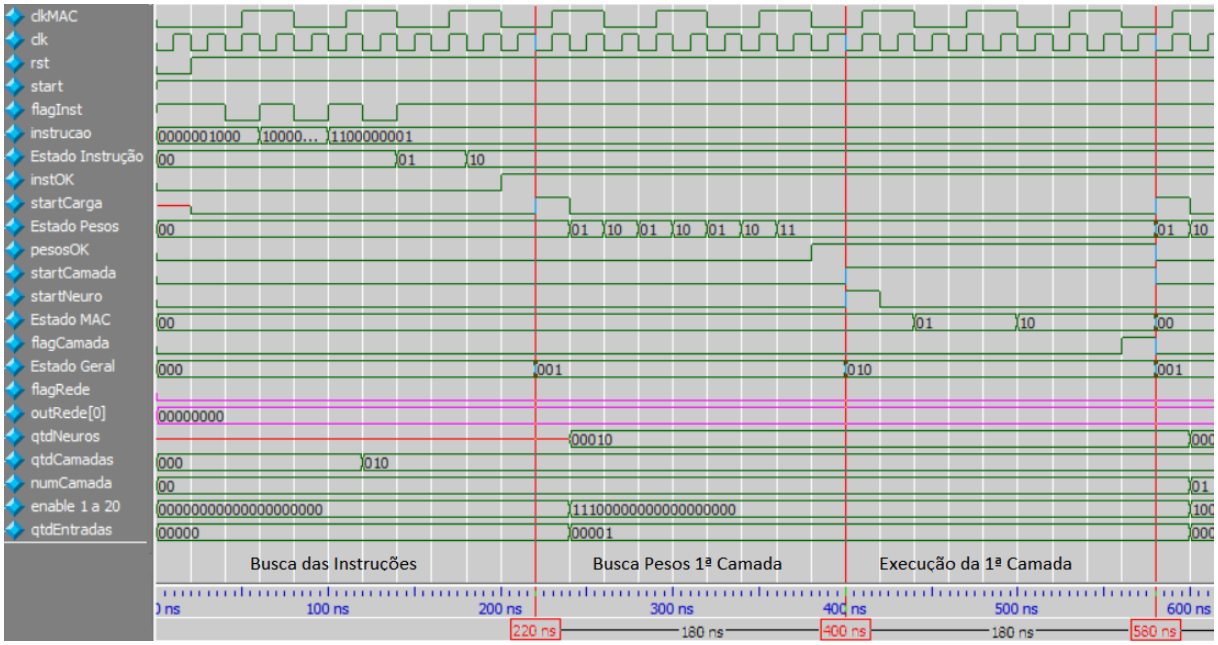

Figura 74 – Carta de Tempo da RNA – Parte 1

A segunda parte da execução da RNA é a Busca de Pesos para a 1ª Camada. No início desta etapa a quantidade de neurônios na camada é atualizada para 3, permitindo o cálculo dos endereços das memórias que devem ser buscados. Esta parte inicia sua execução com o sinal *startCarga* e o estado que controla sua execução oscila entre os estados S1 e S2 até que todos os pesos e *bias* necessários sejam buscados. Tomando a RNA aqui implementada e seguindo a Equação 3.2 tem-se $l_i = l_1, n_1 = 3$ e $e_1 = 2$. O valor esperado para a quantidade de ciclos de *clock* na execução desta busca é $3+2.max\{3,2\} = 9$, que é igual ao observado na simulação (intervalo entre 220 ns e 400 ns na Figura 74).

Assim que a etapa de busca de pesos é finalizada, a execução dos neurônios da camada começa (marcador a 400 ns na Figura 74). Nesta etapa, dois sinais de *clock* são usados, o *clkMAC* para a multiplicação e o *clk* para o restante das operações. Vale notar que ainda no estado S1, quando a RNA está buscando os pesos da camada, os registradores que definem as características da camada a ser executada são atualizados.

A execução da operação dos neurônios ocorre no estado S2 do Controlador Geral. Como para esta camada a quantidade de entradas é igual a 2, são esperados 9 ciclos de *clk* para que toda a soma ponderada seja gerada e a função de ativação da saída dos neurônios calculada. Este valor é observado na simulação. Pela Equação 3.4, o valor esperado era $3(2 + arr(2/2)) = 9$, ou seja, o resultado esperado foi igual ao observado na simulação (intervalo entre 400 ns e 580 ns na Figura 74).

A segunda parte da simulação é vista na Figura 75.

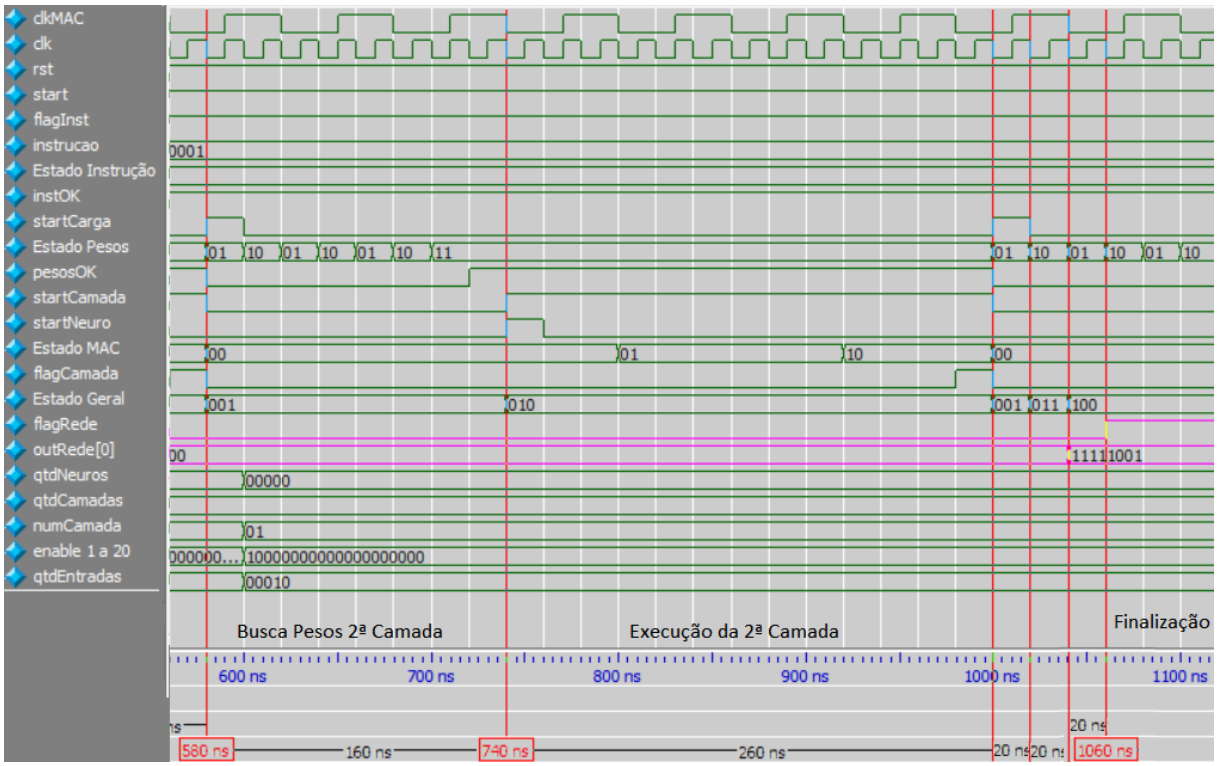

Figura 75 – Carta de Tempo da RNA – Parte 2

Esta segunda parte mostra a execução da camada de saída e a finalização da RNA. Assim que a operação da primeira camada termina, a MdE do Controlador Geral volta ao estado S1 e o sinal de controle *numCamada* é incrementado. Como este sinal ainda é menor que *qtdCamadas*, a Rede Neural Artificial inicia a busca de pesos da camada seguinte. Vale notar que os dados de configuração da RNA são atualizados para valores que correspondem à camada de saída. Tem-se então, $l_i = l_2, n_2 = 1$ e $e_2 = 3$.

A execução da busca de pesos é semelhante ao que foi explicado anteriormente. Para esta busca o valor esperado de quantidade de ciclos de *clock* usados, segundo a

Equação 3.2 é $2 + 2.max\{1, 3\} = 8$, valor observado na simulação (intervalo entre 580 ns e 740 ns na Figura 75).

Novamente, com o fim da busca de pesos a execução dos neurônios na camada é iniciada. Nota-se que para esta segunda camada a quantidade de entradas aumentou para 3. A quantidade esperada de ciclos gastos é $1 + 3(2 + arr(3/2)) = 13$, valor observado na simulação (intervalo 740 ns a 1000 ns na Figura 75).

Com o fim da execução da segunda camada a MdE do Controlador Geral volta ao estado S1 e verifica através de *numCamada* que a camada que acabou de ser executada foi a última da RNA. A MdE vai então ao estado S3, onde a saída da Rede Neural Artificial é atualizada e em seguida vai para o estado S4, onde o *flagRede* é levado a nível lógico 1, indicando que a execução da RNA terminou. Este processo de finalização gasta 3 ciclos de *clk*.

A quantidade mínima de ciclos de *clock* usada na execução da RNA é calculada como:

$$\text{ciclos} = \{\text{qtd ciclos instruções}\} + \left\{3 + 2.max\left\{n_1, e_1\right\} + 3\left(2 + arr\left(\frac{e_1}{2}\right)\right)\right\} + \sum_{i=2}^{numCamadas}\left\{2 + 2max\left\{n_i, e_i\right\} + 1 + 3\left(2 + arr\left(\frac{e_1}{2}\right)\right)\right\} + 3$$

$$\text{ciclos} = \{\text{qtd ciclos instruções}\} + \left\{9 + 2.max\left\{n_1, e_1\right\} + 3arr\left(\frac{e_1}{2}\right)\right\} + \sum_{i=2}^{numCamadas}\left\{9 + 2max\left\{n_i, e_i\right\} + 3arr\left(\frac{e_1}{2}\right)\right\} + 3$$

$$\text{ciclos} = \{\text{qtd ciclos instruções}\} + \sum_{i=1}^{numCamadas}\left\{9 + 2max\left\{n_i, e_i\right\} + 3arr\left(\frac{e_1}{2}\right)\right\} + 3 \quad (3.9)$$

Vale lembrar que esta quantidade é dita mínima pois o valor usado pelo MAC pode diferir em até 2 ciclos de *clock* por camada do valor calculado.

Para esta simulação, este valor é igual a 53 ciclos de *clock*, valor igual ao observado na simulação (Marcador no instante 1060 ns na Figura 75).

Algumas simulações de testes reais foram feitas e suas análises e resultados são mostrados no Capítulo 4.

4 Simulações e Resultados

Alguns testes foram feitos a fim de comprovar e avaliar o funcionamento geral e a flexibilidade da Rede Neural Artificial implementada. Três problemas com arquiteturas diferentes foram simulados para mostrar a possibilidade de múltiplas arquiteturas no mesmo hardware.

O primeiro problema é do tipo aproximação de função. Ele foi testado através de *testbench* e em seguida foi implementado na FPGA a fim de verificar o real funcionamento da RNA. Os erros entre o resultado esperado e o obtido são analisados.

O segundo problema testado foi proposto em (95). Este é um clássico problema de classificação de padrões. A arquitetura da RNA testada foi a mesma do artigo que a propôs. O teste foi simulado através de *testbench* e os resultados foram analisados. Este teste não pôde ser implementado em FPGA pois a quantidade de pinos disponíveis para externalização na placa é menor que a quantidade necessária para este problema.

O terceiro problema testado é do tipo previsão de série temporal. Assim como o segundo problema, a arquitetura da RNA simulada é igual à proposta pelo artigo (12). Os resultados foram obtidos através de *testbench* e analisados. Assim como o segundo problema proposto, a quantidade de pinos disponível na FPGA impossibilitou o teste em placa.

Como a RNA proposta não possui módulo de treinamento, foi necessário para estes testes que um treinamento fosse feito de modo *off-line* a fim de obter os valores de pesos e *bias* necessários para a simulação. Este treinamento foi feito no MATLAB através da *Neural Network Toolbox* que permite a criação e treinamento de diferentes tipos de Redes Neurais Artificiais. Os valores calculados pelo MATLAB são então inseridos nas memórias de pesos e *bias* para que a RNA possa então ser testada. Os códigos em MATLAB usados para o treinamento das RNAs aqui testadas estão no CD anexo.

Todos os três problemas foram simulados através do software ModelSim da Altera.

4.1 Aproximação de Função

Um dos problemas mais clássicos resolvidos por RNAs é a aproximação de funções não lineares. Para testar este tipo de problema, a função Sinc foi simulada.

$$y(x) = \frac{sen(\pi x)}{\pi x} \tag{4.1}$$

Para a aproximação neste trabalho foi usada uma MLP com uma entrada, 13

neurônios na camada oculta, com função de ativação Tangente Hiperbólica, e 1 neurônio com função de ativação Rampa Limitada na saída, como visto na Figura 76.

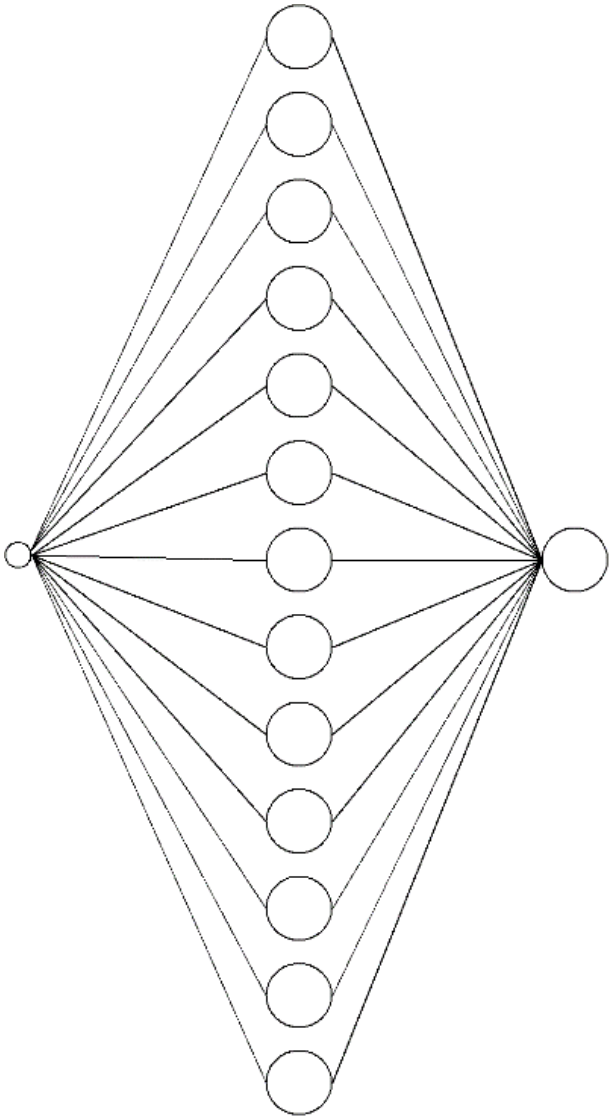

Figura 76 – Arquitetura para aproximação de função

A arquitetura foi treinada no MATLAB usando a *Neural Network Toolbox* e os pesos e *bias* obtidos foram convertidos para a notação usada neste trabalho e inseridos nas memórias dos neurônios.

As instruções necessárias para a simulação foram então construídas a partir das características da arquitetura alvo. A Tabela 16 mostra as instruções e a maneira que elas foram construídas.

Com as instruções prontas, o *testbench* para esta simulação foi criado e a RNA foi então simulada.

O resumo do tempo e área gastos na simulação pode ser visto na Tabela 17.

Como a RNA é reconfigurável, a área para todos os testes é a mesma, indepen-

Tabela 16 – Instruções geradas para a Aproximação de Função

Número da Instrução	Tipo de Camada	Quantidade de Neurônios na Camada	Presença de *Bias*	Função de Ativação	Instrução Gerada
1	Entrada	1	–	–	0000000000
2	Oculta	13	Sim	Tangente Hiperbólica	1001100111
3	Saída	1	Sim	Rampa	1100000101

Tabela 17 – Resumo do resultado da simulação do problema de Aproximação de Função

FPGA usada	EP4CE115F29C7
Número de neurônios implementados	20
Arquitetura da RNA	1 - 13 - 1 (14 neurônios usados)
Total de elementos lógicos	39.344 / 114.480 (34%)
Total de funções combinacionais	38.840 / 114.480 (34%)
Total de registradores dedicados	8.527 / 114.480 (7%)
Total de bits de memória	43.008 / 3.981.312 (1%)
Multiplicadores dedicados de 9 bits	0 / 532 (0%)
Pinos	335 / 529 (63%)
Número de ciclos de *clock*	110
Frequência máxima de operação	77.59 MHz

dente da arquitetura da Rede Neural Artificial, como será visto nas próximas análises. A frequência máxima de operação obtida através da ferramenta *TimeQuest Timing Analyzer*, do Quartus, foi de 77,59 MHz.

Para a verificação do funcionamento, a simulação de uma entrada é mostrada nas Figuras 77 e 78. Nestas figuras, cada estado do Controlador Geral está separado por um marcador em vermelho. Foi usado um *clock* com período de 20 ns, porém a análise do tempo de execução será independente deste valor, pois será usada a quantidade de ciclos de *clock* gastos na simulação completa.

Esta primeira parte da carta de tempo mostra a execução da busca de instruções RNA e da execução da primeira camada desta Rede Neural Artificial. Para a busca e análise das instruções foram necessários 17 ciclos de *clock*, valor dependente da velocidade com que as instruções são colocadas na RNA.

Para a primeira camada da RNA tem-se $l_i = l_1, n_1 = 13$ e $e_1 = 1$. Assim, a quantidade de ciclos de *clock* esperada para a busca de pesos segundo a Equação 3.2 é $3 + 2.max\{13, 1\} = 29$, que é igual ao valor observado na simulação. Para a execução da camada são esperados $3(2 + arr(1/2)) = 9$ ciclos de *clk*, segundo a Equação 3.4.

A segunda parte da carta de tempo (Figura 78) mostra a execução da segunda camada da RNA e a finalização da operação. Nesta segunda camada da Rede Neural Artificial tem-se $l_i = l_2, n_2 = 1$ e $e_i = 13$. Assim, a quantidade de ciclos de *clock* esperada

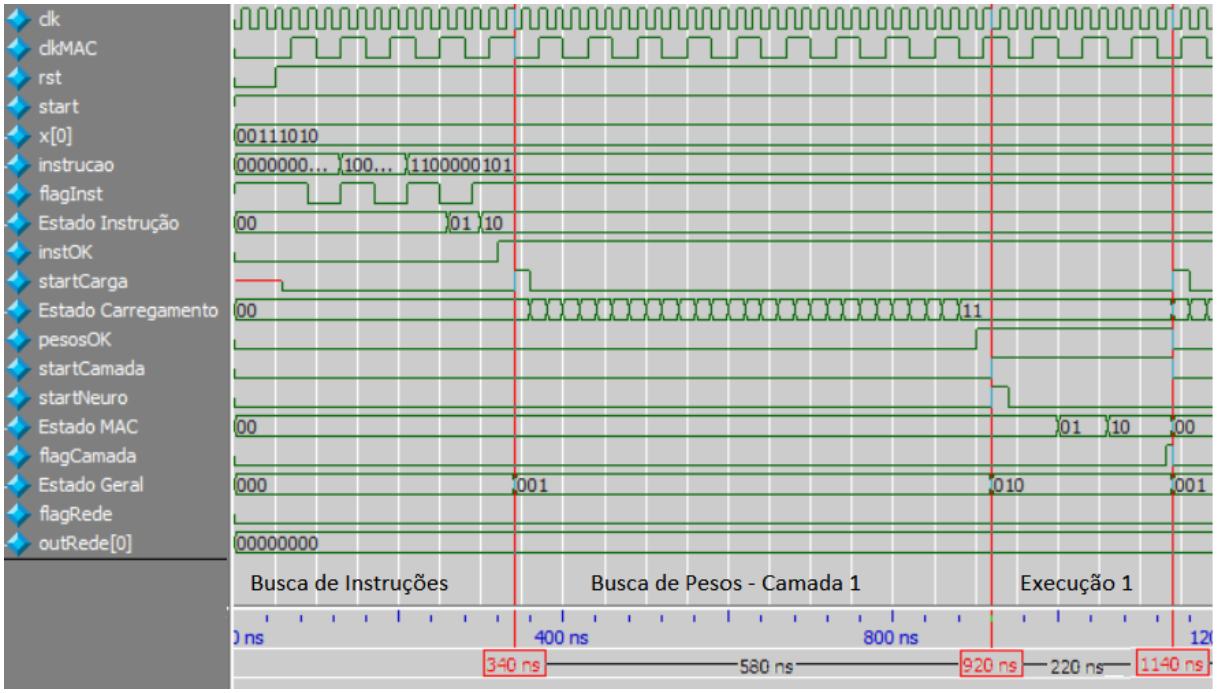

Figura 77 – Carta de Tempo para aproximação de função - Parte 1

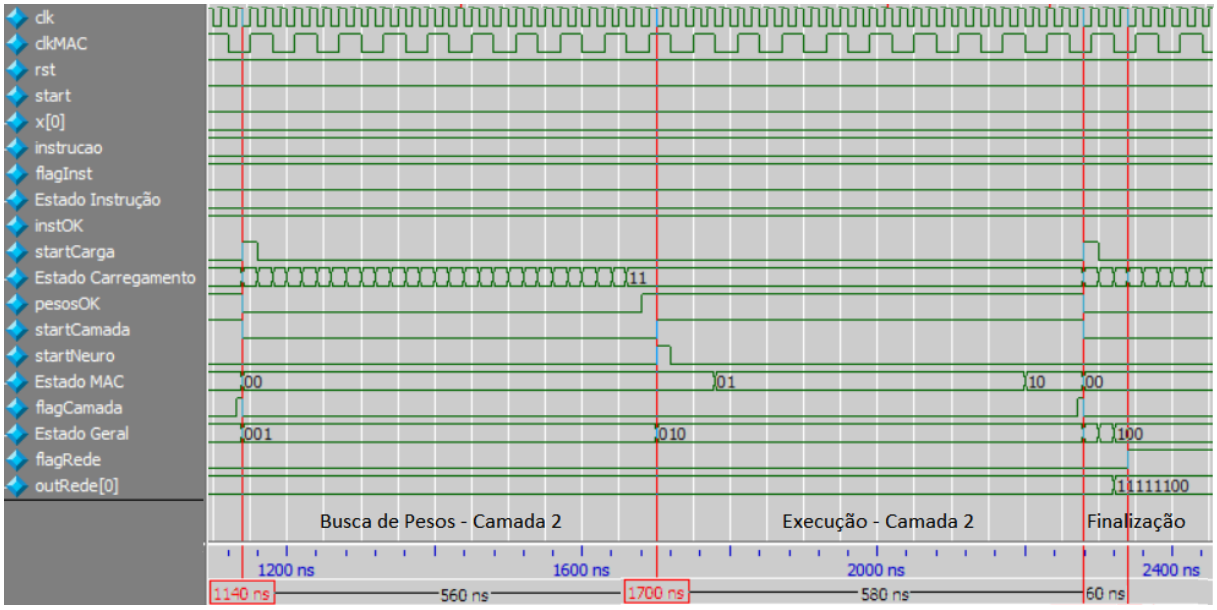

Figura 78 – Carta de Tempo para aproximação de função - Parte 2

para a busca de pesos é $2 + 2.max\{1, 13\} = 28$, segundo a Equação 3.2. Na execução dos neurônios são esperados $1 + 3(2 + arr(13/2)) = 28$ ciclos de *clock*, segundo a Equação 3.4.

A quantidade mínima de ciclos de *clock* esperados é igual a 114 ciclos de *clock*, valor próximo ao atingido pela simulação, que foi de 117 ciclos (instante 4400 ns da 78). Assim como visto anteriormente, esta diferença se dá devido ao uso de um *clock* diferente na multiplicação.

A fim de analisar a exatidão dos resultados da RNA, a aproximação da função

Sinc foi testada para valores entre [-4,4] e os resultados obtidos foram enviados para o MATLAB a fim de se obter o gráfico da Figura 79.

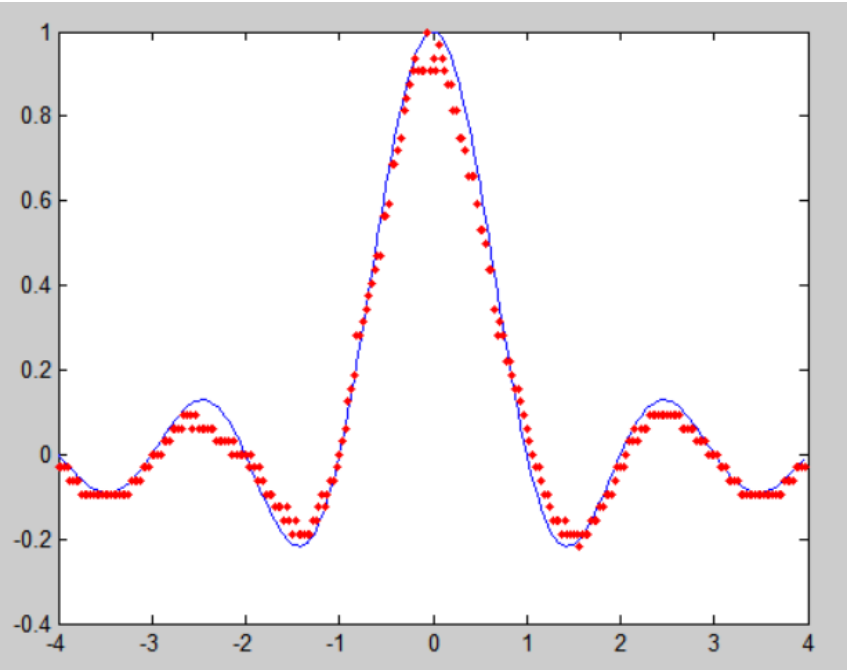

Figura 79 – Resultado da aproximação da função Sinc

Na Figura 79 observa-se que os resultados da simulação obtidos pela RNA (pontos em vermelho) se ajustam bem à curva esperada (em azul). Maiores erros foram encontrados na parte negativa da curva, com entradas em torno de -2,5. Este pior resultado pode ser justificado tanto pela quantidade de bits usada pela RNA quanto pela aproximação da função Tangente Hiperbólica usada. Em geral, não eram esperados resultados exatos, já que o uso de palavras de somente 8 bits traz erros de representação.

O coeficiente de determinação desta aproximação, que mede o quanto o resultado está próximo da curva esperada, foi calculado através do MATLAB e seu valor foi de 0,9908, o que mostra que a curva obtida por esta Rede Neural é bem próxima à esperada.

O erro médio, máximo e a soma dos erros quadrados obtidos pela simulação no hardware e o esperado para esta mesma rede em software são mostrados na tabela a seguir:

Tabela 18 – Resumo do resultado da simulação do problema de Aproximação de Função

	Erro médio	Erro máximo	Soma dos erros quadrados
RNA em Hardware	0,032482639	0,1279633	0,450873315
RNA em Software	0,000287699	0,00241683	0,0000441629

Os maiores valores encontrados na simulação no hardware são devido à notação numérica usada, já que esta Rede Neural Artificial foi projetada para trabalhar com palavras em ponto fixo de 8 bits e sistemas em software têm maiores precisões. Mesmo

assim, os resultados em hardware dão uma boa aproximação para a Função Sinc. Os resultados podem ser melhorados com a modificação da arquitetura da RNA ou com o aumento do número de bits da implementação, porém, esta última traz um significante aumento na área do circuito.

Este problema foi então implementado na FPGA de modelo EP4CE115F29C7 para uma análise dos resultados. Alguns sinais de controle e o estado do controlador geral foram exibidos, a fim de facilitar a análise. Como somente uma entrada é necessária para este problema, as outras 19 entradas possíveis foram selecionadas como pinos do tipo *"input tri-state"*, fazendo com que elas estejam em alta impedância e não influenciem na operação da RNA. Os resultados obtidos pela FPGA foram enviados a um analisador lógico e as formas de onda encontradas podem ser vistas na Figura 80.

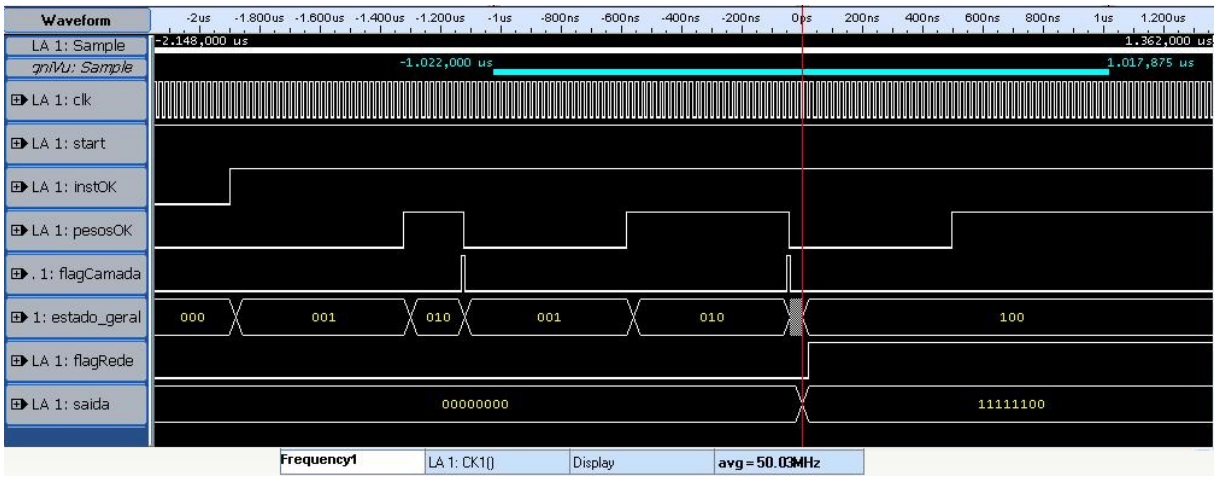

Figura 80 – Resultados do analisador lógico para a aproximação de função

A fim de facilitar a visualização dos resultados, é mostrado na Figura 81 um *zoom* na região em que o *flagRede* vai para 1.

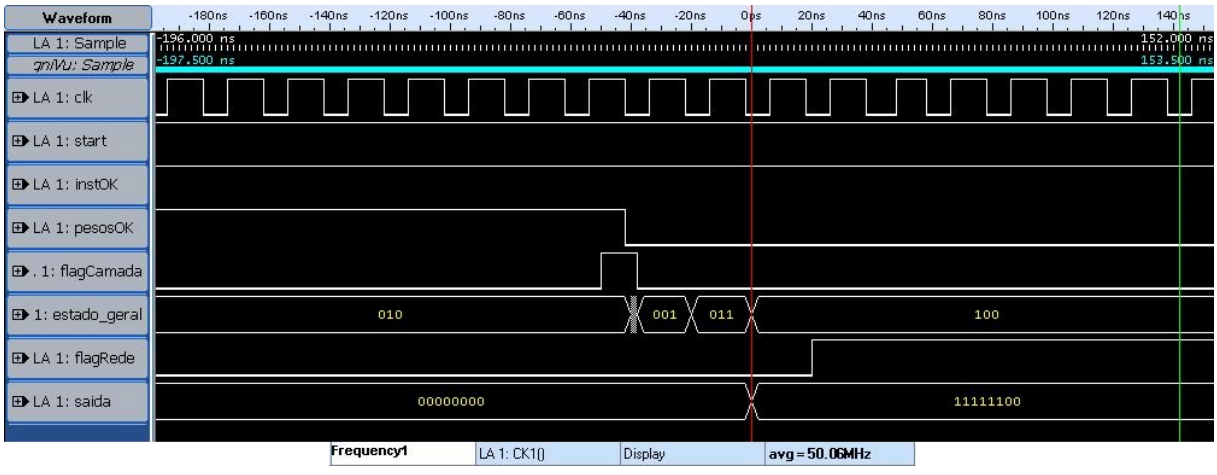

Figura 81 – *Zoom* dos resultados do analisador lógico para a aproximação de função

Para esta aproximação de função, o *clock* usado foi de 50 MHz disponível na FPGA.

A entrada tem valor igual a 1,8125, a mesma usada para a avaliação da carta de tempo da Figura 78. O valor esperado para a saída é de -0,0976. O resultado obtido através da FPGA foi de -0,125, sendo assim, o erro absoluto entre a saída esperada e a obtida é de 0,0274. Uma maneira de melhorar os resultados é através da mudança da arquitetura da RNA, como dito anteriormente.

4.2 Classificação de Padrões

O teste de classificação de padrões escolhido foi o problema Íris (95, 96). Este problema se propõe a classificar flores do tipo íris em 3 classes, a partir de medidas de largura e comprimento da sépala e da pétala. As classes possíveis são Íris Setosa, Íris Versicolour e Íris Virginica. Os dados usados para a classificação foram obtidos do repositório UCI (96) e consistem em 150 amostras, 50 para cada classe.

A arquitetura usada é a mesma da (95), para que os resultados possam ser comparados. Ela é composta de 4 entradas, uma para cada característica a ser analisada, 2 camadas ocultas com 8 e 3 neurônios respectivamente, ambas implementando a função Tangente Hiperbólica, e uma camada de saída com 3 neurônios, com função de ativação Rampa Limitada, cada um deles representando uma classe. Assim, se as características de entrada correspondem a uma Íris Setosa, a saída do neurônio 1 será 1 e a saída dos demais neurônios será -1; para a Íris Versicolour as saídas serão -1, 1 e -1 para os neurônios 1, 2 e 3 respectivamente; e para a Iris Virginica as saídas serão -1, -1 e 1. A arquitetura da Rede Neural Artificial usada para a classificação pode ser vista na Figura 82.

Esta arquitetura foi incialmente treinada no MATLAB e os valores para os pesos e *bias* foram convertidos para a representação aqui usada e em seguida colocados nas memórias da RNA.

Com os pesos prontos e a arquitetura escolhida, basta inserir as instruções na RNA e simular seu funcionamento no *ModelSim*. O conjunto de instruções usado é visto na Tabela 19.

Tabela 19 – Instruções geradas para a Classificação de Padrões

Número da Instrução	Tipo de Camada	Quantidade de Neurônios na Camada	Presença de *Bias*	Função de Ativação	Instrução Gerada
1	Entrada	4	–	–	0000011000
2	Oculta	8	Sim	Tangente Hiperbólica	1000111111
3	Oculta	3	Sim	Tangente Hiperbólica	1000010111
4	Saída	3	Sim	Rampa	1100010101

O resumo do tempo e área gastos na simulação pode ser visto na Tabela 20 a seguir.

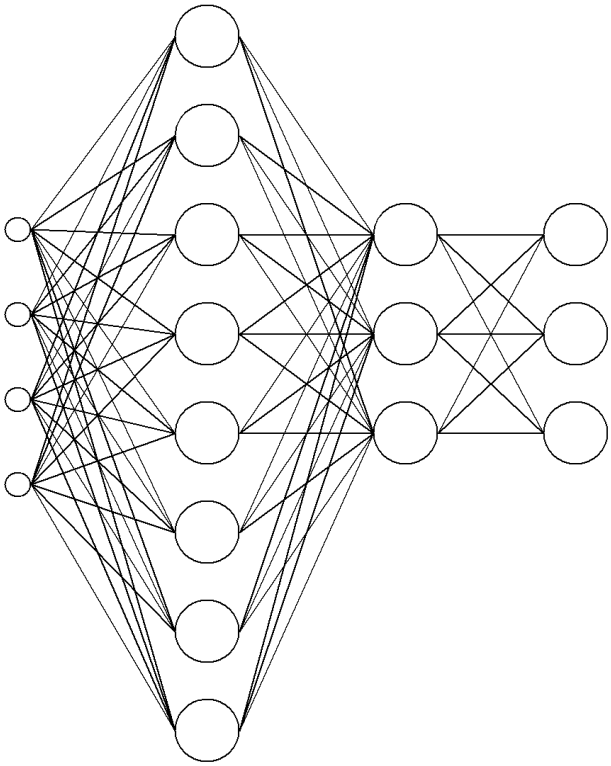

Figura 82 – Arquitetura para o problema Íris

Tabela 20 – Resumo do resultado da simulação do problema de Classificação de Padrões

FPGA usada	EP4CE115F29C7	EP3SL50F484C2
Número de neurônios implementados	20	14
Arquitetura da RNA	4 - 8 - 3 - 3 (14 neurônios usados)	4 - 8 - 3 - 3
Total de elementos lógicos	39.344 / 114.480 (34%)	–
Total de funções combinacionais	38.840 / 114.480 (34%)	–
Total de registradores dedicados	8.527 / 114.480 (7%)	9.791 (26%)
Total de bits de memória	43.008 / 3.981.312 (1%)	181.901 (3%)
Multiplicadores dedicados de 9 bits	0 / 532 (0%)	–
DSPs	–	12 (3%)
LUTs	–	8.782 (23%)
Pinos	335 / 529 (63%)	–
Número de ciclos de *clock*	110	37
Frequência máxima de operação	77.59 MHz	300 MHz

Como esperado, a área usada na arquitetura proposta é a mesma da primeira simulação. A diferença vem da quantidade de ciclos de *clock* necessários para a simulação, que para este caso é 110.

Ao se comparar a área da implementação proposta com a do artigo (95), nota-se que, embora a quantidade de neurônios aqui implementada seja maior (20 neurônios contra 14 neurônios), a quantidade de registradores usados é menor (8.526 contra 9.791). Também, a quantidade de bits de memória usada aqui é menor (43.008 contra 181.901).

Isto mostra a economia em área ao se usar uma topologia reconfigurável. Além disso, esta implementação não usa multiplicadores ou blocos DSPs para a sua operação, o que também é uma vantagem, já que a estrutura fica independente do modelo da FPGA usada e pode, posteriormente, ser usada para criar um sistema em CI.

Por outro lado, como os multiplicadores usados são independentes da FPGA, era de se esperar que a frequência máxima de operação atingida fosse menor que outras implementações, fato observado (Tabela 20). Também, como foi usada a estratégia de criar somente dois multiplicadores que seriam reutilizados até que a soma ponderada de todas as entradas fosse atingida, observa-se uma maior quantidade de ciclos de *clock* para que o resultado final da RNA esteja disponível.

Para melhor visualizar a operação da RNA, é mostrada uma carta de tempo para uma simulação (Figura 83).

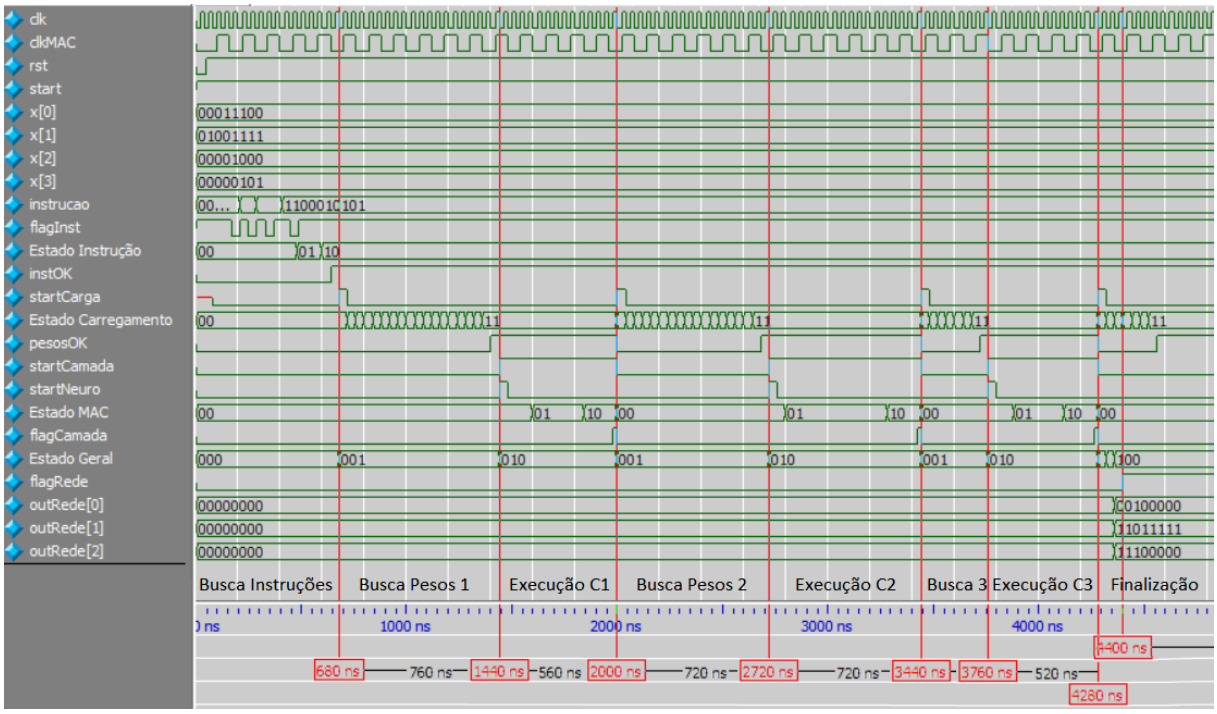

Figura 83 – Carta de Tempo para o problema Íris

A simulação foi feita com *clock* de entrada de 40 ns, porém este valor pode mudar e não é limitante na simulação. Para que a análise seja independente do valor do *clock* de entrada, serão analisadas as quantidades de ciclos de *clock* usadas. As linhas vermelhas demarcam a execução de cada estado, incluindo busca de pesos e operação dos neurônios.

A primeira parte da simulação aborda a reconfiguração da RNA e só é executada uma vez. Esta etapa leva 17 ciclos de *clock* para ser finalizada. Nota-se que a segunda camada é a que leva mais ciclos de *clock* para ser executada, pois esta tem 8 entradas, vindo da camada anterior, fazendo necessários mais ciclos para a busca de pesos e para o cálculo da soma ponderada das entradas.

Ao se comparar a quantidade de ciclos de *clock* usados no primeiro teste e neste segundo teste, fica claro a dependência do valor da quantidade de ciclos usados com a arquitetura da RNA: quanto mais neurônios, entradas e camadas a Rede Neural Artificial tem, maior o tempo gasto na sua execução. Para a primeira camada da RNA tem-se $l_i = l_1, n_1 = 8$ e $e_1 = 4$. Assim, a quantidade de ciclos de *clock* esperada para a busca de pesos é $3 + 2.max\{8, 14\} = 19$ (Equação 3.2). Para a execução da camada são esperados $3(2 + arr(4/2)) = 12$ ciclos de *clk* (Equação 3.4).

Na segunda camada desta Rede Neural Artificial tem-se $l_i = l_2, n_2 = 3$ e $e_i = 8$. Assim, a quantidade de ciclos de *clock* esperada para a busca de pesos é $2+2.max\{3, 8\} = 18$ (Equação 3.2) e na execução dos neurônios são esperados $1 + 3(2 + arr(8/2)) = 19$ ciclos de *clock* (Equação 3.4).

Finalmente na última camada tem-se $l_i = l_3, n_3 = 3$ e $e_3 = 3$. Assim, a quantidade de ciclos de *clock* esperada para a busca de pesos é $2 + 2.max\{3, 3\} = 8$ (Equação 3.2) e na execução dos neurônios são esperados $1+3(2+arr(3/2)) = 13$ ciclos de *clock* (Equação 3.4).

A quantidade total de ciclos de *clock* mínima esperada é de 109, porém, a simulação gastou 110 ciclos para finalizar a execução. Esta diferença se deve ao fato do multiplicador usar um *clock* diferente, o que pode trazer uma certa diferença. Mesmo assim, a quantidade de ciclos calculada e observada são próximas.

Para avaliar a classificação dos padrões, algumas medidas são usadas na literatura como sensibilidade, especificidade e precisão. Para calcular essas medidas, é necessário saber a taxa de verdadeiro positivo (TP), verdadeiro negativo (TN), falso positivo (FP) e falso negativo (FN). Essas medidas são definidas a seguir:

Verdadeiro Positivo (TP): quantidade de classes que deveriam ser positivas e foram classificadas como positivas.

Verdadeiro Negativo (TN): quantidade de classes que deveriam ser negativas e foram classificadas como negativas.

Falso Positivo (FP): quantidade de classes que deveriam ser classificadas como negativas, mas foram classificadas erroneamente como positivas.

Falso Negativo (FN): quantidade de classes que deveriam ser classificadas como positivas, mas foram classificadas erroneamente como negativas.

Sensibilidade: a porcentagem de classificações positivas corretas. Definida como:

$$s = \frac{TP}{TP + FN} \tag{4.2}$$

Especificidade: porcentagem de classificações negativas corretas. Definida como:

$$e = \frac{TN}{TN + FP} \tag{4.3}$$

Precisão: porcentagem de classificações corretas. Definida como:

$$p = \frac{TN + TP}{TN + TP + FN + FP} \tag{4.4}$$

Para cada uma das saídas possíveis deve ser feita a análise das classificações. Os resultados podem ser vistos na tabela a seguir:

Tabela 21 – Análise do resultado da simulação do problema de Classificação de Padrões

Medida	Classe 1	Classe 2	Classe 3	Geral
TP	50	49	50	149
TN	100	100	99	299
FP	0	0	1	1
FN	0	1	1	1
Sensibilidade	100%	98%	100%	99,3%
Especificidade	100%	100%	99%	99,6%
Precisão	100%	99,3%	99,3%	99,5%

Para o caso geral, a precisão da solução desta RNA foi de 99,5%, o que é uma alta taxa para RNAs em hardware. Os resultados para estas medidas para o artigo usado como base não foram mostrados e por isso não foram comparados.

O artigo base indica um valor de erro na saída da RNA. Apesar deste valor não ser significativo na classificação de padrões, ele é mostrado a seguir:

Tabela 22 – Comparação entre os resultados do problema Íris

	Esta abordagem	Abordagem (95)
Erro máximo	0,0766	0,0021
Soma dos quadrados dos erros	0,2309	0,000024

A RNA do artigo base usa sinais em ponto flutuante, sendo assim, já era esperado que o erro apresentado fosse maior do que neste trabalho, pois esta abordagem usa ponto fixo. Porém, este desvio nos resultados esperados não afetou de forma significativa o poder de classificação da Rede Neural Artificial.

4.3 Previsão de Série Temporal

Este último problema analisado corresponde à previsão de uma série temporal, conhecida como Mackey-Glass (12). A previsão de séries temporais também é um problema

importante na RNAs, que consiste em prever novos valores de uma série a partir de valores passados.

A série de Mackey-Glass é definida como (97):

$$\frac{dx}{dt} = \frac{a_m x(t-T)}{b_m + x_m^h(t-T)} - c_m x(t) \tag{4.5}$$

Onde $a_m = 0,2; b_m = 1; c_m = 0,1$ e $d_m = 17$.

A saída esperada para esta série é mostrada na Figura 84.

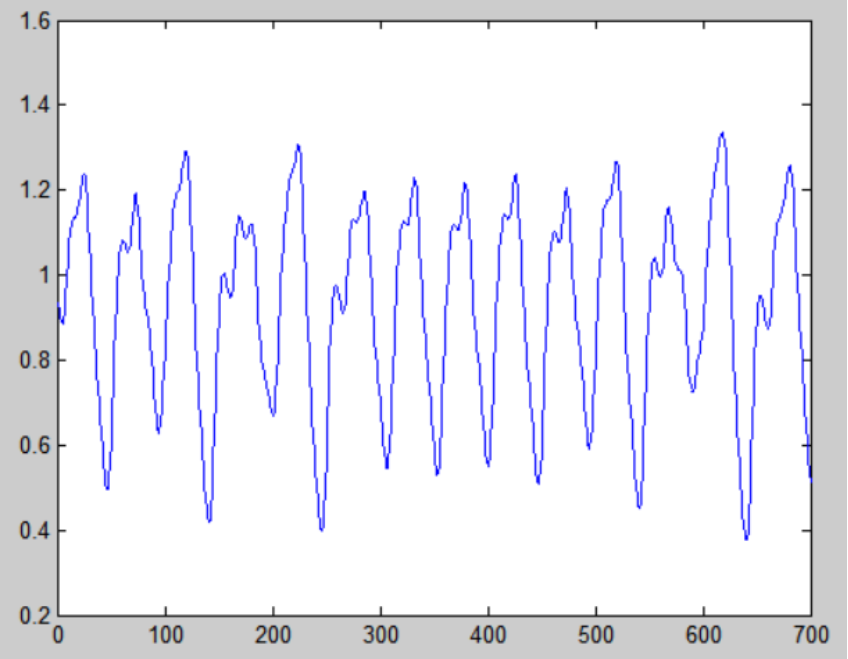

Figura 84 – Resultado esperado para a série temporal Mackey-Glass

A rede para a previsão desta série foi implementada com 2 entradas, 4 neurônios na camada oculta com função de ativação Tangente Hiperbólica e 1 na camada de saída, com função de ativação Rampa Limitada. A arquitetura pode ser vista na Figura 85.

O que se espera desta Rede Neural Artificial é que, a partir de duas entradas que representam os dados passados x(-2) e x(-1), a saída possa prever corretamente o próximo dado x(0). A partir da série de Mackey-Glass obtida no MATLAB, foram gerados 700 dados, 400 deles foram usados para treinamento e 300 para teste.

Assim como nos problemas anteriores, a RNA foi treinada pelo *Neural Network Toolbox* do MATLAB e os pesos e *bias* obtidos foram convertidos e colocados nas memórias da Rede Neural Artificial.

As instruções necessárias para a simulação deste problema são mostradas na Tabela 23.

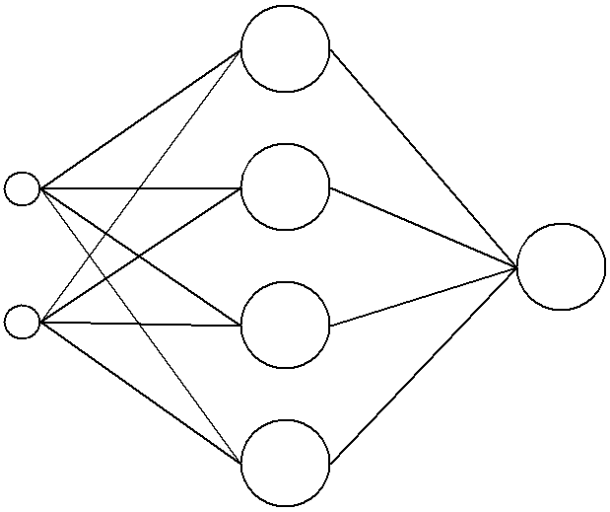

Figura 85 – Arquitetura para a previsão da série temporal Mackey-Glass

Tabela 23 – Instruções geradas para a Previsão de Série Temporal

Número da Instrução	Tipo de Camada	Quantidade de Neurônios na Camada	Presença de *Bias*	Função de Ativação	Instrução Gerada
1	Entrada	2	–	–	0000001000
2	Oculta	4	Sim	Tangente Hiperbólica	1000011111
3	Saída	1	Sim	Rampa	1100000101

O *testbench* foi então gerado e a RNA simulada. Os resultados para esta Rede Neural Artificial são mostrados na Tabela 24.

Tabela 24 – Resumo do resultado da simulação do problema de Previsão de Série Temporal

FPGA usada	EP4CE115F29C7
Número de neurônios implementados	20
Arquitetura da RNA	2 - 4 - 1 (5 neurônios usados)
Total de elementos lógicos	39.344 / 114.480 (34%)
Total de funções combinacionais	38.840 / 114.480 (34%)
Total de registradores dedicados	8.527 / 114.480 (7%)
Total de bits de memória	43.008 / 3.981.312 (1%)
Multiplicadores dedicados de 9 bits	0 / 532 (0%)
Pinos	335 / 529 (63%)
Número de ciclos de *clock*	59
Frequência máxima de operação	77.59 MHz

Não foi mostrada uma comparação entre este trabalho e o trabalho apresentado em (12) pois o autor não apresentou os resultados referentes ao erro da previsão.

Novamente, vale frisar que a área usada é a mesma da primeira e da segunda simulação devido à possibilidade de reconfiguração da RNA. A quantidade de ciclos usado

para a solução deste problema foi de 59 ciclos.

A carta de tempo para esta simulação é mostrada na Figura 86.

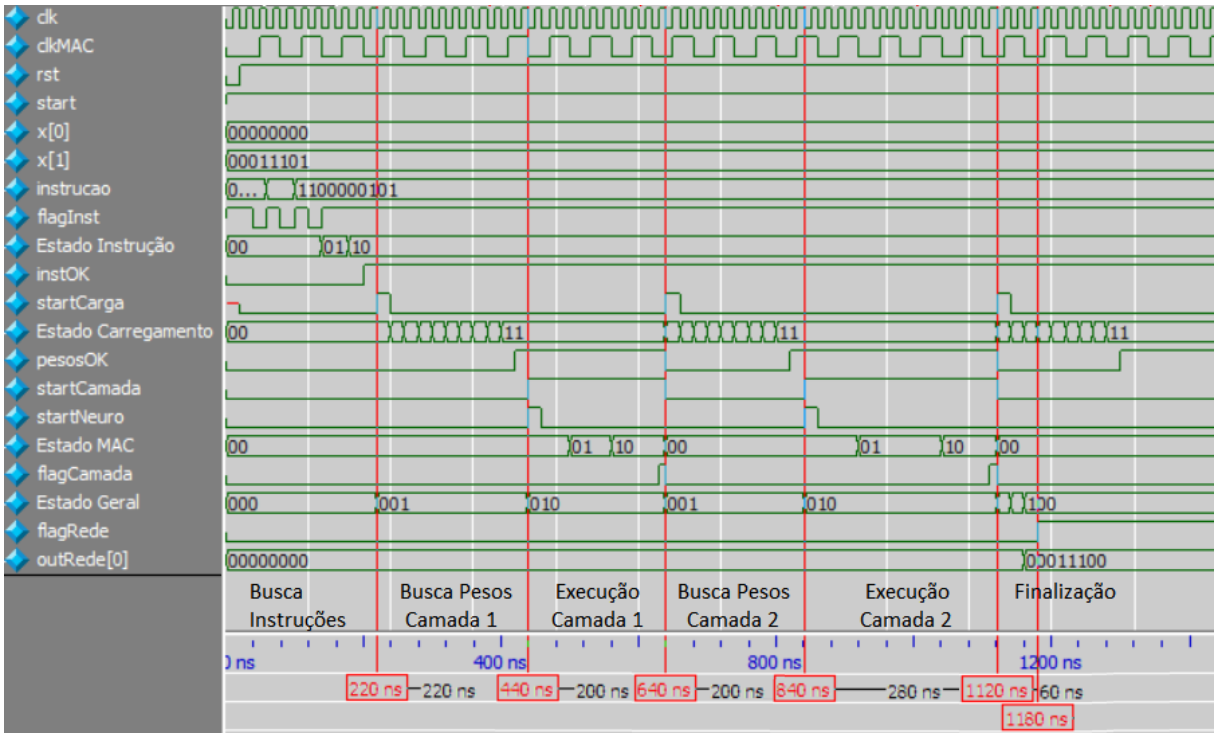

Figura 86 – Carta de Tempo para a série temporal Mackey-Glass

Este problema foi resolvido com uma RNA de duas camadas e poucos neurônios. Ao se observar a carta de tempo da simulação, nota-se que a busca de pesos e o cálculo da soma ponderada são feitos em poucos ciclos de *clock*. Novamente, na carta de tempo, a execução das camadas está marcada com o traço em vermelho. O período do *clock* de entrada usado é de 20 ns.

Inicialmente são gastos 11 ciclos de *clock* para que a Unidade de Instruções conclua sua execução.

O cálculo da quantidade de ciclos usados na primeira camada é feito com os dados $l_i = l_1, n_1 = 4$ e $e_1 = 2$. Assim, a quantidade de ciclos de *clock* esperada para a busca de pesos é $3 + 2.max\{4, 2\} = 11$ (Equação 3.2). Para a execução da camada são esperados $3(2 + arr(2/2)) = 9$ ciclos de *clk* (Equação 3.4).

Na segunda camada desta Rede Neural Artificial tem-se $l_i = l_2, n_2 = 1$ e $e_i = 4$. Assim, a quantidade de ciclos de *clock* esperada para a busca de pesos é $2+2.max\{1, 4\} = 10$ (Equação 3.2) e na execução dos neurônios são esperados $1 + 3(2 + arr(4/2)) = 12$ ciclos de *clock* (Equação 3.4).

Para a operação total da RNA são esperados 57 ciclos de *clock* e o valor observado na simulação foi de 59 ciclos de *clock*. Como já visto nos testes anteriores, a quantidade de ciclos calculada indica o mínimo de ciclos de *clock* gastos para a execução e pode variar

devido ao uso de *clocks* diferentes no sistema.

Esta Rede Neural Artificial foi testada para 300 valores de entrada. Os resultados obtidos neste problema foram exportados para o MATLAB e o gráfico mostrando a saída esperada e a obtida são mostrados na Figura 87.

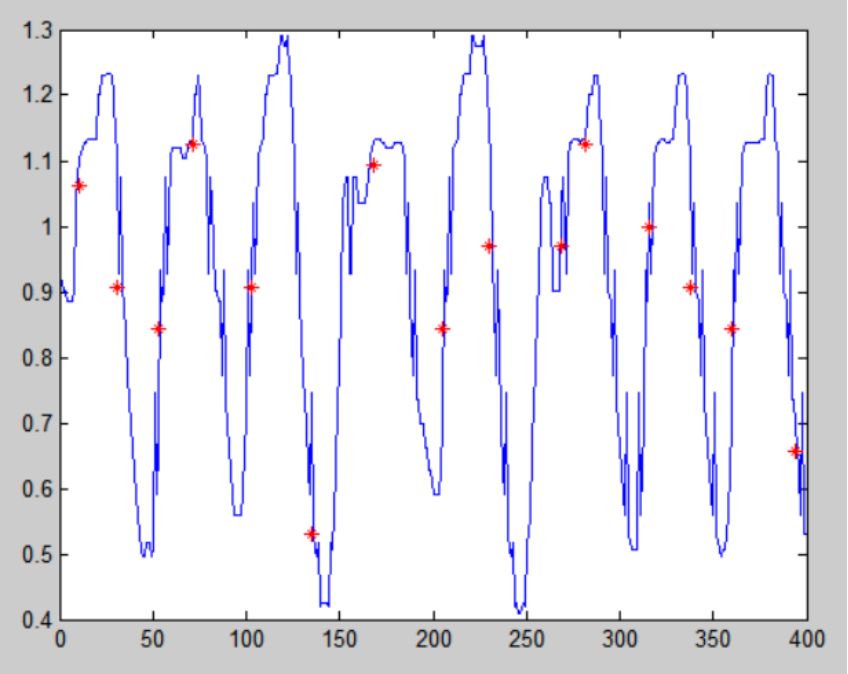

Figura 87 – Resultado da previsão da série temporal Mackey-Glass

A fim de facilitar a visualização, somente alguns pontos da simulação foram mostrados (asteriscos vermelhos na Figura 87). A média do erro obtido foi de 0,001344905, um bom resultado para esta RNA, já que esta é implementada com ponto fixo de 8 bits. Através da observação do erro e da Figura 87 pode-se notar que a previsão obtida foi próxima aos resultados esperados.

Vale ressaltar que em todos os testes aqui simulados a área usada pela RNA é a mesma, ou seja, não houve mudanças no hardware da RNA e não foram necessárias novas compilações do código para garantir a execução. Este fato permite que a Rede Neural Artificial criada seja modificada em campo, a tempo de execução, sem a necessidade de parar o funcionamento em que ela está inserida ou trocar parte deste sistema. Também, ao se observar o relatório de elementos usados na implementação, nota-se que nenhum multiplicador foi usado por esta proposta, o que permite que, em algum trabalho futuro, a RNA aqui desenvolvida seja implementada em um CI.

5 Conclusão

A implementação de Redes Neurais Artificiais em hardware permite a criação de sistemas de detecção de padrões, classificação, aproximação, diagnósticos, entre outros, que podem trabalhar em tempo real, com altas velocidades de operação. Porém, é importante que a RNA seja capaz de se adaptar a mudanças no meio em que está inserida, permitindo uma flexibilidade na arquitetura e, assim, fazendo com que o sistema evolua com mudanças no seu ambiente, alcançando melhores resultados. Este fato é especialmente vantajoso quando a RNA está em locais de difícil acesso, onde a mudança do hardware é complexa. Ter uma Rede Neural Artificial que permita a mudança na arquitetura sem a necessidade de troca no hardware pode trazer grandes vantagens. Exemplos deste tipo de sistemas são dispositivos médicos implantados, dispositivos para comunicação móvel ou até dispositivos para missões espaciais.

Na literatura encontram-se algumas implementações de RNAs reconfiguráveis por meio da multiplexação das camadas da Rede Neural Artificial. Com base nisso, este trabalho apresentou uma nova implementação de RNA Reconfigurável que faz uso da reutilização das camadas da Rede Neural Artificial. A implementação é independente do modelo da FPGA, já que não usou blocos proprietários, como os multiplicadores.

Esta independência entre a implementação da Rede Neural Artificial e da FPGA é importante para garantir que em trabalhos futuros seja possível desenvolver um circuito integrado que contenha a RNA, já que esta implementação em FPGA é uma primeira fase desta pesquisa.

No Capítulo 3 toda a arquitetura aqui desenvolvida foi descrita. É importante destacar o uso do multiplicador Vedic, a fim de fazer a implementação independente da FPGA e possível de implementação em CIs. Este modelo do multiplicador se mostrou eficiente frente a outros modelos, já que seu atraso e seu consumo de potência são pequenos. Para compensar a maior área necessárias pelos multiplicadores, somente dois deles foram implementados pelos neurônios e a multiplicação de cada uma das entradas pelos seus respectivos pesos é feita de modo sequencial. Isto traz maiores tempos de execução para a RNA quando implementada em FPGA.

Outro fator determinante no tempo de execução é que o multiplicador aqui usado tem um atraso maior que o multiplicador disponibilizado pela FPGA. Porém, somente o multiplicador necessita de períodos de *clock* maior. Assim sendo, para evitar comprometer todo o sistema com uma frequência de operação baixa, foram usados dois sinais de *clock*: um de maior frequência para toda a operação da RNA e um de frequência 3 vezes menor para a execução do multiplicador.

Todo controle da RNA é feito através de máquinas de estado, sem o uso de software. Isto tira a necessidade de se ter um processador no sistema em que a Rede Neural Artificial está inserida, diminuindo a área do circuito.

A Rede Neural Artificial foi submetida a três testes práticos: aproximação de função, classificação de padrões e previsão de série temporal. A aproximação da função Sinc além de ter sido simulada no ModelSim, também foi implementada na FPGA e os resultados reais foram avaliados.

O problema de classificação de flores íris foi testado e comparado com o artigo em que foi baseado. O erro atingido por esta implementação foi maior que o do artigo base. Este aumento no erro é justificado pela notação numérica usada, já que este trabalho usa ponto fixo de 8 bits e o artigo comparado usa ponto flutuante, que traz melhores precisões. Porém esta melhor resolução vem a custo de área do circuito, já que mesmo implementando somente 14 neurônios a RNA do artigo comparado tem uma área maior do que a RNA aqui proposta com 20 neurônios. Outra vantagem clara deste trabalho sobre o artigo base é a possibilidade de reconfiguração da Rede Neural Artificial.

O terceiro e último problema testado foi a previsão da série temporal de Mackey-Glass. Este problema também atingiu um baixo valor para o erro, o que mostra que a RNA aqui proposta pode ser usada de maneira eficiente para soluções de previsão.

Em geral a Rede Neural Artificial Reconfigurável implementada obteve resultados satisfatórios, levando-se em consideração a quantidade de bits das palavras usada. Também o fato dela ser independente do modelo da FPGA a torna propícia para implementações em circuitos integrados, além do fato de não necessitar de software para o controle da execução. Outro ponto importante é o fato de que várias arquiteturas possam ser testadas no mesmo circuito. Além disso, a RNA implementada é simples e feita de forma que a modificação seja simples.

Este trabalho gerou uma publicação no *"Tenth International Caribbean Conference on Devices, Circuits And Systems"(ICCDCS 2017)*, realizado em Cozumel, no México. O artigo aceito pode ser visto nos Anexos deste trabalho.

5.1 Trabalhos Futuros

Durante o desenvolvimento deste trabalho alguns pontos foram levantados para que se melhore a RNA implementada:

- Aumentar a quantidade de multiplicadores usados para que o tempo gasto na execução seja menor. Porém, este aumento traz um custo na área total da RNA e, para que este aumento não afete de forma significativa a área do circuito, seria necessário um estudo da relação entre área e tempo de execução da RNA;

- Aumentar o número de bits usados para as entradas e saídas da RNA, para melhorar os resultados, já que a precisão do circuito limita os resultados obtidos. Assim como o aumento da quantidade de multiplicadores, o aumento no número de bits usados leva a uma maior área do circuito;
- A fim de melhorar o atraso nos multiplicadores Vedic, encontrar um somador que seja mais rápido e se adapte bem ao circuito;
- Aumentar o valor da frequência máxima de operação da RNA;
- Implementar o treinamento on-line no hardware da RNA. Este ajuste permite que a RNA aqui proposta possa se adaptar totalmente sem a necessidade de software externo;
- Testar a RNA em uma aplicação real mais complexa;
- Explorar mais os testes da RNA na FPGA;
- Criar um circuito integrado com a RNA.

Referências

1 BASHEER, M. H. I. A. Artificial neural networks: Fundamentals computing design and application. *J. Microbio. Methods*, v. 43, p. 3–31, 2000. Citado 7 vezes nas páginas 14, 18, 19, 20, 27, 28 e 29.

2 AL., R. B. et. *Neural Networks in Healthcare: Potential and Challenges*. [S.l.]: Idea Group Inc., 2006. Citado 4 vezes nas páginas 14, 18, 28 e 37.

3 MARTINS, R. S. *Implementação em hardware de redes neurais artificiais com topologia configurável*. Dissertação (Mestrado) — Universidade Estadual do Rio de Janeiro, Programa de P ós-Graduação em Engenharia Eletrônica, 2010. Citado 8 vezes nas páginas 14, 15, 18, 19, 22, 25, 32 e 33.

4 LIAO, Y. Neural networks in hardware: A survey. *Department of Computer Science, University of California*, 2001. Citado 9 vezes nas páginas 14, 17, 18, 26, 27, 32, 33, 34 e 35.

5 DAS, S.; PEDRONI, B. Implementation of a restricted boltzmann machine in a spiking neural network. Citado na página 14.

6 GUO, L. et al. Automatic epileptic seizure detection in EEGs based on line length feature and artificial neural networks. *J. Neurosci. Methods*, v. 191, p. 101–109, 2010. Citado na página 14.

7 DIAS, F. M.; ANTUNESA, A.; MOTAB, A. M. Artificial neural networks: A review of commercial hardware. *Eng. Appl. Artif. Intell.*, v. 17, n. 8, p. 945–952, 2004. Citado 2 vezes nas páginas 14 e 32.

8 MISRA, J.; SAHA, I. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing*, v. 74, n. 13, p. 239–255, 2010. Citado na página 14.

9 COLLINS, D. R. e. a. Neural network hardware: What does the future hold? *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, v. 1, 1991. Citado 3 vezes nas páginas 14, 22 e 56.

10 ALI, H. K.; MOHAMMED, E. Z. Design artificial neural network using FPGA. *IJCSNS*, v. 10, n. 8, 2010. Citado na página 14.

11 BAHOURA, M.; CHAN-WANG, P. M. FPGA-implementation of high-speed MLP neural network. *Proceedings of The 18th International Conference on Electronics Circuits and Systems (ICECS)*, 2011. Citado na página 14.

12 YOUSSEF, A.; MOHAMMED, K.; A., N. Two novel generic, reconfigurable neural network FPGA architectures. *Artificial Intelligence with Applications in Engineering and Technology (ICAIET)*, 2014. Citado 9 vezes nas páginas 15, 35, 36, 37, 38, 55, 104, 114 e 116.

13 BIRDI, Y.; AURORA, T.; ARORA, P. Study of artificial neural networks and neural implants. *International Journal on Recent and innovation Trends in Computing and*

Communication, v. 1, n. 4, p. 258–62, 2013. Citado 5 vezes nas páginas 17, 18, 26, 27 e 28.

14 HAYKIN, S. S. et al. *Neural networks and learning machines*. [S.l.]: Pearson Upper Saddle River, NJ, USA, 2009. v. 3. Citado 6 vezes nas páginas 17, 18, 20, 21, 26 e 30.

15 CAVUSLU, M.; KARAKUZU, C.; SAHIN, S. Neural network hardware implementation using FPGA. In: *Neural Information Processing*. [S.l.: s.n.], 2006. Citado 9 vezes nas páginas 17, 18, 25, 26, 32, 33, 35, 36 e 37.

16 OMONDI, A. R.; RAJAPAKSE, J. C. *FPGA implementations of neural networks*. [S.l.]: Springer, 2006. v. 365. Citado 8 vezes nas páginas 17, 18, 24, 28, 35, 36, 43 e 56.

17 STEFANOWSKI, J. Artificial neural networks–basics of MLP, RBF and Kohonen Networks. *Lecture 13 in Data Mining*, 2010. Citado 2 vezes nas páginas 17 e 18.

18 DEOTALE, P. D.; DOLE, L. Design of FPGA based general purpose neural network. In: IEEE. *Information Communication and Embedded Systems (ICICES), 2014 International Conference on*. [S.l.], 2014. p. 1–5. Citado 8 vezes nas páginas 17, 18, 26, 32, 33, 34, 35 e 36.

19 MUTHURAMALINGAM, A.; HIMAVATHI, S.; SRINIVASAN, E. Neural network implementation using FPGA: issues and application. *International journal of information technology*, v. 4, n. 2, p. 86–92, 2008. Citado 3 vezes nas páginas 17, 35 e 36.

20 DREW, P. J.; MONSON, J. R. T. Implementation of a restricted boltzmann machine in a spiking neural network. *Surgery*, v. 127, p. 3–11, 2000. Citado 5 vezes nas páginas 18, 19, 20, 27 e 29.

21 PANICKER, M.; BABU, C. Efficient FPGA implementation of sigmoid and bipolar sigmoid activation functions for multilayer perceptrons. *IOSR Journal of Engineering*, p. 1352–1356, 2012. Citado 5 vezes nas páginas 18, 35, 37, 45 e 55.

22 ZURADA, J. M. *Introduction to artificial neural systems*. [S.l.]: West St. Paul, 1992. v. 8. Citado 3 vezes nas páginas 19, 20 e 29.

23 MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, Springer, v. 5, n. 4, p. 115–133, 1943. Citado na página 19.

24 SUTTISINTHONG, N. et al. Selection of proper activation functions in back-propagation neural network algorithm for single-circuit transmission line. In: *Proceedings of the International MultiConference of Engineers and Computer Scientists*. [S.l.: s.n.], 2014. v. 2. Citado 2 vezes nas páginas 21 e 22.

25 SARTIN, M. A.; SILVA, A. C. R. da. Aproximação da função tangente hiperbólica em hardware. In: *Proceedings of International Conference on Engineering and Computer Education*. [S.l.: s.n.], 2013. v. 8, p. 361–365. Citado 4 vezes nas páginas 21, 22, 25 e 56.

26 SARTIN, M. A.; SILVA, A. C. D. Approximation of hyperbolic tangent activation function using hybrid methods. In: IEEE. *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on*. [S.l.], 2013. p. 1–6. Citado 3 vezes nas páginas 21, 43 e 44.

27 RASCHKA, S. *Python machine learning*. [S.l.]: Packt Publishing Ltd, 2015. Citado 4 vezes nas páginas 22, 24, 25 e 26.

28 ABROL, S.; MAHAJAN, R. Implementation of single artificial neuron using various activation functions and XOR gate on FPGA chip. In: IEEE. *Advances in Computing and Communication Engineering (ICACCE), 2015 Second International Conference on*. [S.l.], 2015. p. 118–123. Citado 2 vezes nas páginas 22 e 43.

29 BISHOP, C. M. *Neural networks for pattern recognition*. [S.l.]: Oxford university press, 1995. Citado na página 25.

30 ORHAN, U.; HEKIM, M.; OZER, M. EEG signals classification using the k-means clustering and a multilayer perceptron neural network model. *Expert Systems with Applications*, Elsevier, v. 38, n. 10, p. 13475–13481, 2011. Citado na página 29.

31 GOMPERTS, A.; UKIL, A.; ZURFLUH, F. Development and implementation of parameterized FPGA-based general purpose neural networks for online applications. *IEEE Transactions on Industrial Informatics*, IEEE, v. 7, n. 1, p. 78–89, 2011. Citado na página 31.

32 DAPONTE, P.; GRIMALDI, D. Artificial neural networks in measurements. *Measurement*, Elsevier, v. 23, n. 2, p. 93–115, 1998. Citado 4 vezes nas páginas 32, 33, 34 e 35.

33 KEULEN, E. van et al. Neural network hardware performance criteria. In: IEEE. *Neural Networks, 1994. IEEE World Congress on Computational Intelligence., 1994 IEEE International Conference on*. [S.l.], 1994. v. 3, p. 1885–1888. Citado na página 32.

34 GOSER, K. F. Implementation of artificial neural networks into hardware: Concepts and limitations. *Mathematics and computers in simulation*, Elsevier, v. 41, n. 1, p. 161–171, 1996. Citado 2 vezes nas páginas 32 e 34.

35 ALIPPI, C.; NIGRI, M. E. Hardware requirements to digital VLSI implementation of neural networks. In: IEEE. *Neural Networks, 1991. 1991 IEEE International Joint Conference on*. [S.l.], 1991. p. 1873–1878. Citado 2 vezes nas páginas 32 e 35.

36 MOERLAND, P.; FIESLER, E. *Neural network adaptations to hardware implementations*. [S.l.], 1997. Citado 3 vezes nas páginas 32, 33 e 35.

37 LINDSEY, C. S.; LINDBLAD, T. Survey of neural network hardware. In: INTERNATIONAL SOCIETY FOR OPTICS AND PHOTONICS. *SPIE's 1995 Symposium on OE/Aerospace Sensing and Dual Use Photonics*. [S.l.], 1995. p. 1194–1205. Citado 3 vezes nas páginas 32, 33 e 34.

38 NIKHIL, S.; LAKSHMI, M. P. V. Implementation of a high speed multiplier desired for high-performance applications using kogge stone adder. In: IEEE. *Inventive Computation Technologies (ICICT), International Conference on*. [S.l.], 2016. v. 1, p. 1–4. Citado 9 vezes nas páginas 32, 35, 36, 43, 47, 48, 50, 51 e 53.

39 AL-NSOUR, M.; ABDEL-ATY-ZOHDY, H. S. Implementation of programmable digital sigmoid function circuit for neuro-computing. In: IEEE. *Circuits and Systems, 1998. Proceedings. 1998 Midwest Symposium on*. [S.l.], 1998. p. 571–574. Citado 3 vezes nas páginas 32, 43 e 44.

40 ROJAS, R. *Neural networks: a systematic introduction*. [S.l.]: Springer Science & Business Media, 2013. Citado 4 vezes nas páginas 33, 34, 37 e 55.

41 JR, T. M.; WALKER, D. J.; SIVILOTTI, M. A. A digital neural network architecture for VLSI. In: IEEE. *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*. [S.l.], 1990. p. 545–550. Citado 2 vezes nas páginas 34 e 35.

42 TOMMISKA, M. Efficient digital implementation of the sigmoid function for reprogrammable logic. *IEEE Proc.-Comput. Digital Tech*, v. 150, n. 6, p. 403–411, 2003. Citado 7 vezes nas páginas 35, 36, 43, 44, 45, 46 e 56.

43 EL-MADANY, H. T. et al. Design of FPGA based neural network controller for earth station power system. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, v. 10, n. 2, p. 281–290, 2012. Citado 2 vezes nas páginas 35 e 36.

44 AGO, Y.; ITO, Y.; NAKANO, K. An FPGA implementation for neural networks with the FDFM processor core approach. *International Journal of Parallel, Emergent and Distributed Systems*, Taylor & Francis, v. 28, n. 4, p. 308–320, 2013. Citado na página 35.

45 A.SILVA A. D. D. NETO, e. a. C. Definition of an architecture to configure artificial neural networks topologies using partial reconfiguration in FPGA. *IEEE Latin America Transactions*, p. 2094–2100, 2015. Citado 4 vezes nas páginas 35, 37, 41 e 55.

46 SHYU, K.-K. et al. Total design of an FPGA-based brain–computer interface control hospital bed nursing system. *IEEE Transactions on Industrial Electronics*, IEEE, v. 60, n. 7, p. 2731–2739, 2013. Citado na página 35.

47 GRANADO, J. et al. Using FPGAs to implement artificial neural networks. In: IEEE. *Electronics, Circuits and Systems, 2006. ICECS'06. 13th IEEE International Conference on*. [S.l.], 2006. p. 934–937. Citado 3 vezes nas páginas 35, 36 e 56.

48 CORIC, S.; LATINOVIC, I.; PAVASOVIC, A. A neural network FPGA implementation. In: IEEE. *Neural Network Applications in Electrical Engineering, 2000. NEUREL 2000. Proceedings of the 5th Seminar on*. [S.l.], 2000. p. 117–120. Citado 2 vezes nas páginas 35 e 36.

49 BIRADAR, R. G. et al. FPGA implementation of a multilayer artificial neural network using system-on-chip design methodology. In: IEEE. *Cognitive Computing and Information Processing (CCIP), 2015 International Conference on*. [S.l.], 2015. p. 1–6. Citado 2 vezes nas páginas 35 e 36.

50 ZHU, J.; SUTTON, P. FPGA implementations of neural networks–a survey of a decade of progress. *Field Programmable Logic and Application*, Springer, p. 1062–1066, 2003. Citado na página 35.

51 YAMUNA, S. et al. *High speed neuron implementation using Vedic Mathematics*. [S.l.]: Discovery, 2015. 25–32 p. Citado 2 vezes nas páginas 35 e 36.

52 JIMENEZ, R. et al. Implementation of a neural network for digital pulse shape analysis on a FPGA for on-line identification of heavy ions. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, Elsevier, v. 674, p. 99–104, 2012. Citado 2 vezes nas páginas 35 e 36.

53 SILVA, R. M. D.; NEDJAH, N.; MOURELLE, L. D. M. Hardware implementations of MLP artificial neural networks with configurable topology. *Journal of Circuits, Systems, and Computers*, World Scientific, v. 20, n. 03, p. 417–437, 2011. Citado 2 vezes nas páginas 35 e 36.

54 FINKER I. DEL CAMPO, J. E. R.; DOCTOR, F. Multilevel adaptive neural network architecture for implementing single-chip intelligent agents on FPGAs. *International Joint Conference on Neural Networks*, 2013. Citado 2 vezes nas páginas 35 e 37.

55 BONNICI, M. et al. Artificial neural network optimization for FPGA. In: IEEE. *Electronics, Circuits and Systems, 2006. ICECS'06. 13th IEEE International Conference on*. [S.l.], 2006. p. 1340–1343. Citado na página 36.

56 KAPOOR, R.; RAJPUT, G. Logic gates realization using spiking neural network and vedic maths-a comprehensive study. 2015. Citado na página 36.

57 ELDREDGE, J. G.; HUTCHINGS, B. L. Density enhancement of a neural network using FPGAs and run-time reconfiguration. In: IEEE. *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on*. [S.l.], 1994. p. 180–188. Citado na página 36.

58 MAKWANA, H. H.; SHAH, D. J.; GANDHI, P. P. FPGA implementation of artificial neural network. *International Journal of Emerging Technology and Advanced Engineering*, Citeseer, v. 3, n. 1, p. 672–679, 2013. Citado na página 36.

59 NEDJAH, N. et al. Dynamic MAC-based architecture of artificial neural networks suitable for hardware implementation on fpgas. *Neurocomputing*, Elsevier, v. 72, n. 10, p. 2171–2179, 2009. Citado na página 37.

60 YOUSSEF, A.; MOHAMMED, K.; NASAR, A. A reconfigurable, generic and programmable feed forward neural network implementation in FPGA. In: IEEE. *Computer Modelling and Simulation (UKSim), 2012 UKSim 14th International Conference on*. [S.l.], 2012. p. 9–13. Citado na página 37.

61 SATYANARAYANA, S.; TSIVIDIS, Y. P.; GRAF, H. P. A reconfigurable analog VLSI neural network chip. In: *NIPS*. [S.l.: s.n.], 1989. p. 758–768. Citado 2 vezes nas páginas 37 e 38.

62 NEDJAH, N.; SILVA, R. M. da; MOURELLE, L. de M. Compact yet efficient hardware implementation of artificial neural networks with customized topology. *Expert Systems with Applications*, Elsevier, v. 39, n. 10, p. 9191–9206, 2012. Citado 3 vezes nas páginas 37, 40 e 55.

63 PRADO, R. N. A. et al. Arquitetura para aplicações genéricas de redes neurais artificiais com fácil configuração de topologias multilayer perceptron em FPGA. In: *10th Brazilian Congress on Computational Intelligence (CBIC 2011)*. [S.l.: s.n.], 2011. Citado na página 37.

64 PRADO, R. N. d. A. *Desenvolvimento de uma arquitetura em hardware prototipada em FPGA para aplicações genéricas utilizando redes neurais artificiais embarcadas*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, 2011. Citado na página 37.

65 HIMAVATHI, D. A. S.; MUTHURAMALINGAM, A. Feedforward neural network implementation in FPGA using layer multiplexing for effective resource utilization. *IEEE Trans. Neural Networks*, v. 18, p. 880–888, 2007. Citado 2 vezes nas páginas 37 e 55.

66 ZAMANLOOY, B.; MIRHASSANI, M. Efficient VLSI implementation of neural networks with hyperbolic tangent activation function. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, IEEE, v. 22, n. 1, p. 39–48, 2014. Citado 5 vezes nas páginas 42, 43, 44, 45 e 46.

67 BASTERRETXEA, K.; TARELA, J.; CAMPO, I. D. Digital design of sigmoid approximator for artificial neural networks. *Electronics Letters*, IET, v. 38, n. 1, p. 35–37, 2002. Citado 4 vezes nas páginas 42, 43, 44 e 45.

68 TISAN, A. et al. Digital implementation of the sigmoid function for FPGA circuits. *ACTA Technica Napocensis*, v. 50, n. 2, p. 15–20, 2009. Citado 2 vezes nas páginas 42 e 43.

69 HAVEL, V.; VLCEK, K. Computation of a nonlinear squashing function in digital neural networks. In: IEEE. *Design and Diagnostics of Electronic Circuits and Systems, 2008. DDECS 2008. 11th IEEE Workshop on*. [S.l.], 2008. p. 1–4. Citado na página 43.

70 AMIN K. M. CURTIS, B. R. H.-G. H. Piecewise linear approximation applied to nonlinear function of a neural network. *IEEE Proceedings — Circuits Devices and Systems*, v. 144, n. 6, p. 313–317, 1997. Citado 4 vezes nas páginas 43, 44, 45 e 86.

71 NAMIN, A. H. et al. Artificial neural networks activation function HDL coder. In: IEEE. *Electro/Information Technology, 2009. eit'09. IEEE International Conference on*. [S.l.], 2009. p. 389–392. Citado 4 vezes nas páginas 43, 44, 45 e 46.

72 ALIPPI, C.; STORTI-GAJANI, G. Simple approximation of sigmoidal functions: realistic design of digital neural networks capable of learning. In: IEEE. *Circuits and Systems, 1991., IEEE International Sympoisum on*. [S.l.], 1991. p. 1505–1508. Citado 2 vezes nas páginas 43 e 44.

73 MYERS, D.; HUTCHINSON, R. Efficient implementation of piecewise linear activation function for digital VLSI neural networks. *Electronics Letters*, v. 25, p. 1662, 1989. Citado na página 44.

74 BASTERRETXEA, K.; TARELA, J.; CAMPO, I. D. Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons. *IEE Proceedings-Circuits, Devices and Systems*, IET, v. 151, n. 1, p. 18–24, 2004. Citado na página 45.

75 ZHANG, M.; VASSILIADIS, S.; DELGADO-FRIAS, J. G. Sigmoid generators for neural computing using piecewise approximations. *IEEE transactions on Computers*, IEEE, v. 45, n. 9, p. 1045–1049, 1996. Citado na página 45.

76 KWAN, H. K. Simple sigmoid-like activation function suitable for digital hardware implementation. *Electronics letters*, IET, v. 28, n. 15, p. 1379–1380, 1992. Citado na página 45.

77 SINGH, K. N.; TARUNKUMAR., H. A review on various multipliers designs in VLSI. *2015 Annual IEEE India Conference (INDICON).*, 2015. Citado 4 vezes nas páginas 47, 48, 50 e 51.

78 SONI, P. et al. Implementation of 16x16 bit multiplication algorithm by using vedic mathematics over booth algorithm. *IJRET : International Journal of Research in Engineering and Technology*, v. 04, n. 05, 2015. Citado 2 vezes nas páginas 47 e 48.

79 PALDURAI, K. et al. Implementation of MAC using area efficient and reduced delay vedic multiplier targeted at FPGA architectures. In: IEEE. *Communication and Network Technologies (ICCNT), 2014 International Conference on*. [S.l.], 2014. p. 238–242. Citado na página 47.

80 PREMANANDA SAMARTH S. PAI, B. S. S. S. B. B. Design and implementation of 8-bit vedic multiplier. *International Journal of Advanced Research in Electrical Electronics and Instrumentation Engineering*, v. 2, n. 12, p. 5877–5882, 2013. Citado 3 vezes nas páginas 47, 76 e 78.

81 RAO, J.; DUBEY, S. A high speed wallace tree multiplier using modified booth algorithm for fast arithmetic circuits. *IOSR Journal of Electronics and Communication Engineering (IOSRJECE)*, v. 3, n. 1, p. 07–11, 2012. Citado na página 47.

82 SHARMA, B.; BAKSHI., A. Comparison of 24x24 bit multipliers for various performance parameters. *International Conference on Advent Trends in Engineering, Science and Technology (ICATEST)*, 2015. Citado 3 vezes nas páginas 47, 49 e 53.

83 TIWARI, H. D. et al. Multiplier design based on ancient indian vedic mathematics. In: IEEE. *SoC Design Conference, 2008. ISOCC'08. International*. [S.l.], 2008. v. 2, p. II–65. Citado 2 vezes nas páginas 48 e 50.

84 SWEE, K. L. S.; HIUNG, L. H. Performance comparison review of 32-bit multiplier designs. In: IEEE. *Intelligent and Advanced Systems (ICIAS), 2012 4th International Conference on*. [S.l.], 2012. v. 2, p. 836–841. Citado 2 vezes nas páginas 48 e 51.

85 K, P.; HARIHARAN, K. Implementation of signed vedic multiplier target at FPGA architectures. *ARPN Journal of Engeneering and Applied Sciences*, v. 10, n. 05, 2015. Citado na página 48.

86 SATISH, D.; RAJU, B. R. A high speed 16 * 16 multiplier based on urdhva tiryakbhyam sutra. *IJSEAT*, v. 1, n. 5, p. 126–132, 2013. Citado 3 vezes nas páginas 48, 49 e 53.

87 WALLACE, C. S. A suggestion for a fast multiplier. *IEEE Transactions on electronic Computers*, IEEE, n. 1, p. 14–17, 1964. Citado na página 50.

88 OTHMAN, M.; ALI, M. M. et al. High performance parallel multiplier using wallace-booth algorithm. In: IEEE. *Semiconductor Electronics, 2002. Proceedings. ICSE 2002. IEEE International Conference on*. [S.l.], 2002. p. 433–436. Citado na página 50.

89 ANJANA, R. et al. Implementation of vedic multiplier using kogge-stone adder. In: IEEE. *Embedded Systems (ICES), 2014 International Conference on*. [S.l.], 2014. p. 28–31. Citado na página 51.

90 PICHHODE, K. et al. FPGA implementation of efficient vedic multiplier. In: IEEE. *Information Processing (ICIP), 2015 International Conference on*. [S.l.], 2015. p. 565–570. Citado 3 vezes nas páginas 51, 53 e 76.

91 BANSAL, Y.; MADHU, C.; KAUR, P. High speed vedic multiplier designs-a review. In: IEEE. *Engineering and Computational Sciences (RAECS), 2014 Recent Advances in*. [S.l.], 2014. p. 1–6. Citado 3 vezes nas páginas 53, 76 e 81.

92 GRAUPE, D. *Principles of artificial neural networks*. [S.l.]: World Scientific, 2013. v. 7. Citado na página 55.

93 HOLT, J. L.; BAKER, T. E. Back propagation simulations using limited precision calculations. In: IEEE. *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*. [S.l.], 1991. v. 2, p. 121–126. Citado na página 56.

94 POHOKAR, S. et al. Design and implementation of 16× 16 multiplier using vedic mathematics. In: IEEE. *Industrial Instrumentation and Control (ICIC), 2015 International Conference on*. [S.l.], 2015. p. 1174–1177. Citado na página 76.

95 FERREIRA, E. N. D. S. B. A. P. D. A. A high performance full pipelined arquitecture of MLP neural networks in FPGA. *Proc. IEEE 17th Int. Conf. Electron. Circuits Syst.*, p. 742–745, 2010. Citado 4 vezes nas páginas 104, 110, 111 e 114.

96 LICHMAN, M. *UCI Machine Learning Repository*. 2013. Disponível em: <http://archive.ics.uci.edu/ml>. Citado na página 110.

97 JAFRI, Y. Z. et al. Chaotic time series prediction and mackey-glass simulation with fuzzy logic. *International Journal of Physical Sciences*, Academic Journals, v. 7, n. 17, p. 2596–2606, 2012. Citado na página 115.

Apêndices

APÊNDICE A – Aproximação da Função Logística Sigmóide

As equações que definem a parte positiva da aproximação da função Sigmóide Logística são mostradas a seguir.

$$y = \begin{cases} 0,25x + 0,5, & 0 \leq x < 1 \\ 0,125x + 0,625, & 1 \leq x < 2,5 \\ 0,03125x + 0,859375, & 2,5 \leq x < 4,5 \\ 1, & x \geq 4,5 \end{cases} \tag{A.1}$$

A parte com entradas negativas é obtida através da relação $y(-x) = 1 - y(x)$.

Cada equação que define um intervalo é implementada separadamente. Esta implementação é feita através de blocos lógicos por meio de uma transformação direta, ou seja, os resultados possíveis são analisados e colocados diretamente na saída.

Para todos os intervalos o sinal de entrada de 32 bits é definido como:

$x_{31}x_{30}x_{29}x_{28}x_{27}x_26x_{25}x_{24}x_{23}x_{22}x_{21}x_{20}$, $x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2x_1x_0$

Cada intervalo será analisado separadamente a seguir.

Intervalo 1: $0,25x + 0,5$*;* $0 \leq x < 1$

Para este intervalo, o menor valor possível de entrada é 0 e o limite superior é o valor imediatamente menor que 1, que nesta notação é representado como

$$000000000000, 11111111111111111111$$

Pode-se observar que os bits 20 a 31 são sempre iguais a 0 neste intervalo. O sinal de entrada é mostrado a seguir:

$$000000000000, x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2x_1x_0$$

A multiplicação 0,25x pode ser substituída por dois deslocamentos para a direita. Após dois deslocamentos para a direita, o sinal se torna:

$$000000000000, 00x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2$$

Em seguida deve-se somar 0,5 ao valor obtido acima. Como 0,5 é representado com 32 bits da forma $000000000000, 10000000000000000000$, a adição só afeta o bit imediatamente depois da vírgula. Assim o resultado é para esta equação é:

$$000000000000, 10x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2$$

Como a saída é representada por somente 8 bits, o resultado sinal fica:

$$000,\ 10x_{19}x_{18}x_{17}$$

Este resultado pode ser implementado por simples atribuições.

Intervalo 2: $0,125x + 0,625$*;* $1 \leq x < 2,5$

Este intervalo é definido entre 1 e o valor imediatamente inferior a 2,5, ou seja, entre $000000000001,00000000000000000000$ e $000000000010,01111111111111111111$. Neste intervalo os bits 22 a 31 são iguais a 0 e o restante do sinal varia. Este intervalo pode ser definido como:

$$0000000000x_{21}x_{20},\ x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2x_1x_0$$

A multiplicação por 0,125 pode ser feita através de 3 deslocamentos para a direita e o sinal obtido após essa multiplicação é:

$$000000000000,\ 0x_{21}x_{20}x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3$$

A seguir deve-se somar 0,625 ao resultado obtido acima. Este valor é representado por $000000000000,10100000000000000000$, ou seja, os três bits imediatamente depois da vírgula serão afetados pela adição. Tomando os bits 22 a 18, a sequência de valores possíveis é:

Tabela 25 – Intervalo 2 da Aproximação da Logística Sigmóide - Valores possíveis

x	x_{22}	x_{21}	x_{20}	x_{19}	x_{18}
1	0	0	1	0	0
1,25	0	0	1	0	1
1,5	0	0	1	1	0
1,75	0	0	1	1	1
2	0	1	0	0	0
2,25	0	1	0	0	1

Nota-se que os bits 21 e 20 da entrada assumem valores 001 ou 010. Assim, somar 101 a esses valores dá como resultado 110 ou 111. Levando esses valores para uma tabela verdade:

Tabela 26 – Intervalo 2 da Aproximação da Logística Sigmóide - Valores possíveis

Entradas			Saídas		
x_{22}	x_{21}	x_{20}	y_{19}	y_{18}	y_{17}
0	0	1	1	1	0
0	1	0	1	1	1

Observa-se pela tabela verdade que os bits 19 e 18 da saída sempre serão 1 e que o bit 17 será igual ao inverso do bit 20. Os demais bits não são afetados pela soma.

Assim, a saída de 8 bits pode ser representada por:

$$000,\ 11\bar{x_{20}}x_{19}x_{18}$$

Intervalo 3: $0,03125x + 0,859375$*;* $2,5 \leq x < 4,5$

Este terceiro intervalo é definido entre $000000000010,1000000000000000000$ e $000000000100,0111111111111111111$. Os bits 23 a 31 são iguais a 0 e o restante do sinal varia, como mostrado a seguir:

$$000000000x_{22}x_{21}x_{20},\ x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2x_1x_0$$

A multiplicação por 0,03125 é obtida através de 5 deslocamentos para a direira e o resultado pode ser visto a seguir:

$$000000000000,\ 00x_{22}x_{21}x_{20}x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5$$

Deve-se então somar 0,859375 ao resultado do deslocamento. Este valor é representado por $000000000000,1101100000000000000$. A análise se torna mais simples se feita bit a bit. Como a saída terá apenas 8 bits, pode-se truncar o resultado do deslocamento para $000,00x_{22}x_{21}x_{20}$ e o valor a ser somado torna-se $000,11011$. Assim, deve-se analisar a soma somente de 5 bits. A Tabela 27 é a tabela verdade para este caso:

Tabela 27 – Intervalo 2 da Aproximação da Logística Sigmóide - Valores possíveis

Entradas			Saídas				
x_{22}	x_{21}	x_{20}	y_{20}	y_{19}	y_{18}	y_{17}	y_{16}
0	1	0	1	1	1	0	1
0	1	1	1	1	1	1	0
1	0	0	1	1	1	1	1

Pela tabela verdade tem-se que o bit 16 da saída é igual ao inverso do bit 20 da entrada e a saída 17 é obtida como a função lógica $\bar{x_{21}} + x_{20}$. Os bits 18, 19 e 20 são sempre iguais a 1. Assim, o valor final da saída é:

$$000,\ 111(\bar{x_{21}} + x_{20})\bar{x_{20}}$$

Intervalo 4: 1*;* $x \geq 4,5$

Este é o intervalo mais simples pois seu valor é fixo. Assim, a saída para este intervalo é igual a:

$$001,00000$$

A aproximação geral para todos os resultados é resumida na tabela a seguir:

Tabela 28 – Transformação direta para a função Sigmóide Logística

Intervalo / Bit	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
$0 \leq x < 1$	0	0	0	1	0	x_{19}	x_{18}	x_{17}
$1 \leq x < 2,5$	0	0	0	1	1	$\bar{x_{20}}$	x_{19}	x_{18}
$2,5 \leq x < 4,5$	0	0	0	1	1	1	$\bar{x_{21}} + x_{20}$	$\bar{x_{20}}$
$x \geq 4,5$	0	0	1	0	0	0	0	0

APÊNDICE B – Aproximação da Função Tangente Hiperbólica

A equação para a Tangente Hiperbólica é mostrada a seguir.

$$y = \begin{cases} x, & 0 \leq x < 0,5 \\ 0,5x + 0,25, & 0,5 \leq x < 1 \\ 0,25x + 0,5, & 1 \leq x < 2 \\ 1, & x \geq 2 \end{cases} \tag{B.1}$$

Se as entradas forem negativas, o resultado é obtido através da relação $y(-v) = -y(v)$.

Cada equação que define um intervalo é implementada separadamente. Esta implementação é feita através de blocos lógicos por meio de uma transformação direta, ou seja, os resultados possíveis são analisados e colocados diretamente na saída.

Para todos os intervalos o sinal de entrada de 32 bits é definido como:

$$x_{31}x_{30}x_{29}x_{28}x_{27}x_26x_{25}x_{24}x_{23}x_{22}x_{21}x_{20},\ x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2x_1x_0$$

Cada intervalo será analisado separadamente a seguir.

Intervalo 1: x*;* $0 \leq x < 0,5$

Este intervalo é simples de ser implementado, já que a entrada é colocada diretamente na saída. Como a saída tem somente 8 bits, ela pode ser representada como:

$$x_{22}x_{21}x_{20},\ x_{19}x_{18}x_{17}x_{16}x_{15}$$

Intervalo 2: $0,5x + 0,25$*;* $0,5 \leq x < 1$

Este intervalo tem como limites os valores $000000000000, 10000000000000000000$ e $000000000000, 11111111111111111111$. Pode-se notar que os bits 20 a 31 são sempre iguais a 0 e o bit 19 é sempre igual a 1. A entrada é então representada por:

$$000000000000,\ 1x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2x_1x_0$$

A primeira operação deste intervalo é a multiplicação da entrada por 0,5, que pode ser substituída por um deslocamento para a direita. O resultado é:

$$000000000000,\ 01x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2x_1$$

Em seguida deve-se somar 0,25 ao resultado do deslocamento. Este valor é representado por $000000000000, 0100000000000000000$, ou seja, a soma só irá afetar os dois bits após a vírgula do resultado do deslocamento. Esses dois bits têm valor fixo e igual a 01. Somar 01 a este valor dá um resultado de 10. Assim, a saída é representada por:

$$000000000000,\ 10x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2x_1$$

Como a saída é representada por somente 8 bits, o resultado sinal fica:

$$000,\ 10x_{18}x_{17}x_{16}$$

Intervalo 3: $0,25x + 0,5$*;* $1 \leq x < 2$

Neste intervalo, as entradas são representadas entre $000000000001, 00000000000000000000$ e $000000000001, 11111111111111111111$, o que significa que os bits 21 a 31 são sempre 0 e o bit 20 é sempre 1, como mostrado a seguir:

$$000000000001,\ x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2x_1x_0$$

De início deve-se multiplicar a entrada por 0,25, ou seja, deslocar a entrada duas vezes para a direita, para obter o sinal a seguir:

$$000000000000,\ 01x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2$$

Em seguida deve-se somar 0,5 ao resultado do deslocamento. Esta soma só afeta o bit imediatamente após a vírgula, o transformando de 0 para 1. Assim, a saída fica

$$000000000000,\ 11x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}x_{13}x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2$$

Truncando a saída para 8 bits tem-se:

$$000,\ 11x_{19}x_{18}x_{17}$$

Intervalo 4: 1*;* $x \geq 25$

Este intervalo tem como saída um valor fixo, mostrado a seguir:

$$001, 00000$$

A aproximação geral para todos os resultados é resumida na tabela a seguir:

Tabela 29 – Transformação direta para a função Tangente Hiperbólica

Intervalo / Bit	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
$0 \leq x < 0,5$	x_{22}	x_{21}	x_{20}	x_{19}	x_{18}	x_{17}	x_{16}	x_{15}
$0,5 \leq x < 1$	0	0	0	1	0	x_{18}	x_{17}	x_{16}
$1 \leq x < 2$	0	0	0	1	1	x_{19}	x_{18}	x_{17}
$x \geq 2$	0	0	1	0	0	0	0	0

Anexos

Implementation of a Reconfigurable Neural Network in FPGA

Janaina G. M. Oliveira, Robson Luiz Moreno, Odilon de Oliveira Dutra , Tales C. Pimenta
Group of Microelectronics - University of Itajuba - UNIFEI - Itajuba, Brazil
Email: see http://www.microeletronica.unifei.edu.br

***Abstract*—This article proposes a new hardware implementation for a Reconfigurable Neural Network for systems in which the topology needs flexibility. The used architecture is a Multi-Layer Perceptron, where the entry of a layer depends on the output of the previous layer. The approach allows flexibility in the number of network inputs, neurons, layers and in the activation function executed by neurons. Despite having been developed for FPGAs, the implemented circuit allows its implementation in ASIC, since it does not use proprietary internal blocks of the FPGA. By submitting the network to approximation tests, its operation and flexibility has been checked and validated.**

***Index Terms*—Artificial Neural Network, Reconfiguration, FPGA, Verilog, Flexibility.**

I. Introduction

Artificial Neural Networks (ANNs) are systems based on the behavior of human brain and its ability to adapt, learn and generalize. ANNs are effective solutions in systems where not all parameters are known or systems that do not have their operation fully understood, such as pattern recognition, approximations, medical diagnosis, control systems and projections. Those characteristics make them suitable for an extended range of applications, whether implemented in software or in hardware [1].

An ANN implemented in hardware allows the use of the intrinsic parallelism of the network, obtaining faster and more efficient system in terms of area [2]. On the other hand, the hardware development is slower and more complex than the software development [2]. One way to circumvent this issue and allow faster and easier prototyping is through the use of FPGA, proved to be efficient in neural network applications by decreasing its development time [3].

There are some cases in which patterns vary in such way the ANN performance is decreased. In such situations it is better to retrain and update the ANN structure and parameters in order to keep its performance. However, it is hard to make such changes in hardware implemented ANNs [4]. Using reconfigurable ANNs overcomes this issue as it joins software flexibility with hardware parallel capabilities [5].

In this matter, in Section II we present some reconfigurable ANN important characteristics. In Section III we present our ANN architecture which implements a Multi-Layer Perceptron (MLP) that can be reconfigured to modify its own structure, containing multipliers and activation functions with advantageous designs that makes use of a FPGA, without being dependent on FPGA intellectual property (IP). This approach makes possible its implementation in ASICs (Application Specific Integrated Circuits). In Section IV we show the obtained results with comparisons with other works and finally, in Section V we present our conclusions.

978-1-5386-1962-9/17/$31.00

II. Reconfigurable Neural Networks Architectures

Applications in robotics, space missions, mobile communication, medical systems [3], and others devices that are difficult to access and might be susceptible to changes in the environment [4] obtain advantages if using reconfigurable neural networks. They add flexibility to the network implemented in hardware and thus allow changes in its structure at run time.

Various features of an ANN may vary in a reconfigurable approach, such as the number of layers and neurons, the activation function, among others [4], [5]. In these last two mentioned approaches, the ANN is implemented with only one neuron layer to be utilized as many times as necessary. A unity control provides the reconfiguration of this single layer each time it represents a different level inside the ANN. This is the type of architecture implemented in this work.

More details of the adopted implementation will be given in the next section.

III. Proposed Architecture

The proposed ANN allows the selection of the number of inputs, layers and neurons per layer, the presence or absence of bias and the activation function implemented by each layer (bipolar step, linear, hyperbolic tangent and sigmoid logistic). These choices can be made at run time without the need of a new synthesis, keeping the network size fixed, but reconfigurable.

The neural network was implemented with 20 neurons, forming a real layer that will be reused during the processing (forming virtual layers). Because the neural network proposed is reconfigurable, several arrangements can be obtained in the same hardware. The smallest possible topology in this proposal is formed by only one neuron with one input. The biggest neural network is formed by 20 inputs, 3 hidden layers with 20 neurons each and an output layer also with 20 neurons.

A large variety of problems can be solved with inputs and outputs defined with 8 bits in fixed point [1]. Therefore, the neural network inputs and outputs are defined with 8 bits, where 3 are integer part and 5 are decimal part. The weights

and biases are set to 16 bits, 6 to integer part and 10 to decimal part. All the signals are defined with two's complement.

The ANN architecture developed in this work is depicted in Figure 1. It is divided in four major blocks. The module "Instructions Unit" is responsible for receiving the instructions used in the network configuration. Its data is sent to the "Main Controller", responsible for managing the entire execution of the ANN. The "Memory Unit" block contains memories with the weights and bias for each neuron. Finally, the "Layer" is composed by 20 neurons responsible for the network execution. Also, in the architecture, each block has a clock signal synchronizing its operation. This signal is not shown in Figure 1.

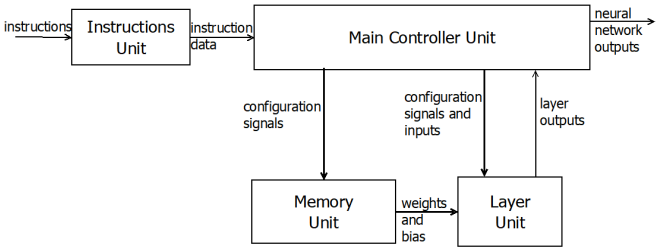

Fig. 1: General Structure

The operation of each block is detailed in the following subsections.

A. Instructions Unit

This module is responsible for receiving, extracting and organizing the necessary information that defines the network structure to be executed. All the instructions that are used to configure the neural network come from the user.

This is the initial step of the ANN processing, and it is performed whenever a new arrangement is required. The data containing the configuration information is sent to the Main Controller to ensure correct operation.

B. Memory Unit

The Memory Unit is formed by 21 memories blocks and a Search Controller block. Each memory stores the weights of one neuron and the last memory stores the biases to the entire network. All the memories are accessed in parallel manner every time a layer is being executed. For this implementation, the memories available in FPGA are used. If a future ASIC implementation is going to be developed, these blocks could be quickly implemented.

C. Layer Unit

The Layer is responsible for processing the arithmetic of the ANN. It consists of 20 neurons operating in parallel. This block is reused during ANN execution, bringing area economy and flexibility to the implementation. As soon as the layer finishes its execution, the results are sent to Main Controller and the layer gets configured to process another group of inputs.

The neurons that compose the layer are formed by a Multiplier-Accumulator (MAC) and 4 Activation Functions. They can be seen in Figure 2.

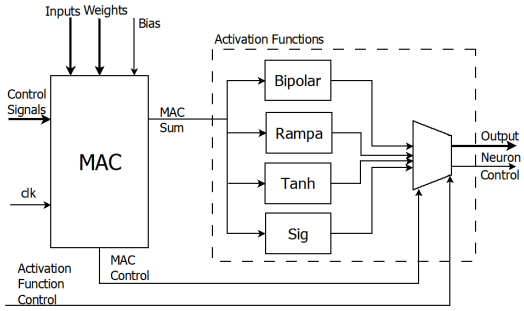

Fig. 2: Neuron

The signal "Inputs" in Figure 2 stands to 20 inputs with 8 bits each, and the signal "Weights" stands to 20 weights with 16 bits each. The signal "Bias" also has 16 bits. The MAC contains two multipliers that will be used in sequence for the 20 inputs of the neuron.

It was decided not to use the multiplier available in FPGA but another implementation, turning the topology into a Vendor IP free ANN and allowing VLSI implementations. This modification can increase the area spent on the implementation; however, as the multipliers are reused along the ANN processing, only 2 multipliers were implemented, outlining this issue.

This work adopts the VEDIC multiplier [6] as it is more promising if compared with others approaches [7]. For instance it presents lower power consumption than Booth multiplier eventhough their implementation areas and processing delays are almost equivalent [7]. This power consumption economy is important as the ANN is going to be part of a bigger VLSI implementation, e.g. portable or even implantable healthcare chip.

The VEDIC multiplier is part of the MAC block, responsible to multiply the inputs and weights and sum the products. Once the MAC operation is ready, the result (MAC sum) is sent to Activation Functions blocks as shown in Figure 2. The four possible functions, bipolar step, linear, hyperbolic tangent and logistic sigmoid, are executed in parallel; the selection of which one will define the neuron output is done by a Multiplexer controlled by the Main Controller (Figure 2).

The hyperbolic tangent and logistic sigmoid approximations are based on direct transformation of input to output [8], [9]. This implementation explores the fact that the hyperbolic tangent and the logistic sigmoid are symmetric functions and takes only the positive part of these functions. The function is divided in 4 intervals, each one with a linear approximation.

D. Main Controller

The Main Controller ensures the functioning of the entire neural network. This module is composed by a FSM responsible for managing which layer is being performed, the number

of neurons in this layer, which activation function must be processed, among other actions.

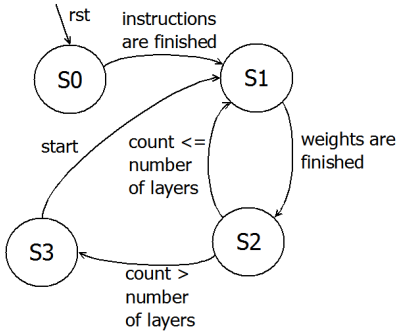

Fig. 3: Controller FSM

This block controls all the layers that must be executed. It ensures the correct inputs are applied to each layer and its correct configuration. This module is also responsible to connect the output of one layer to the input to the next layer, ensuring the full processing of the neural network.

All neural network execution control is done by hardware, without needing any other software system to assist its execution. Also, the entire ANN implementation is independent of the FPGA model enabling its application in a ASIC circuit in the future.

IV. Results and Simulations

The proposed reconfigurable ANN with 20 neurons was implemented in FPGA Cyclone IV EP4CE115F29C7N using Verilog. The summary of its area consumption and maximum frequency compared to three another implementations is shown in Table I. The first and the second implementations [10] are neural networks that do not allow reconfiguration, that means, the network topology is fixed. The third implementation allows the reconfiguration of the network [5], in which the neurons implemented are multiplexed, allowing multiple neural networks topologies in the same hardware, as in this work.

By observing Table I, it is also possible to notice that this works implementation has the largest number of implemented neurons, which leads, in theory, to greater area consumption. However, the amount of registers used is less than the amount used by application ([10], Iris), even that this approach has only 14 neurons implemented. The amount of memory used is also the lowest among the architectures that shows the Total of Memory Bits used.

Since the proposed ANN does not use any FPGA embedded multiplier, it was expected that the maximum operational frequency was lower than that observed in others approaches. This fact was indeed observed; however, the loss in speed is compensated by the ability and ease of build the neural network as an ASIC as it is independent of proprietary IPs.

In order to evaluate the correct operation and the time spend in execution of the neural network, two problems proposed in literature were simulated, and the results of the simulations were compared. The problems were solved adjusting the here implemented neural network to meet the arrangement in each compared problem. The area of the neural network implemented did not vary, since it allows reconfiguration in execution time. Therefore, the same hardware was used to test the two problems proposed: the Iris problem [10] and the approximation of a function [10]. All the problems were trained in MATLAB.

In the Iris problem, 4 measurements of the Iris flowers are used as inputs of the neural network. These characteristics separates the Iris flowers in 3 classes [10]. The arrangement defined by [10] to solve this problem has 4 inputs, two hidden layers with 8 neurons and 3 neurons, respectively implementing the tangent hyperbolic as activation function and the output layer with 3 neurons and linear activation function.

The time spent to solve Iris problem, as well as the results obtained are shown in Table II. As expected, the time spent for the Iris problem simulation was greater in the approach proposed here. This happens because it was chosen not to use the embedded multiplier available in FPGA; and, although the use of a different multiplier affects the run-time, it also makes the circuit possible to be implemented in ASICs. The neural network developed in this work obtained 99.3% of correct classification.

The second tested problem was the approximation of the function $y = sin(0,003x)/e^{(0,03x)}$ [10]. The neural network used in this approximation has 1 input, 1 hidden layer with 5 neurons and tangent hyperbolic activation function and the output layer with 1 neuron implementing a linear activation function. The hardware used for this simulation is the same used in the previous problem. New instructions were sent to the ANN and a new topology was tested, new weights and bias were also required. The summary of time spent and the results compared with another implementation are shown in Table III.

Table III also shows that the error achieved by this approach was higher than the error of [10]. One reason for this is the fact that the implementation in this work uses a fixed-point representation with 8 bits, while the comparative approach uses floating point in its calculation. The advantage of using fixed-point comes from the fact of getting simpler and smaller circuits. The amount of bits required has already been discussed in the literature, and 8 bits are said to be sufficient for a wide range of implementations [1].

The arrangement used to solve the problem is another reason for the increased error values, then a new ANN arrangement with 1 input, 8 neurons in hidden layer implementing the hyperbolic tangent function and 1 output layer with 1 neuron with linear activation function was utilized. The results are shown in Table IV. It can be noted that the errors achieved in the new simulation decreases significantly, while the time spent in the simulation had a small increase. The new arrangement has 4 more neurons than the original but the area is the same, since the implemented topology is reconfigurable, allowing multiple neural networks arrangements without in execution time.

The two problems solved have different arrangements and

TABLE I: Resource Requirements and Comparison

Circuit Characteristics	This approach	[10], Iris	[10], Approximation Function	[5]
Device Selected	EP4CE115F29C7N	EP3SL50F484C2	EP3SL50F484C2	Xilinx
Number of Neurons	20	14	6	4
Total Logic Elements	42,499 (37%)	–	–	1,888
Total Registers	8,582 (7%)	9,791 (26%)	5,294 (14%)	–
Total Memory Bits	40,960 (1%)	181,901 (3%)	116,405 (2%)	7 (BRAMs)
Embedded 9-bit Multiplier	0 (0%)	–	–	4
DSPs	–	12 (3%)	8 (2%)	–
LUTs	–	8,782 (23%)	4,591 (12%)	–
Pins	194 (37 %)	–	–	–
Fmax (MHz)	77.59	300	300	110

TABLE II: Evaluation of the Iris Problem

	This work	[10] - Iris Problem
Arrangement	4 - 8 - 3 - 3	
Maximum absolute error	0.0766	0.0021
Error Sum of Squares	0.2309	2.4E-5
Number of clock cycles	90	37
% of Correct Classification	99.3%	–
Time	1.16 us	0.124 us

TABLE III: Evaluation of the Function Approximation Problem

	This work	[10] - Approximation Problem
Arrangement	1 - 5 - 1	
Maximum absolute error	0.1927	0.0051
Mean absolute error	0.0403	1.5E-4
Error Sum of Squares	1.54	4.9E-5
Number of clock cycles	53	12
Time	683 ns	40.7 ns

were simulated with the same hardware, so there is no change in the size of the network implemented in FPGA. This feature gives more flexibility to the neural network, making it similar to the networks implemented in software and still has the advantages of a parallel processing that only the hardware can achieve. The increase in the time spent observed is because the implementation uses a multiplier different of the available in FPGA. In the matter of the number of clock cycles used to solve each problem varies because it depends on the amount of layers and the amount of neurons present in each layer. The advantages of this work topology in terms of area are clearest when larger neural networks are necessary.

V. Conclusion

Using ANNs in hardware with reconfigurable architecture increases flexibility. This is advantageous for systems concerned in variable environment pattern recognition as robotics, space missions, mobile communication, medical systems. Following this concept, this article has proposed an implementation of reconfigurable neural network that is independent of the FPGA model, meaning that the implementation does not use FPGA proprietary circuit. This characteristic allows the ANN to be implemented in ASIC circuits, allowing implementation of dedicated chips.

The network was submitted to two tests in order to check its performance and compare with other approaches proposed in literature. Although the errors reached were a bit worse than the comparison, the networks flexibility allows changing the topology used without changing the hardware, allowing the improvement of the tests results.

Further work must be done in order to implement the neural network in ASIC circuits in order to be used in specific applications as medical systems.

TABLE IV: Evaluation of the Function Approximation Problem - Improved Arrangement

	This work - Version 1	This work - Version 2
Arrangement	1 - 5 - 1	1 - 8 - 1
Maximum absolute error	0.1927	0.0804
Mean absolute error	0.0403	0.0039
Error Sum of Squares	1.54	0.21
Clock Cycles	53	64
Time	683 ns	824 ns

Acknowledgment

This work was supported by CAPES, CNPq and FAPEMIG.

References

[1] Y. Liao, "Neural networks in hardware: A survey," *Department of Computer Science, University of California*, 2001.

[2] F. M. Dias, A. Antunesa, and A. M. Motab, "Artificial neural networks: A review of commercial hardware," *Eng. Appl. Artif. Intell.*, vol. 17, no. 8, pp. 945–952, 2004.

[3] J. E. R. Finker, I. del Campo and F. Doctor, "Multilevel adaptive neural network architecture for implementing single-chip intelligent agents on FPGAs," *International Joint Conference on Neural Networks*, 2013.

[4] e. a. C.A. A.Silva, A. D. D. Neto, "Definition of an architecture to configure artificial neural networks topologies using partial reconfiguration in FPGA," *IEEE Latin America Transactions*, pp. 2094–2100, 2015.

[5] A. Youssef, K. Mohammed, and N. A., "Two novel generic, reconfigurable neural network FPGA architectures," *Artificial Intelligence with Applications in Engineering and Technology (ICAIET)*, 2014.

[6] B. S. S. S. B. B.S. Premananda, Samarth S. Pai, "Design and implementation of 8-bit vedic multiplier," *International Journal of Advanced Research in Electrical Electronics and Instrumentation Engineering*, vol. 2, no. 12, pp. 5877–5882, 2013.

[7] B. Sharma and A. Bakshi., "Comparison of 24x24 bit multipliers for various performance parameters." *International Conference on Advent Trends in Engineering, Science and Technology (ICATEST)*, 2015.

[8] B. R. H.-G. H. Amin, K. M. Curtis, "Piecewise linear approximation applied to nonlinear function of a neural network," *IEEE Proceedings Circuits Devices and Systems*, vol. 144, no. 6, pp. 313–317, 1997.

[9] M. Tommiska, "Efficient digital implementation of the sigmoid function for reprogrammable logic," *IEEE Proc.-Comput. Digital Tech*, vol. 150, no. 6, pp. 403–411, 2003.

[10] E. N. D. S. B. A. P. Do A. Ferreira, "A high performance full pipelined arquitecture of MLP neural networks in FPGA," *Proc. IEEE 17th Int. Conf. Electron. Circuits Syst.*, pp. 742–745, 2010.