UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Gabriel de Jesus Coelho da Silva

IMPLEMENTAÇÃO DE REDES NEURAIS ARTIFICIAIS EM HARDWARE PARA INFERÊNCIA

Santa Maria, RS
2019

Gabriel de Jesus Coelho da Silva

IMPLEMENTAÇÃO DE REDES NEURAIS ARTIFICIAIS EM HARDWARE PARA INFERÊNCIA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Engenharia de Computação**. Defesa realizada por videoconferência.

ORIENTADOR: Prof. José Eduardo Baggio

Santa Maria, RS
2019

Gabriel de Jesus Coelho da Silva

IMPLEMENTAÇÃO DE REDES NEURAIS ARTIFICIAIS EM HARDWARE PARA INFERÊNCIA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Engenharia de Computação**.

Aprovado em 15 de julho de 2019:

José Eduardo Baggio, Dr. (UFSM)
(Presidente/Orientador)

Leonardo Londero de Oliveira, Dr. (UFSM)

Guilherme Henrique dos Santos, Bel. (UFSC) (videoconferência)

Santa Maria, RS
2019

DEDICATÓRIA

Aos meus pais, Dilso e Rosângela.

AGRADECIMENTOS

Aos meus pais, Dilso e Rosângela, pelo amor incondicional e imensurável, que não medem esforços para me ajudar e me ver feliz. Sem vocês nada disso (nem eu) existiria. Meu sucesso só se deve a vocês e é, portanto, dedicado sempre a vocês.

A todos os meus amigos que, felizmente, tornaram-se muitos durante a jornada. Aos que surgiram e aos que partiram.

Aos professores que compartilharam sua sabedoria durante a graduação.

“Au milieu de l’hiver, j’apprenais enfin qu’il y avait en moi un été invincible.”

(Albert Camus)

RESUMO

IMPLEMENTAÇÃO DE REDES NEURAIS ARTIFICIAIS EM HARDWARE PARA INFERÊNCIA

AUTOR: Gabriel de Jesus Coelho da Silva
ORIENTADOR: José Eduardo Baggio

O crescente interesse na utilização de redes neurais artificiais em serviços para usuários finais, que exigem baixa latência e alta responsividade, tornam desejável o uso de aceleradores em hardware dedicados para inferência. Dispositivos programáveis do tipo FPGA (*Field-Programmable Gate Arrays*) apresentam flexibilidade ideal para aceleração de redes neurais com capacidade de suportar diferentes modelos de arquitetura de rede, mantendo a performance desejada. Um modelo de redes neurais artificiais modular é desenvolvido em linguagem de descrição de hardware a fim de permitir inferência em dispositivos reconfiguráveis de forma performática desejável. O desenvolvimento modular permite fácil extensibilidade de forma a suportar novas arquiteturas de redes neurais e diferentes tipos de funções de ativação. A validação do projeto é efetuada através da implementação em hardware de uma rede neural simples e amplamente conhecida (função *OU-exclusivo* (XOR)).

Palavras-chave: Redes Neurais Artificiais. Hardware. VHDL. FPGA. Inferência.

ABSTRACT

A HARDWARE IMPLEMENTATION OF ARTIFICIAL NEURAL NETWORKS FOR INFERENCE

AUTHOR: Gabriel de Jesus Coelho da Silva
ADVISOR: José Eduardo Baggio

The growing investment in the use of artificial neural networks for end-user services, which require low latency and high responsiveness, make it desirable to have dedicated hardware accelerators for inference. FPGA (*Field-Programmable Gate Arrays*) programmable devices have the required ideal flexibility for the deployment of artificial neural network accelerators, while being able to support different architectural network models and still keeping performance. A modular artificial neural network design is developed in hardware description language in order to allow inference from reconfigurable devices with desirable performance. The modular design enables it to be easily scaled to support new neural network architectures and different activation functions. The project's validation is verified by a hardware implementation of a simple and widely known neural network (*exclusive-OR* (XOR) function).

Keywords: Neural Networks. Hardware. VHDL. FPGA. Inference.

LISTA DE FIGURAS

Figura 2.1 – Diagrama da relação de FPGAs entre os diversos dispositivos semicondutores. 22
Figura 2.2 – Disposição de blocos lógicos em FPGA. 23
Figura 2.3 – Exemplo de LUT de 3 entradas. 24
Figura 2.4 – LUT para circuito de maioria de votos. 24
Figura 2.5 – Diagrama de blocos da TPU. 27
Figura 2.6 – Acelerador de CNNs na Catapult. 28
Figura 2.7 – Diagrama da NPU do Project Brainwave. 30
Figura 2.8 – Forma de onda obtida na simulação. Observa-se que a saída é a esperada para a função XOR. 36
Figura 2.9 – Forma de onda obtida na simulação. Observa-se que a saída é a esperada para a função XOR. 37
Figura 2.10 – Esquemático de alto nível (*top-level design*) para a rede neural artificial. 38
Figura 2.11 – Esquemático dos neurônios artificiais arranjados na topologia da rede neural. 39
Figura B.1 – Forma de onda completa para o circuito da rede neural para a função XOR. 54
Figura C.1 – Esquemático do neurônio artificial. 55

LISTA DE GRÁFICOS

Gráfico 2.1 – Função de Heaviside. 19
Gráfico 2.2 – Função sigmoide. 20
Gráfico 2.3 – Função tangente hiperbólica. 20
Gráfico 2.4 – Função ReLU. 21

LISTA DE ILUSTRAÇÕES

Ilustração 2.1 – Diagrama do neurônio artificial de McCulloch e Pitts (1943). 18
Ilustração 2.2 – Diagrama de uma rede neural *perceptron* multicamadas. 19
Ilustração 2.3 – Diagrama da rede neural em VHDL. 32
Ilustração 2.4 – Disposição dos neurônios nas camadas ocultas. 32
Ilustração 2.5 – Máquina de estados responsável pelo controle do neurônio (`neuron.vhd`). 33
Ilustração 2.6 – Arquitetura da rede neural artificial para resolução do problema XOR. 35

LISTA DE TABELAS

Tabela 2.1 – Tabela-verdade para circuito de maioria de votos. 23
Tabela 2.2 – Tabela-verdade para função XOR de duas entradas. 35
Tabela 2.3 – Resumo de utilização do dispositivo FPGA para implementação da rede neural da função XOR. 37

LISTA DE ABREVIATURAS E SIGLAS

ANN	Rede neural artificial (*artificial neural network*)
$FPGA$	*Field-Programmable Gate Array*
$ASIC$	Circuito integrado de aplicação específica*Application-Specific Integrated Circuit*
CPU	Unidade central de processamento (*Central Processing Unit*)
GPU	Unidade de processamento gráfico (*Graphics Processing Unit*)
LB	Elemento de lógica (*Logic Block*)
IOB	Elemento de entrada/saída (*Input/Output Block*)
SB	Elemento de conexão (*Switch Block*)
CB	Elemento de conexão (*Connection Block*)
IP	Propriedade intelectual (*Intelectual Property*)
CAD	Desenho assistido por computador (*Computer-Aided Design*)
LUT	Tabela de *look-up* (*Look-Up Table*)
FF	*Flip-Flop*
$SRAM$	*Static Random Access Memory*
MUX	Multiplexador (*Multiplexer*)
TPU	*Tensor Processing Unit*
$PCIe$	*Peripheral Component Interconnect Express*
MMU	Unidade de multiplicação de matrizes (*Matrix Multiply Unit*)
MAC	Multiplicador e acumulador
$DRAM$	*Dynamic Random Access Memory*
$ReLU$	Unidade retificadora linear (*Rectified Linear Unit*)
I/O	Entrada/saída (*Input/Output*)
HDL	Linguagem de descrição de hardware (*Hardware Description Language*)
CNN	Redes neurais convolucionais (*Convolutional Neural Networks*)
PE	Bloco de processamento (*Processing Element*)
NPU	Unidade de processamento neural (*Neural Processing Unit*)
$VHDL$	*VHSIC Hardware Description Language*
XOR	OU-exclusivo (*exclusive-OR*)

LISTA DE SÍMBOLOS

x_m	m-ésima entrada do neurônio artificial
w_{k_m}	m-ésimo peso associado ao k-ésimo neurônio artificial
b_k	*Bias* associado ao k-ésimo neurônio artificial
$\varphi(\cdot)$	Função de ativação
y_k	Saída do k-ésimo neurônio artificial
$\sigma(x)$	Função sigmoide
$tanh(x)$	Função tangente hiperbólica
$ReLU(x)$	Função unidade retificadora linear
h_n	n-ésima camada oculta
n_k	k-ésimo neurônio

SUMÁRIO

1 INTRODUÇÃO **15**
1.1 OBJETIVOS 16
2 DESENVOLVIMENTO **17**
2.1 FUNDAMENTAÇÃO TEÓRICA 17
2.1.1 Redes neurais artificiais **17**
2.1.2 FPGAs **22**
2.1.3 Estado-da-arte **25**
2.1.3.1 Google Tensor Processing Unit (TPU) *25*
2.1.3.2 Microsoft Catapult *27*
2.1.3.3 Microsoft Project Brainwave *29*
2.2 METODOLOGIA 30
2.2.1 Implementação em VHDL **31**
2.2.2 Validação da rede **35**
3 CONCLUSÃO **40**
3.1 TRABALHOS FUTUROS 40
REFERÊNCIAS BIBLIOGRÁFICAS **41**
ANEXO A – CÓDIGO DA REDE NEURAL ARTIFICIAL EM VHDL **44**
ANEXO B – FORMA DE ONDA COMPLETA **54**
ANEXO C – ESQUEMÁTICO DO NEURÔNIO ARTIFICIAL **55**

1 INTRODUÇÃO

O aprendizado de máquina tem se tornado cada vez mais eficaz em aplicações que utilizam dados para obter predições, tomar decisões e prover melhor entendimento de dados anteriormente ininteligíveis devido à seus imensos volumes (JORDAN; MITCHELL, 2015; QIU et al., 2016; SCHMIDHUBER, 2015).

Uma das subcategorias de aprendizado de máquina que apresentou avanços expressivos é a de aprendizado profundo (*deep learning*) com redes neurais artificiais (LECUN; BENGIO; HINTON, 2015; KRIZHEVSKY; SUTSKEVER; HINTON, 2012; SILVER et al., 2017; MNIH et al., 2013; HINTON et al., 2012; RADFORD et al., 2019). Sua efetividade pode ser atribuída à vasta quantidade e diversidade de dados agora disponíveis e sendo constantemente gerados (NETZER et al., 2011; NAJAFABADI et al., 2015; QIU et al., 2016) e a avanços nas tecnologias subjacentes de processamento computacional (DEAN et al., 2012; ABADI et al., 2016; JOUPPI et al., 2017; OVTCHAROV et al., 2015). Isto têm permitido a implementação de redes neurais em variados cenários reais, incluindo aplicações financeiras (LIN; HU; TSAI, 2012), sistemas de recomendação, direcionamento de publicidade, reconhecimento de voz e facial (AMODEI et al., 2016; HINTON et al., 2012) e aplicações sensíveis à latência como detecção de obstáculos e vias para veículos autônomos (BOJARSKI et al., 2016; MULLER et al., 2006; CHEN et al., 2015).

Redes neurais artificiais para aprendizado supervisionado (*supervised learning*) são sujeitas à duas fases distintas: **treinamento** (também referido como aprendizado), quando são apresentadas à rede vastas quantidades de atributos (*features*) de dados de entrada e suas classificações de saída correspondentes, ajustando assim os pesos da rede e, consequentemente, uma função que mapeia os dados de entrada para os dados de saída; e **inferência**, quando os pesos já estão ajustados ao modelo e predições podem ser feitas a partir de novos dados de entrada desconhecidos. Uma vez implementado, o sistema consegue continuar se aprimorando a partir da ingestão de novos dados obtidos de outras fontes (*e.g.* usuários do sistema constantemente gerando novos dados) (HAYKIN, 2009; GOODFELLOW; BENGIO; COURVILLE, 2016; BISHOP, 2006).

Atualmente, é comum implementar tais sistemas em cenários reais a partir da utilização de um dispositivo central baseado em nuvem para treinamento e inferência devido ao processamento intensivo envolvido. Esta implementação é realizada através do envio de atributos dos dados de entrada via rede de comunicação (*i.e.* internet), onde são processados em servidores em nuvem e posterior envio do resultado de saída novamente através da rede. Esta implementação é limitada à conectividade da rede e impacta diretamente na quantidade de atributos que podem ser utilizados (LIN; KOLCZ, 2012; RANJAN, 2014).

Outra implementação é a de treinar o sistema em nuvem e enviar os pesos ao

dispositivo. Entretanto, sistemas baseados em CPU e/ou GPU apresentam latência significativa e consumo de potência excessivo; e sistemas desenvolvidos com propósito específico (*i.e. ASIC; Application-Specific Integrated Circuit*) não apresentam a flexibilidade necessária e desejada para se aproveitar da capacidade de retroingestão de dados e auto-aprimoramento.

1.1 OBJETIVOS

Um sistema baseado em hardware através de FPGAs (*Field Programmable Gate-Array*) é proposto para implementação de redes neurais artificiais em dispositivos finais (*on-device*; *on the edge*) em razão de sua flexibilidade em receber novos modelos e pesos; e pela sua eficiência energética e baixa latência para inferência.

A validação é efetuada através da implementação em hardware de uma rede neural simples e amplamente conhecida (função *OU-exclusivo* (XOR)).

2 DESENVOLVIMENTO

2.1 FUNDAMENTAÇÃO TEÓRICA

São apresentados os conceitos necessários para o desenvolvimento e entendimento do trabalho além da análise de sua aplicabilidade para o mesmo.

2.1.1 Redes neurais artificiais

As redes neurais artificiais são as interconexões de unidades processuais relativamente simples denominadas **neurônios artificiais** seguindo um modelo de interconexões de camadas estruturais que compõe uma **arquitetura neural** (HAYKIN, 2009; BISHOP, 2006; GOODFELLOW; BENGIO; COURVILLE, 2016).

O mapeamento de dados de entrada para dados de saída é feito através de um processo de **aprendizado** (também denominado treinamento). O **aprendizado supervisionado** consiste na inserção de dados de entrada com seus respectivos dados de saída desejados de forma a ajustar os pesos sinápticos da rede neural a fim de minimizar o erro do mapeamento de entrada-saída (HAYKIN, 2009; BISHOP, 2006).

Também há a possibilidade de reajuste dos pesos sinápticos ou até modificação da própria topologia para que a rede se adapte à mudanças no ambiente em que está atuando ou torne-se mais eficiente com o decorrer do tempo. Devido à natureza distribuída das informações armazenadas na rede neural além de seu processamento altamente paralelo, há alta **tolerância a falhas** (HAYKIN, 2009).

A utilização de grandes quantidades de camadas intermediárias permite classificações de dados de entrada mais complexos e é chamada de *deep learning*, ou aprendizado profundo (GOODFELLOW; BENGIO; COURVILLE, 2016).

O neurônio artificial representa um modelo simplificado do neurônio biológico. O primeiro modelo foi desenvolvido por McCulloch e Pitts (1943) com a estrutura mostrada na Figura 2.1.

O neurônio k apresenta entradas $x_0, x_1, \ldots, x_m$, sem limitação para o número de entradas m, com pesos sinápticos $w_0, w_1, \ldots, w_{k_m}$ associados para cada entrada, de forma que, conforme a Equação (2.1):

$$v_k = \sum_{j=0}^{m} w_{kj} x_j \tag{2.1}$$

Comumente tem-se a entrada $x_0 = 1$ com seu peso associado denominado *bias*,

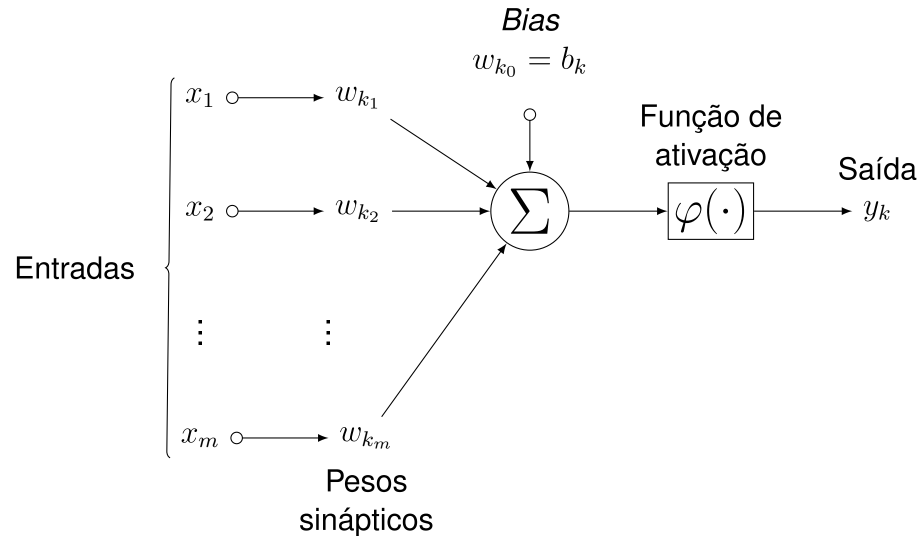

Ilustração 2.1 – Diagrama do neurônio artificial de McCulloch e Pitts (1943).

Fonte: Adaptado de Haykin (2009).

tal que $w_{k_0} = b_k$. Tem-se, então, da Equação (2.1):

$$v_k = u_k + b_k = \sum_{j=1}^{m}(w_{kj}x_j) + b_k \tag{2.2}$$

Uma função de ativação $\varphi(\cdot)$ é associada para o neurônio k definindo sua saída tal que:

$$y_k = \varphi(u_k + b_k) \tag{2.3}$$

O modelo de neurônio artificial de McCulloch e Pitts (1943) utiliza a função de Heaviside como função de ativação devido ao seu comportamento de limiar (*"threshold"*) observado nos neurônios biológicos. Desta forma, a saída y_k do neurônio é dada por:

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

A partir do neurônio artificial de McCulloch e Pitts (1943), a rede neural de uma camada denominada *perceptron* e seu respectivo teorema de aprendizado foi desenvolvida por Rosenblatt (1958) permitindo classificação de padrões linearmente separáveis, mas ainda sendo não-performático para classificações complexas.

O desenvolvimento de redes neurais com mais camadas internas (*i.e.* camadas entre a camada de entrada e a camada de saída) e a utilização de funções de ativação diferenciáveis não-lineares levou ao desenvolvimento dos *perceptrons* multi-camadas. O êxito destas redes neurais se dá majoritariamente devido ao mecanismo de aprendizado

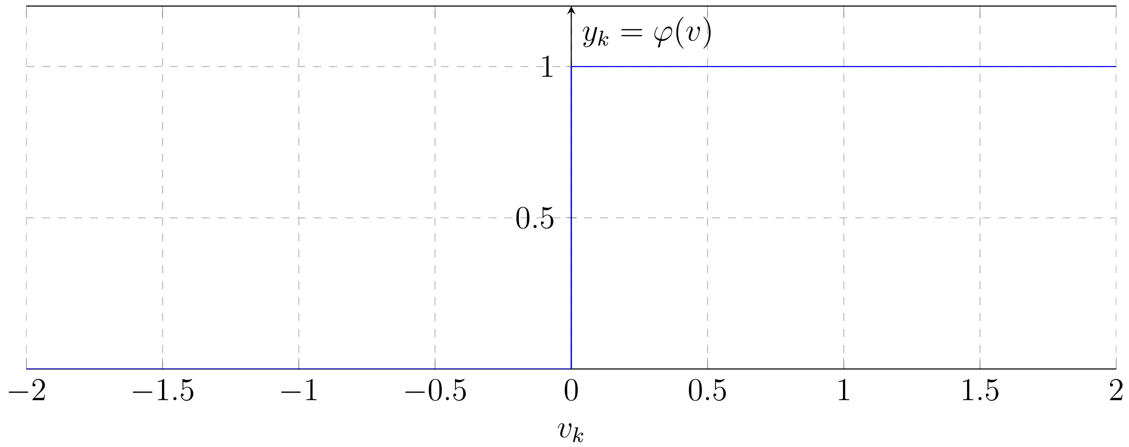

Gráfico 2.1 – Função de Heaviside.

Fonte: Adaptado de Haykin (2009).

"back-propagation" (RUMELHART; HINTON; WILLIAMS, 1986) que apresentou-se efetivo e computacionalmente eficiente. O diagrama de uma rede *perceptron* multi-camadas é exibido na Figura 2.2.

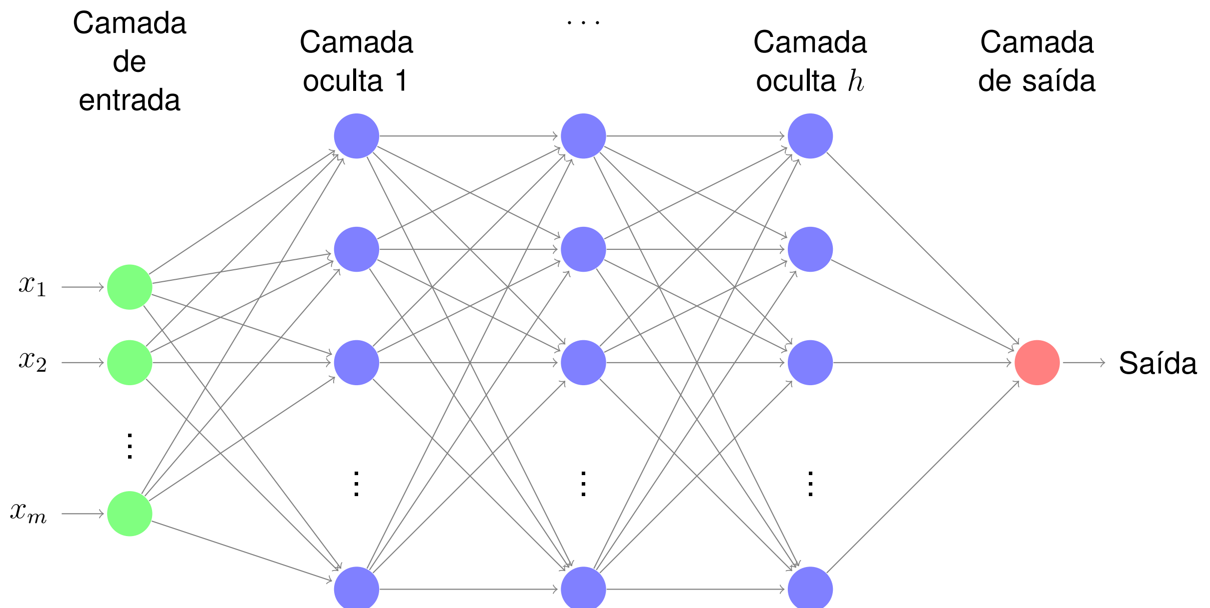

Ilustração 2.2 – Diagrama de uma rede neural *perceptron* multicamadas.

Fonte: Adaptado de Haykin (2009).

Funções de ativação comumente associadas ao *perceptron* multi-camadas são a sigmoide (2.4) e tangente hiperbólica (2.5) devido à suas características matemáticas desejáveis. Observa-se, no gráfico, o intervalo de saída de $[0, 1]$ para a sigmoide e $[-1, 1]$

para a tangente hiperbólica.

$$\varphi(\cdot) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

$$\varphi(\cdot) = \tanh x = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.5}$$

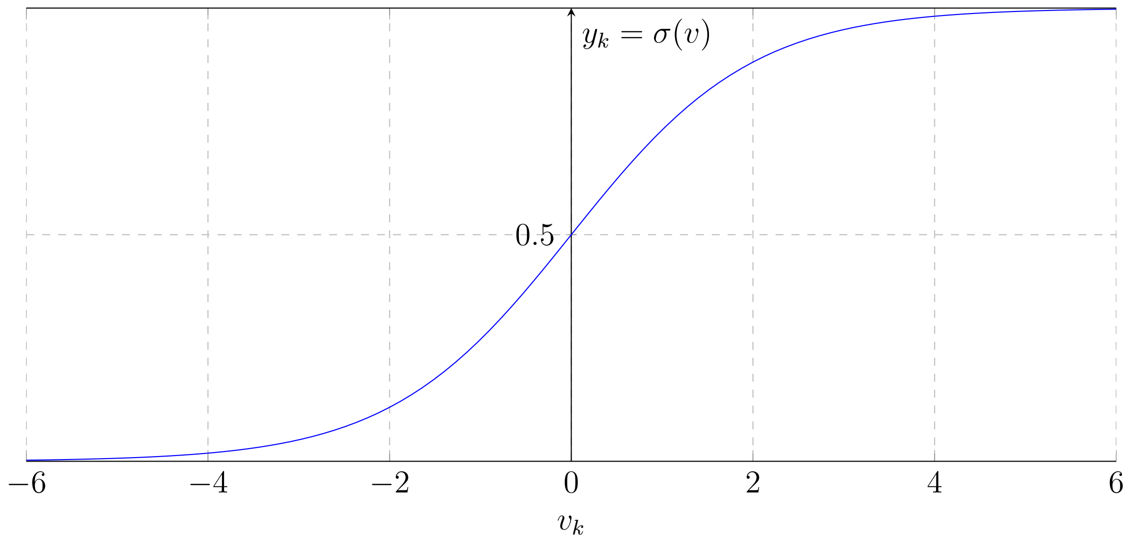

Gráfico 2.2 – Função sigmoide.

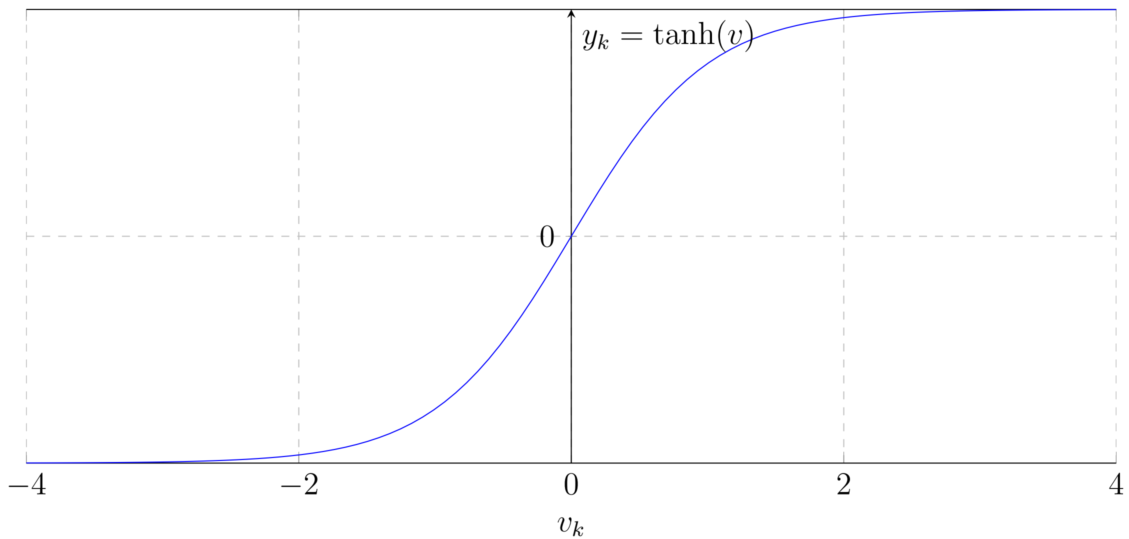

Gráfico 2.3 – Função tangente hiperbólica.

A função não-linear mais comumente utilizada para as camadas ocultas tem sido a unidade retificadora linear (*Rectified Linear Unit*, ReLU), que apresenta aprendizado extremamente rápido para redes de muitas camadas, além de possuir implementação simples

e fácil (GLOROT; BORDES; BENGIO, 2011). Sua Equação é exibida em 2.6.

$$\varphi(\cdot) = ReLU(x) = max(x, 0) \tag{2.6}$$

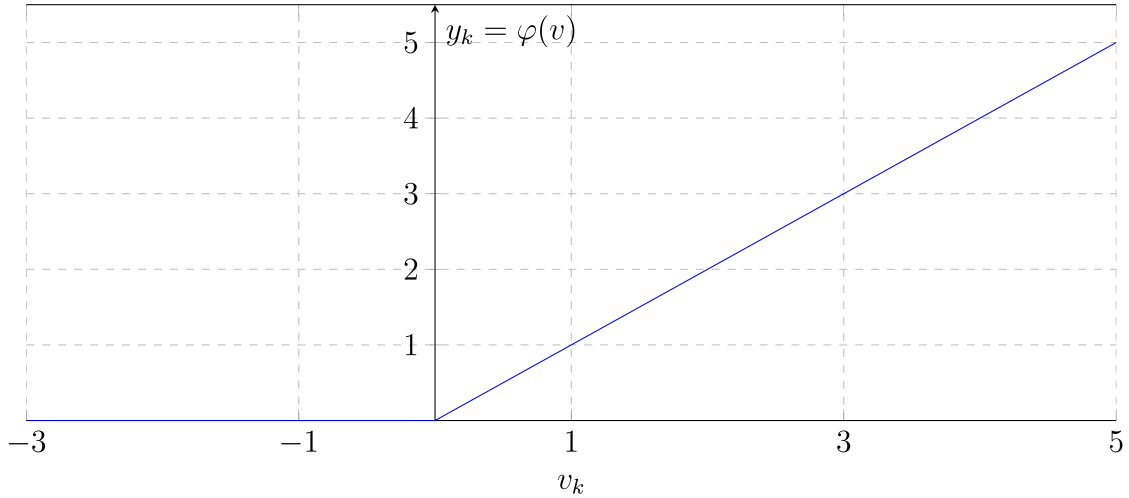

Gráfico 2.4 – Função ReLU.

Nas redes *perceptron* multi-camadas totalmente conectadas (*"fully-connected"*), as saídas de todos os neurônios das camadas internas são conectadas às entradas de todos os neurônios da camada seguinte. Não há limite para o número de neurônios nas camadas ou para o número de camadas. O fluxo de dados da rede progride unicamente da entrada para a saída, de camada em camada. As camadas internas são responsáveis por realizar transformações não-lineares nos dados de entrada relevantes e reduzi-los a um espaço de dados com classes de interesse mais facilmente separáveis (HAYKIN, 2009).

2.1.2 FPGAs

Em ASICs (*Application-Specific Integrated Circuit*, circuito integrado de aplicação específica), o circuito é desenvolvido e tem seu desenho final impresso em placas (*"wafers"*) permanentemente, fornecendo alto desempenho mas baixa flexibilidade (WESTE; HARRIS, 2010; KAESLIN, 2014). Unidades de processamento, tais como CPUs ou GPUs, são desenvolvidas da mesma forma mas permitem certa flexibilidade por receberem instruções para processamento de dados, porém perdendo desempenho devido à isso. As FPGAs (*Field-Programmable Gate Array*) são dispositivos de lógica programável, de forma a apresentar uma relação intermediária entre a flexibilidade das unidades de processamento e o desempenho dos circuitos de aplicação específica (KAESLIN, 2014). O diagrama na Figura 2.1 apresenta diversos dispositivos semicondutores e destaca como as FPGAs se relacionam com os mesmos.

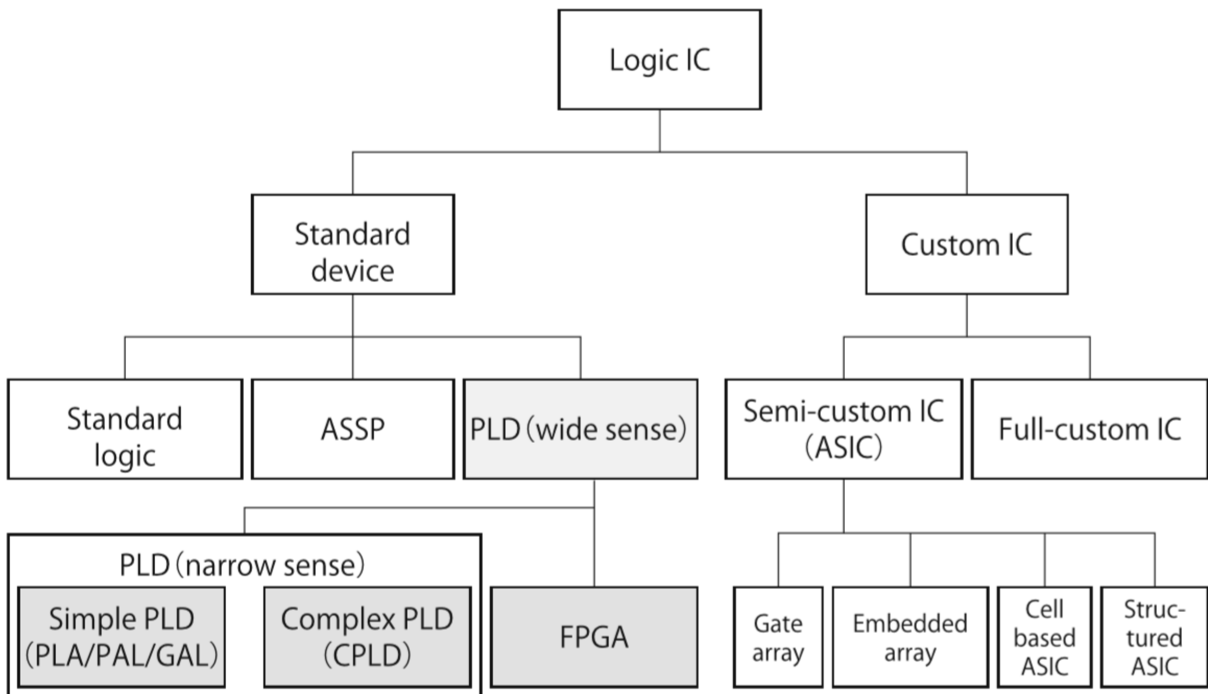

Figura 2.1 – Diagrama da relação de FPGAs entre os diversos dispositivos semicondutores.

Fonte: Amano (2018).

FPGAs são compostas por matrizes de elementos de lógica (LB, *logic block*) cercadas por elementos de entrada/saída (IOB, *input/output block*); todos conectados por elementos de conexão (SB, *switch block*; e CB, *connection block*), além de redes de *clock*, blocos de memórias e blocos somadores (ou outros blocos proprietários, *"IP blocks"*). O envio de dados de *design* à FPGA faz com que seus elementos sejam configurados de forma a estabelecer um circuito lógico previamente programado em ferramentas de CAD (*Computer-Aided Design*) utilizando-se linguagens de descrição de hardware (*i.e.* VHDL ou

Verilog) (AMANO, 2018). A Figura 2.2 apresenta a disposição dos blocos em uma FPGA.

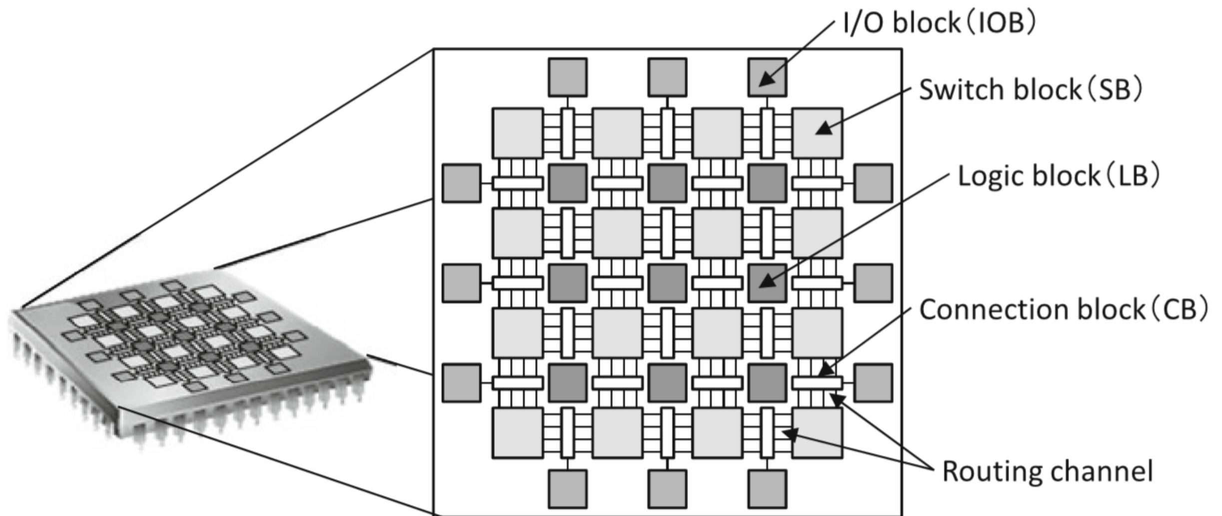

Figura 2.2 – Disposição de blocos lógicos em FPGA.

Fonte: Amano (2018).

FPGAs modernas possuem blocos lógicos compostos por tabelas de *look-up* (LUT, *look-up table*), *flip-flops* (FF) e um seletor. De acordo com um bit de configuração, o seletor é configurado de forma a fornecer como saída do bloco o conteúdo do *flip-flop* ou da LUT. LUTs de k entradas podem expressar quaisquer tabelas-verdade com k entradas e necessitam de 2^k bits na memória de configuração. Desta forma, há um *trade-off* com relação ao tamanho das LUTs e a eficiência de área/latência (*delay*) da FPGA (AMANO, 2018). Um exemplo de LUT contendo a implementação de um circuito de maioria de votos é exibido na Figura 2.4.

Tabela 2.1 – Tabela-verdade para circuito de maioria de votos.

A	B	C	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Fonte: Adaptado de Amano (2018).

Os blocos de conexão, responsáveis pelo roteamento e interconexão dos blocos lógicos e de entrada/saída, são compostos por *switches* programáveis através da lógica de

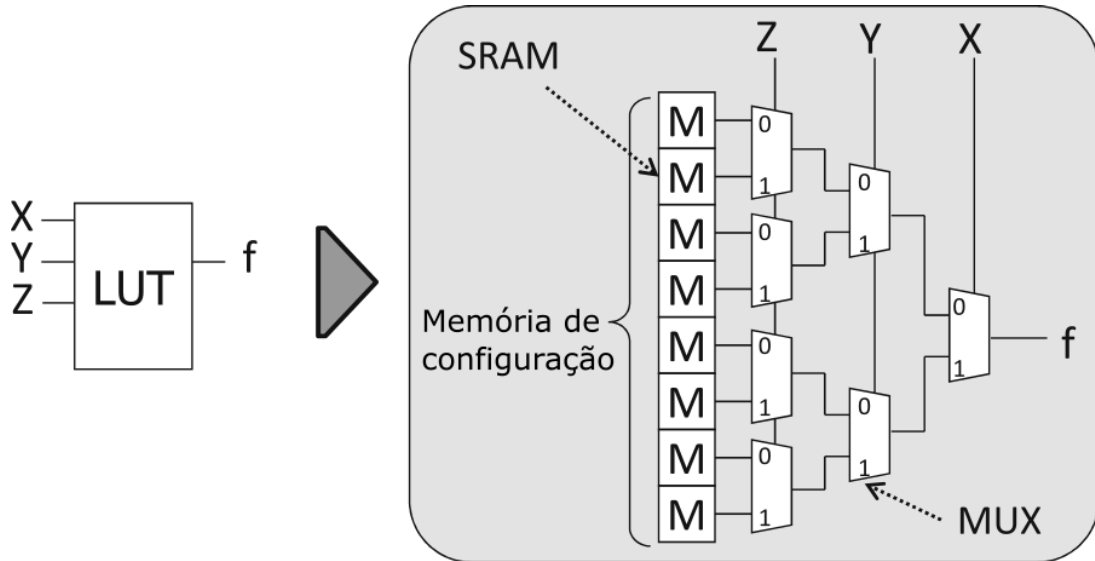

Figura 2.3 – Exemplo de LUT de 3 entradas.

Fonte: Adaptado de Amano (2018).

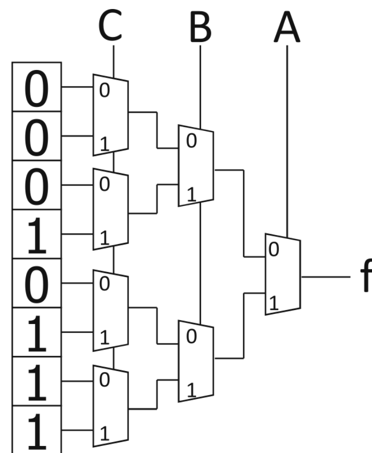

Figura 2.4 – LUT para circuito de maioria de votos.

Fonte: Adaptado de Amano (2018).

configuração. Nestes, também há *trade-off* entre quantidade de *switches* e flexibilidade de roteamento (AMANO, 2018).

2.1.3 Estado-da-arte

A crescente necessidade de fornecer mais eficiência computacional em contraste com a dificuldade em desenvolver sistemas mais rápidos e eficientes (*e.g.* devido à desaceleração da Lei de Moore, dentre outros fatores), tem levado sistemas computacionais a envolverem co-processadores de domínio específico em conjunto com os processadores padrão. Desta forma, tem-se co-processadores que desenvolvem apenas uma tarefa de forma extremamente eficiente, de forma que otimizações previamente utilizadas (*e.g.* caches, execução fora de ordem *etc*) não apresentam vantagens suficientes para aplicações em domínios muito específicos. A especificidade de arquiteturas de domínio específico apresenta um desafio em encontrar domínios que, apesar de específicos, possuam demanda suficiente para justificar a alocação de recursos para tal.

Desta forma, diversas aplicações comerciais que utilizam-se de redes neurais vêm sendo aceleradas através de arquiteturas de domínio específico (HENNESSY, 2017).

2.1.3.1 Google Tensor Processing Unit (TPU)

Devido à intensa demanda de aplicações voltadas à usuários finais utilizando redes neurais em serviços da Google, viu-se a necessidade de construir dispositivos que permitissem inferência com baixa latência para oferecer serviços mais responsivos aos consumidores.

Foi desenvolvido um co-processador para inferência, o *Tensor Processing Unit* (TPU), acoplado a servidores através de PCIe; recebendo instruções desenvolvidas em um *framework* de alto nível (TensorFlow), permitindo assim flexibilidade e uso em aplicações de aprendizado de máquina que venham a surgir. O treinamento da rede neural ainda é executado em GPUs.

Consiste em uma unidade de multiplicação de matrizes (*Matrix Multiply Unit*, MMU) contendo 256x256 multiplicadores e acumuladores (MACs) capazes de realizar operações de multiplicação e adição em inteiros de 8-bits, com ou sem sinal; cujos produtos (16-bits) são salvos em 4 MB de acumuladores de 32-bits (4096x256 elementos de 32-bits); a MMU realiza uma soma parcial de 256 elementos por ciclo de *clock*. Os pesos para a matriz são armazenados em blocos de 64 KB e lidos de uma memória DRAM de 8 GB. Os resultados intermediários são armazenados em um *buffer* unificado de 24 MB. Consiste em um *pipeline* de quatro estágios.

As principais instruções são:

`Read_Host_Memory` carrega dados da memória do servidor para o *buffer* unificado;

`Read_Weights` carrega os pesos da memória para os blocos de pesos;

`MatrixMultiply/Convolve` executa uma operação de multiplicação ou convolução matricial a partir de dados do *buffer* unificado, com resultado armazenado nos acumuladores. A operação matricial multiplica uma entrada variável $B \times 256$ que é multiplicada por pesos constantes 256x256, produzindo uma saída $B \times 256$ e levando B ciclos de *pipeline* para completar;

`Activate` executa a função de ativação do neurônio artificial, com opções para ReLU, sigmoide, dentre outras. Possui como entrada os acumuladores e como saída o *buffer* unificado;

`Write_Host_Memory` armazena dados do *buffer* unificado para o servidor.

Possui 700 MHz de *clock*, 65.536 MACs de 8-bits 28 MBs de memória (*buffer*) interna e *bandwidth* de memória de 34 GB/s e potência de em média 40 Watts, fabricada em tecnologia de 28 nm.

O uso de FPGAs foi abandonado no projeto pois a tecnologia de FPGAs do momento não apresentava competitividade em performance em relação às GPUs (JOUPPI et al., 2017; HENNESSY, 2017).

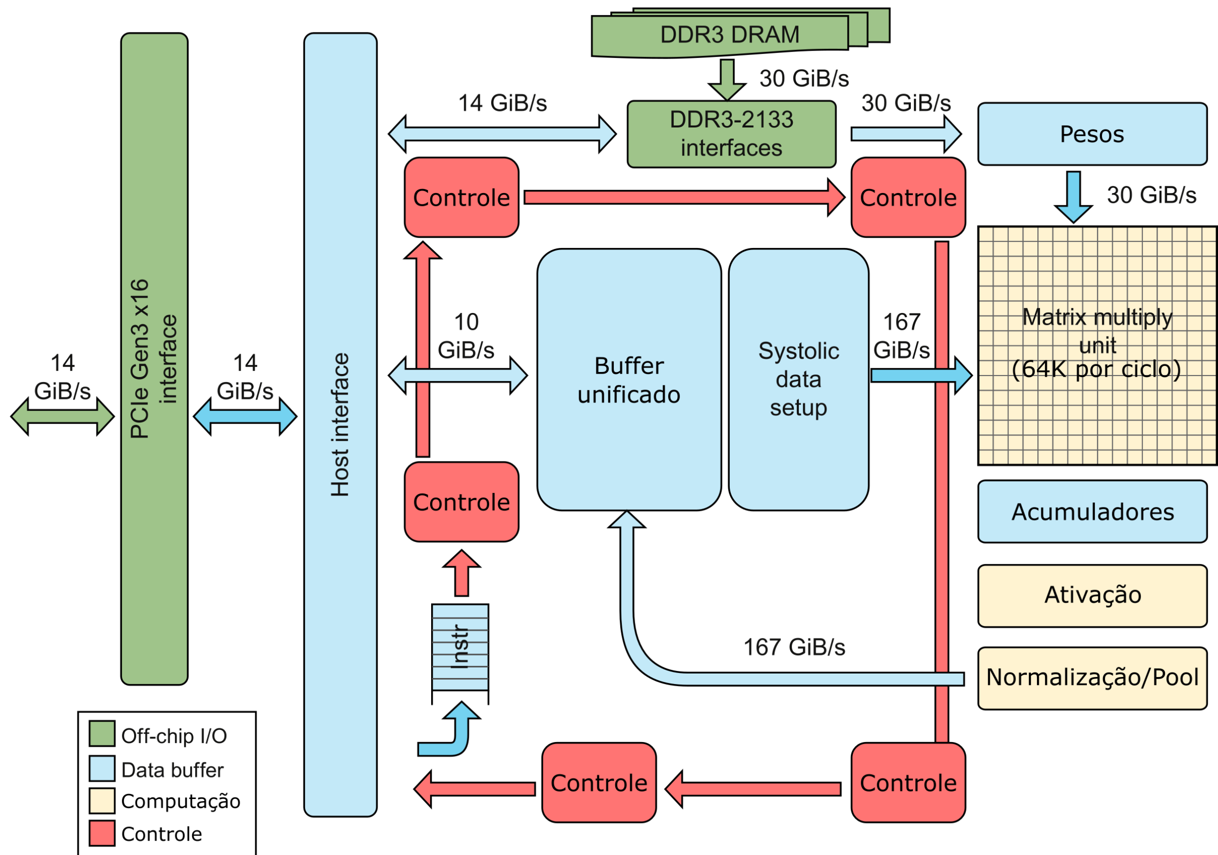

Figura 2.5 – Diagrama de blocos da TPU.

Fonte: Adaptado de Jouppi et al. (2017), Hennessy (2017).

2.1.3.2 Microsoft Catapult

Com o objetivo de acelerar diferentes aplicações em servidores, a Microsoft utilizou-se de FPGAs acopladas aos servidores, que são programadas em linguagem de descrição de hardware (*Hardware Description Languages*, HDL). O servidor utiliza FPGAs Stratix V D5 com 8 GB/s de *bandwidth* PCIe e 21,3 GB/s de *bandwidth* com a memória interna, além de memórias DDR3-1333 de 8 GB. (OVTCHAROV et al., 2015).

O sistema foi utilizado inicialmente para acelerar pesquisas no Bing, o mecanismo de pesquisas da Microsoft, consistindo em múltiplas FPGAs realizando operações diferentes, *e.g.* extração de atributos, *scoring* de resultados de busca, dentre outros. Uma desvantagem de utilizar FPGAs em comparação com ASICs é o *clock* reduzido, desta forma, buscou-se explorar o paralelismo, inclusive através de *pipelining*.

Posteriormente, um acelerador para redes neurais convolucionais (*convolutional neural networks*, CNN) foi desenvolvido, permitindo reconfiguração em tempo de execução com parâmetros que incluem o número de camadas ocultas, dimensão das camadas e precisão numérica. Consiste em blocos de processamento (*processing elements*, PE) arranjados em uma matriz bidimensional. Imagens são enviadas à DRAM e repassadas a

um *buffer* na FPGA. Os PEs realizam as operações que produzem o mapa de atributos de saída. Os resultados finais são recirculados aos *buffers* de entrada para operar na próxima camada da rede neural (OVTCHAROV et al., 2015; HENNESSY, 2017).

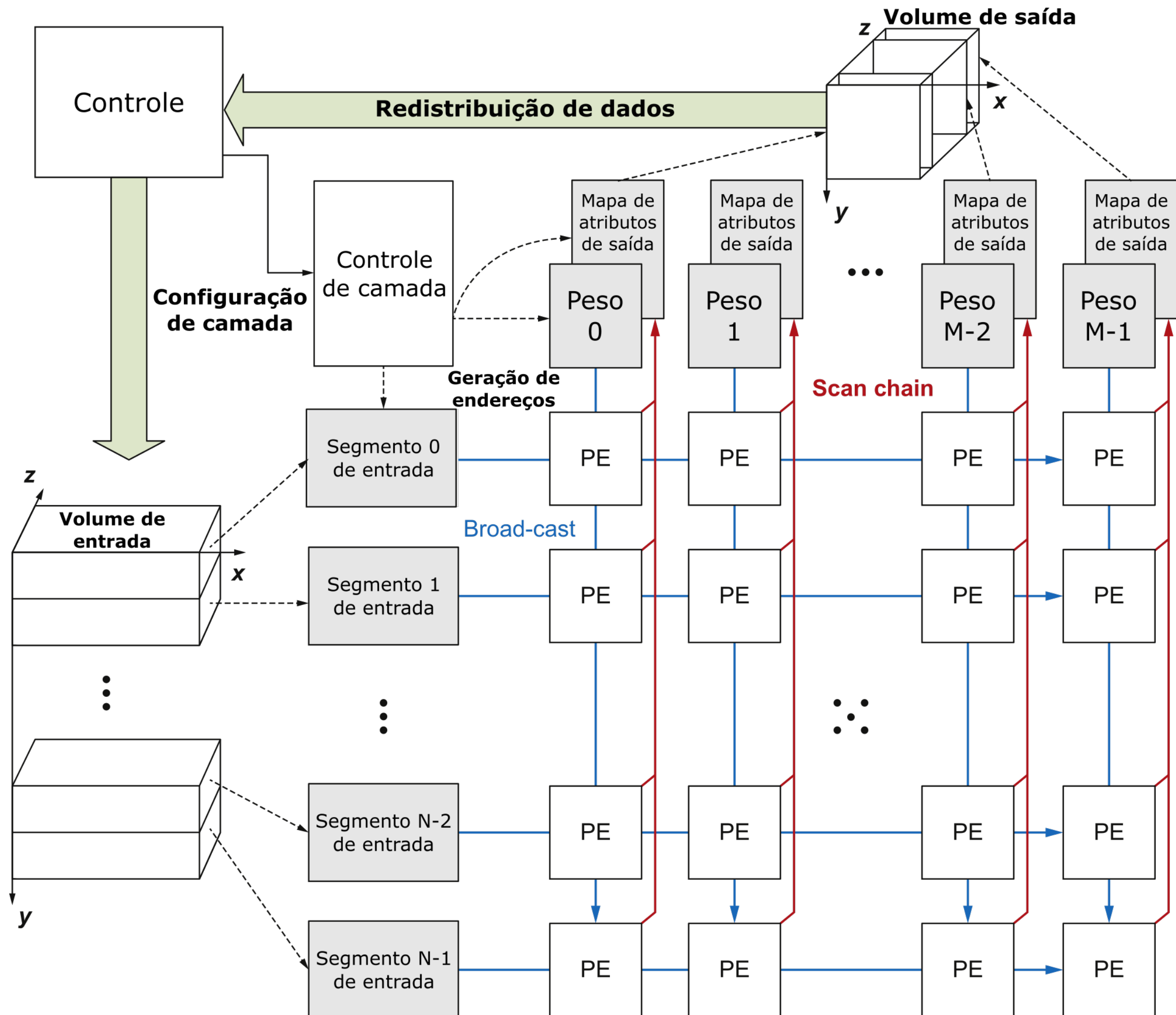

Figura 2.6 – Acelerador de CNNs na Catapult.

Fonte: Adaptado de Hennessy (2017).

2.1.3.3 Microsoft Project Brainwave

A partir do sucesso do acelerador de CNNs Catapult, a Microsoft desenvolveu um sistema de aceleração de redes neurais profundas (*i.e.* múltiplas camadas ocultas) altamente adaptativo e escalável, utilizando-se de uma infraestrutura de FPGAs mais fortemente integrada com o servidor. Utiliza-se principalmente de FPGAs Intel Stratix 10, mas possui flexibilidade para implementação em outros modelos de FPGA (CHUNG et al., 2018).

Seu cerne consiste em unidades de processamento neural (*Neural Processing Unit*, NPU) programadas em cada FPGA que possuem um conjunto de instruções simples e com precisão configurável. O sistema pode distribuir a carga de trabalho em mais de uma FPGA, caso necessário. Os modelos são treinados em *frameworks* de alto nível e convertidos para uma representação intermediária que é posteriormente mapeada para FPGA. Quaisquer operações que não são apropriadas para a FPGA podem ser designadas a CPUs dos servidores.

As NPUs oferecem tipos de dados com precisão específica em tempo de compilação para obter melhor performance e menor perda de acurácia; modelo de programação simples com conjunto de instruções extensivo, adaptável a algoritmos futuros; e microarquitetura escalável que maximiza a eficiência de hardware para tarefas pequenas (ideal para apresentar tempo de resposta rápido para usuários finais). As operações comuns em redes neurais (*i.e.* multiplicação matricial, funções de ativação) são executadas nas NPUs através de multiplicadores e acumuladores e outros hardwares específicos. Os pesos são carregados para as memórias internas das FPGAs a fim de diminuir a latência nas operações (FOWERS et al., 2018; CHUNG et al., 2018).

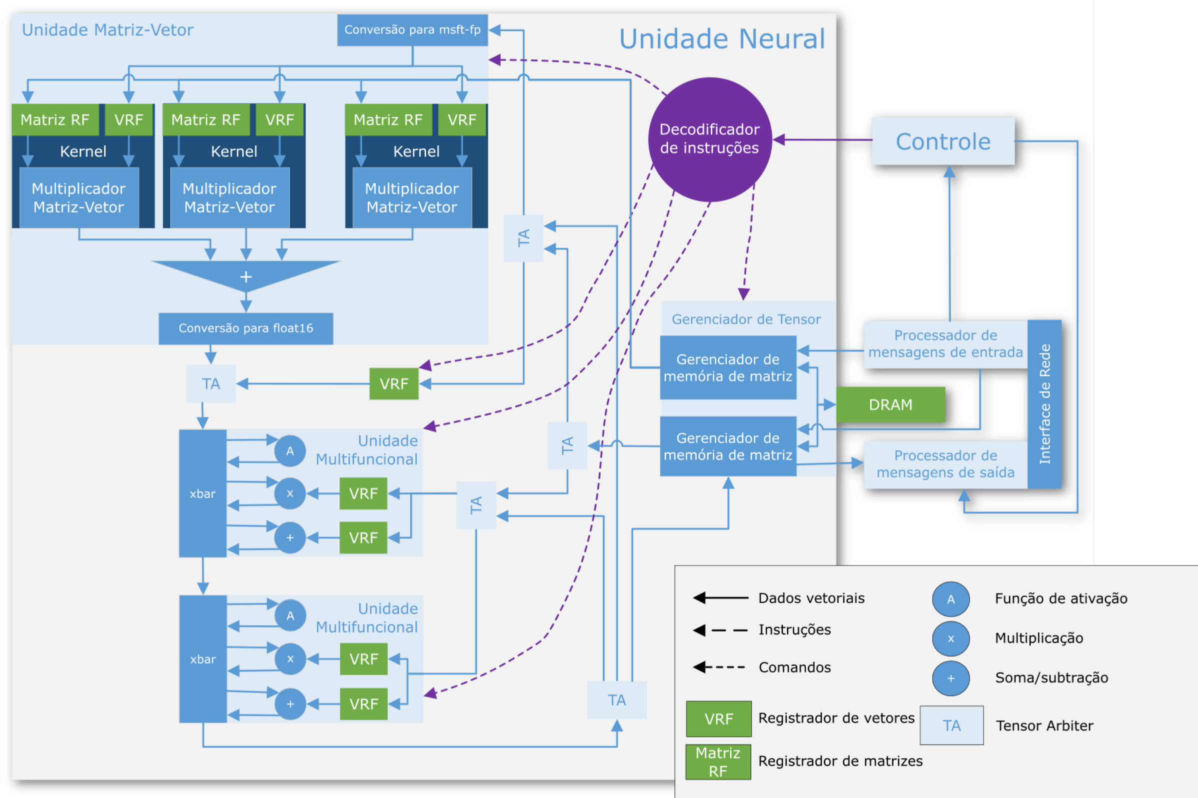

Figura 2.7 – Diagrama da NPU do Project Brainwave.

Fonte: Adaptado de Chung et al. (2018).

2.2 METODOLOGIA

Observam-se, da análise bibliográfica, características dos dispositivos programáveis (FPGAs) que se adéquam às necessidades de redes neurais artificiais.

Redes neurais artificiais derivam seu poder computacional através de sua estrutura distribuída e paralela, além de apresentarem tolerância a falhas devido à distribuição de informação por toda a estrutura da rede. FPGAs possuem paralelismo inerente e são concebidas de forma a explorar esta característica. A possibilidade de auto-modificação de sua topologia e do aprendizado contínuo através de constante modificação dos pesos sinápticos também apresenta-se congruente com a programabilidade de FPGAs (HAYKIN, 2009).

A alta paralelidade e distribuição da informação nas redes neurais permite sacrificar precisão numérica em favor de eficiência computacional (HAYKIN, 2009). Desta forma, performa-se operações numéricas utilizando representação de ponto-fixo na fase de inferência, em contraste com ponto-flutuante para a fase de treinamento, através de quantizações aplicadas aos dados de entrada. (HENNESSY, 2017 apud VANHOUCKE; SENIOR; MAO, 2011). A largura de dados de ponto-fixo é tipicamente de 8 ou 16 bits. Deve-se

observar que a acurácia é reduzida ao realizar esta operação (HENNESSY, 2017 apud BHATTACHARYA; LANE, 2016).

Desta forma, uma arquitetura modular em código de descrição de hardware foi concebida com o intuito de permitir escalabilidade, extensibilidade e flexibilidade para diferentes dispositivos finais (*end-device*) e modelos de redes neurais. A rede neural é primeiro desenvolvida e validada em *frameworks* de alto nível (*e.g.* TensorFlow) de forma a permitir prototipação rápida e treinamento e, posteriormente, o código em VHDL final interligando os módulos é gerado a partir de uma representação intermediária obtida do *framework*.

2.2.1 Implementação em VHDL

O código em linguagem de descrição de hardware (VHDL) foi desenvolvido de forma modular, permitindo o uso de diferentes funções de ativação e mesclagem de diferentes descrições de neurônios ou de funções de ativação (*e.g.* comportamental, estrutural, otimização em área, otimização em velocidade *etc*). A mesma modularidade é observada nas redes neurais artificiais, de forma que o desenvolvimento seguindo o mesmo padrão torna-se intuitivo. A adição de diferentes funções de ativação ou arquiteturas neurais torna-se trivial.

Sua organização hierárquica, do nível mais baixo ao nível mais alto, é dada por:

`types.vhd` Configuração do tamanho da parte inteira e da parte fracionária utilizada no ponto fixo em todos os componentes (padrão `Q16.16`). Também contém algumas funções úteis para conversão e aplicação dos valores em VHDL. Responsável por configurar a precisão desejada para a rede (compromisso entre performance e precisão);

`act_func.vhd` Descrição das funções de ativação. Outras funções de ativação ou reimplementações das funções de ativação existentes devem ser implementadas neste arquivo através de uma arquitetura [1];

`neuron.vhd` Descrição do neurônio artificial. Novas descrições de neurônios devem ser implementadas neste arquivo através de uma arquitetura [2]. Neurônios instanciam uma função de ativação descrita em `act_func.vhd`. Desta forma, é possível mesclar diferentes funções de ativação em uma mesma rede neural, conforme necessário;

`network.vhd` Descrição da rede neural e suas interconexões. Neste arquivo, os neurônios são instanciados e posicionados na rede, conforme a arquitetura desejada. Pode-se descrever mais de uma arquitetura. A entrada é diretamente

[1] do VHDL `architecture`.
[2] *Idem.*

conectada aos neurônios de entrada. O controle da rede é feito através das conexões dos neurônios das camadas escondidas.

Um diagrama da distribuição da configuração modular é exibido na Ilustração 2.3. A rede neural é composta de camadas ocultas $h_1, h_2, \ldots, h_n$ interconectadas entre si.

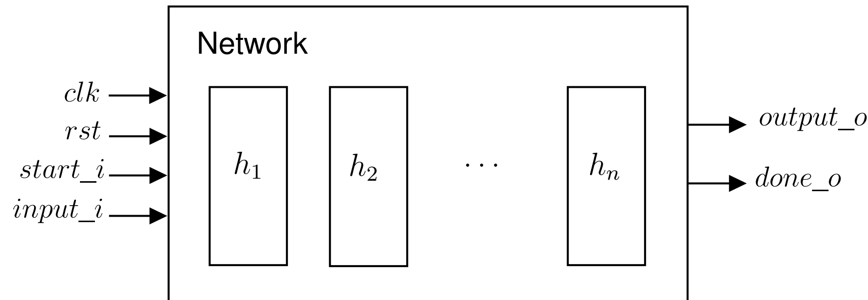

Ilustração 2.3 – Diagrama da rede neural em VHDL.

Fonte: Próprio autor.

Cada camada oculta é composta por neurônios $n_1, n_2, \ldots, n_k$, conforme a Ilustração 2.4. Nota-se que as camadas ocultas correspondem apenas à disposição dos neurônios e não constituem um componente por si só.

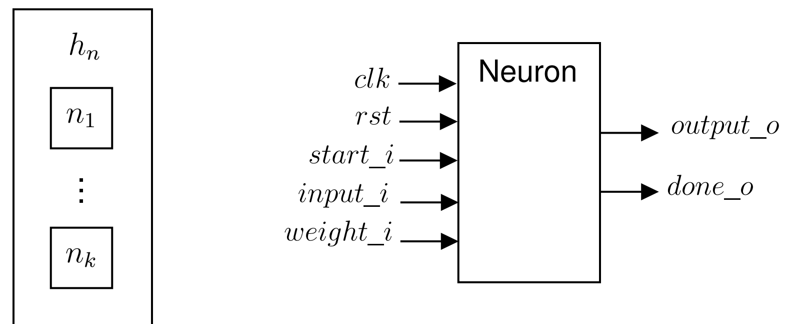

Ilustração 2.4 – Disposição dos neurônios nas camadas ocultas.

Fonte: Próprio autor.

O neurônio (`neuron.vhd`) é controlado através de uma máquina de estados simples, descrita na Ilustração 2.5.

Considerando-se um neurônio k, a descrição dos estados e suas respectivas ações é dada por:

`idle` Estado inicial, aguarda o sinal `start_i` indicando início da computação;

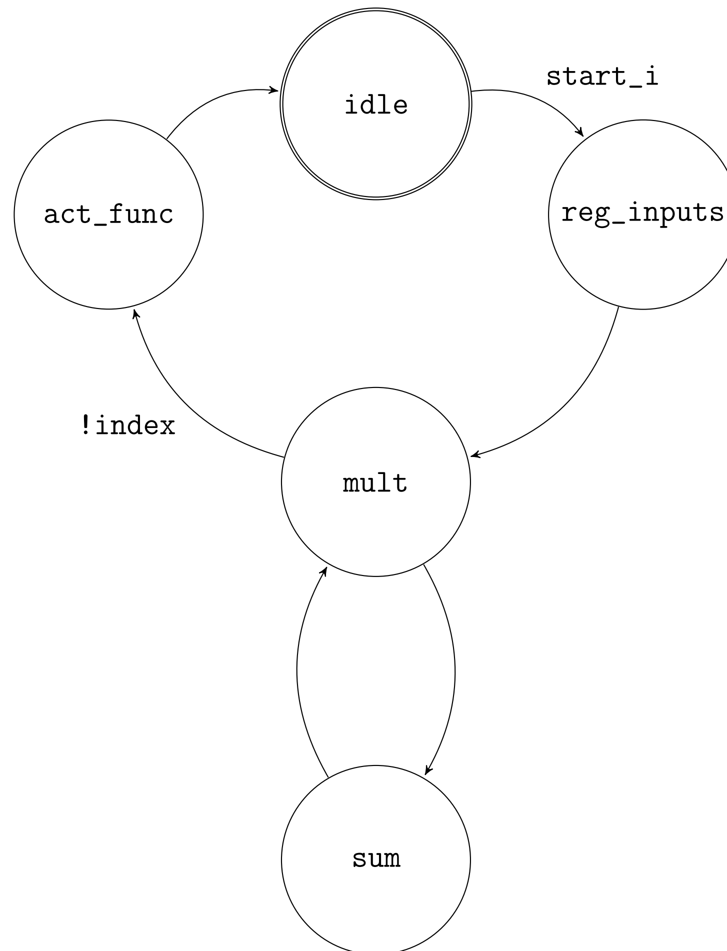

Ilustração 2.5 – Máquina de estados responsável pelo controle do neurônio (`neuron.vhd`).

Fonte: Próprio autor.

`reg_inputs` Registram-se os sinais de entrada $x_0, x_1, \ldots, x_m$; adiciona-se o *bias* b ao acumulador de soma total; registram-se o número de entradas m do neurônio;

`mult` Verifica-se se a computação atual já foi executada para todas as m entradas e, em caso positivo, segue ao estado `act_func`; caso contrário, multiplica-se a entrada x_m por seu respectivo peso sináptico w_{k_m}, onde $m > 1$ (*i.e.* exclui-se *bias*), de forma que:

$$(w_{kj} x_j) + b_k \tag{2.7}$$

`sum` Adiciona-se o resultado da operação anterior com o acumulador de soma total; subtrai-se um do índice responsável por armazenar a entrada sendo operada no momento, tal que:

$$\sum_{j=1}^{m} (w_{kj} x_j) + b_k \tag{2.8}$$

`act_func` Instancia-se a função de ativação escolhida para o neurônio k, definida em uma das arquiteturas de `act_func.vhd`. O resultado contido no acumulador de soma total é utilizado como entrada para a função de ativação e sua saída é instanciada à saída principal do neurônio k. Tem-se, portanto:

$$y_k = \varphi(\sum_{j=1}^{m}(w_{kj}x_j) + b_k) \tag{2.9}$$

2.2.2 Validação da rede

A fim de validar o funcionamento da rede neural artificial desenvolvida em VHDL, uma rede neural simples e amplamente estudada foi realizada.

A arquitetura do *perceptron* multicamadas permite a separação de padrões não-linearmente separáveis. A função de *OU-exclusivo* (*exclusive-OR*, XOR) possui alta relevância na área de engenharia elétrica e computação e consiste em uma classificação não-linear de pontos no hipercubo unitário, possuindo resolução simples por redes neurais através de uma única camada oculta, além de significância histórica por ser responsável por demonstrar a capacidade das redes neurais de separar padrões não-lineares (HAYKIN, 2009).

A tabela-verdade da função XOR é apresentada na Tabela 2.2.

Tabela 2.2 – Tabela-verdade para função XOR de duas entradas.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Fonte: Próprio autor.

Desta forma, o problema da função XOR foi escolhido para validar o sistema desenvolvido.

Por ser um exercício tipicamente utilizado ao introduzir *backpropagation* e o *perceptron* de múltiplas camadas, seus pesos sinápticos são conhecidos, fazendo desta uma boa métrica para comparar o funcionamento da rede.

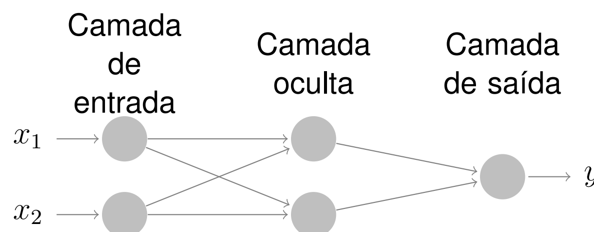

Ilustração 2.6 – Arquitetura da rede neural artificial para resolução do problema XOR.

Fonte: (HAYKIN, 2009).

Para o problema XOR, os pesos e o *bias* do neurônio um (superior) da camada

escondida é dado por:

$$\begin{aligned} w_{11} = w_{12} &= +1 \\ b_1 &= -\frac{3}{2} \end{aligned} \tag{2.10}$$

O neurônio dois (inferior) da camada escondida possui pesos e *bias*:

$$\begin{aligned} w_{21} = w_{22} &= +1 \\ b_2 &= -\frac{1}{2} \end{aligned} \tag{2.11}$$

Os pesos e o *bias* da camada de saída são dados por:

$$\begin{aligned} w_{31} &= -2 \\ w_{32} &= +1 \\ b_3 &= -\frac{1}{2} \end{aligned} \tag{2.12}$$

Utilizando-se destes pesos, pode-se fazer uma rede neural simples através de uma função de ativação *threshold*, para validar o funcionamento do código em VHDL.

Os resultados obtidos em simulação são mostrados na Figura 2.8 e 2.9.

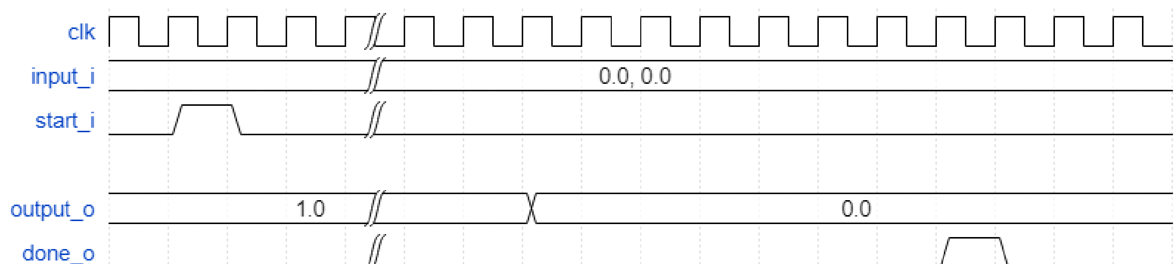

Figura 2.8 – Forma de onda obtida na simulação. Observa-se que a saída é a esperada para a função XOR.

Fonte: Próprio autor.

A forma de onda completa é encontrada na Figura B.1 (Anexo).

O circuito foi sintetizado para uma FPGA *Xilinx Spartan-6 XC6SLX16*, com tecnologia de 45 nm, utilizando o software *Xilinx ISE 14.7*. Um resumo da utilização do dispositivo é exibido na Tabela 2.3.

O esquemático de alto nível (*top-level design*) da rede neural gerado pelo software é exibido na Figura 2.10.

Observa-se, na Figura 2.11, como o esquemático gerado pelo software segue o padrão modular definido, conforme ilustrado na Ilustração 2.3.

O esquemático do neurônio é encontrado na Figura C.1 (Anexo).

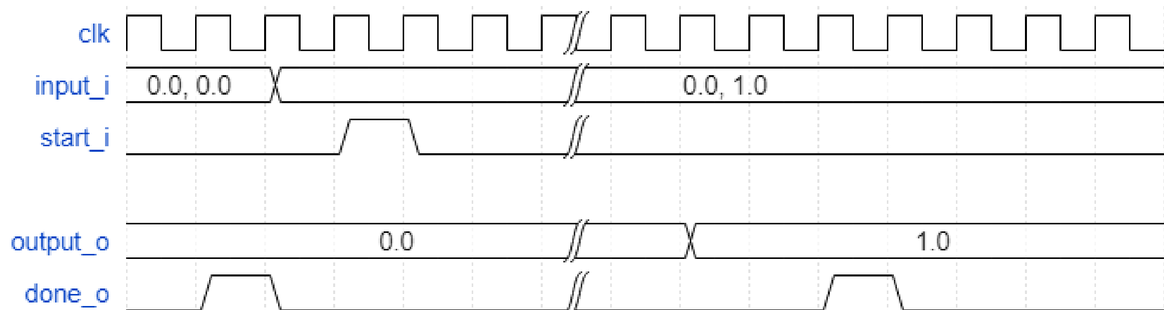

Figura 2.9 – Forma de onda obtida na simulação. Observa-se que a saída é a esperada para a função XOR.

Fonte: Próprio autor.

Tabela 2.3 – Resumo de utilização do dispositivo FPGA para implementação da rede neural da função XOR.

Utilização lógica	**Utilizados**	**Disponíveis**	**Utilização**
Slice Registers	107	18224	0%
Slice LUTs	223	9112	2%
Fully used LUT-FF pairs	105	225	46%
Bonded IOBs	35	232	15%
BUFG/BUFGCTRLs	4	16	25%

Fonte: Próprio autor.

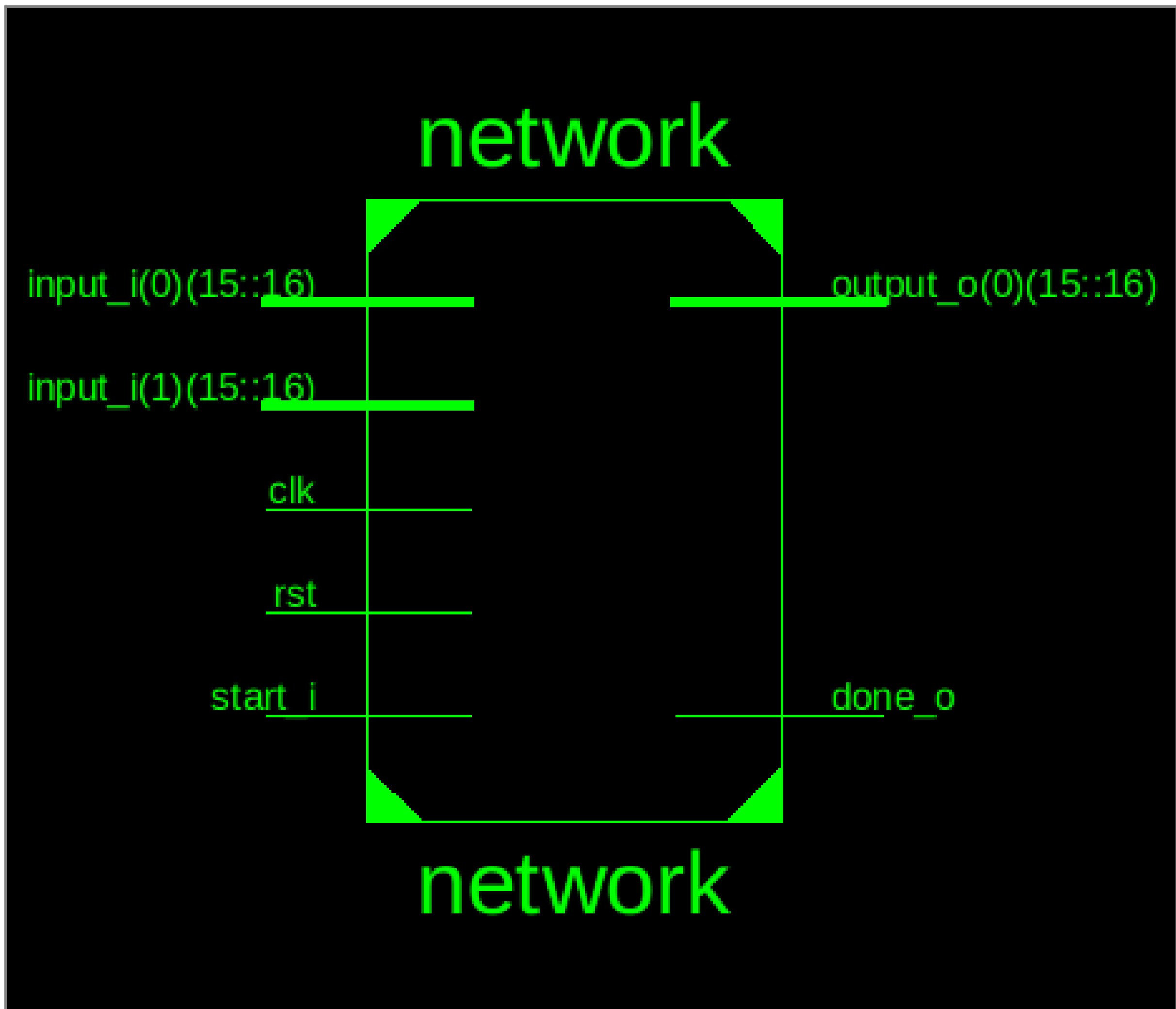

Figura 2.10 – Esquemático de alto nível (*top-level design*) para a rede neural artificial.

Fonte: Próprio autor.

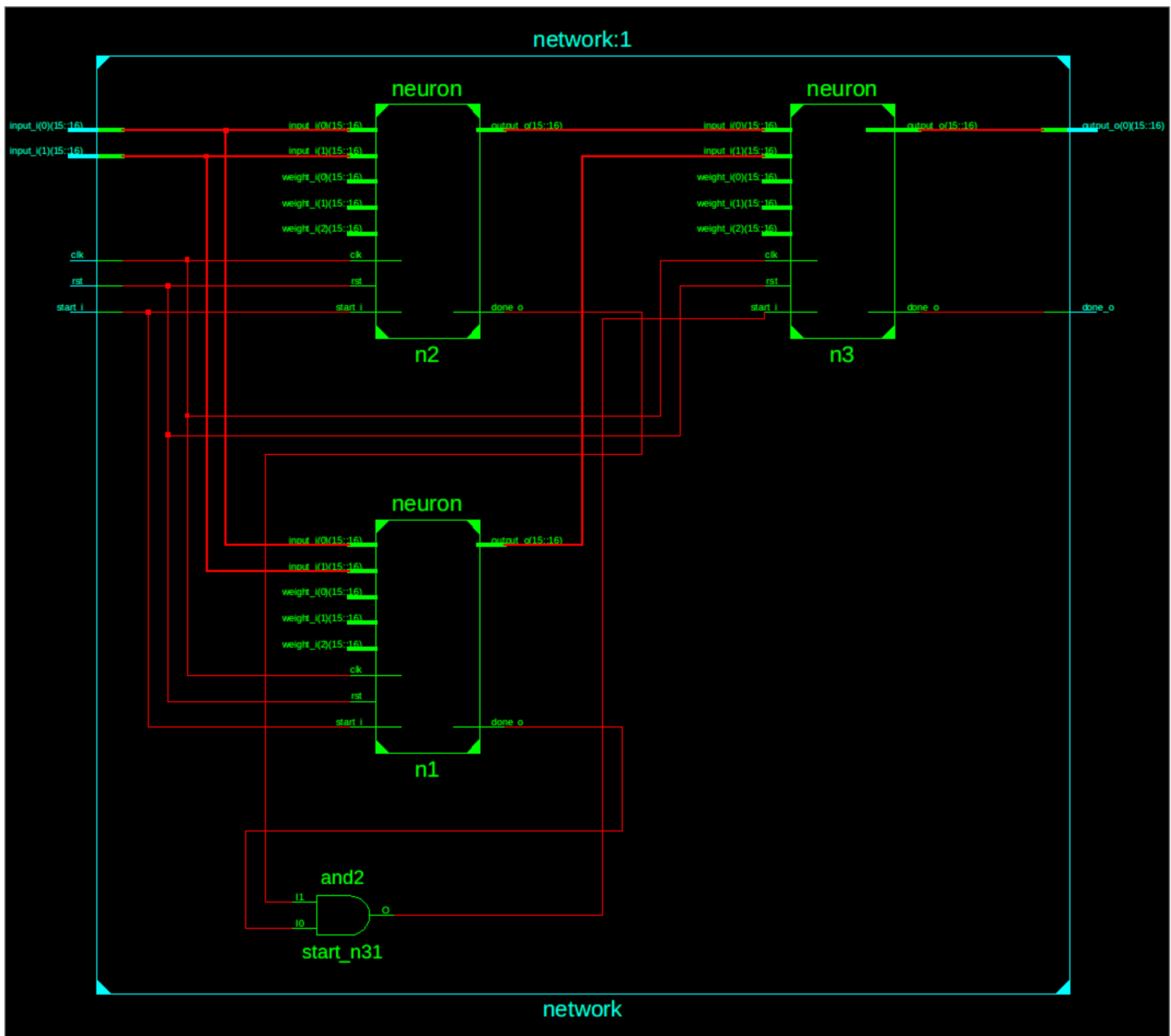

Figura 2.11 – Esquemático dos neurônios artificiais arranjados na topologia da rede neural.

Fonte: Próprio autor.

3 CONCLUSÃO

Observou-se, na revisão bibliográfica e na análise do estado-da-arte, uma tendência na utilização de conjuntos de instruções a fim de implementar as arquiteturas de redes neurais, além de auxílio de ferramentas de alto nível e representações intermediárias para abstração de complexidades.

A abordagem deste trabalho buscou uma alternativa a isso inspirando-se nas similaridades entre as estruturas internas de FPGAs e de redes neurais.

No entanto, observou-se que a alta complexidade e escala massiva inerente às redes neurais artificiais inviabilizou o desenvolvimento efetivo de redes neurais em dispositivos programáveis. Há também uma grande dificuldade em efetuar o roteamento das interconexões complexas das redes neurais através da FPGA.

Há também uma necessidade ímpar em utilizar-se de ferramentas auxiliares de abstração para desenvolvimento e treinamento das redes neurais em alto nível para posterior geração automática de código de descrição de hardware. A extrema complexidade das interconexões neurais torna o desenvolvimento manual inviável.

3.1 TRABALHOS FUTUROS

A partir das estruturas já desenvolvidas, pode-se complementar as funções de ativação de forma a permitir arquiteturas mais complexas de redes neurais artificiais e, portanto, melhores resultados de inferência.

Uma representação intermediária e ferramentas para conversão de arquiteturas de alto nível para código de descrição de hardware também beneficiaria no complemento de trabalhos futuros mais complexos.

A aplicação do código em FPGAs pode comprovar a eficácia e capacidade desta alternativa em dispositivos físicos reais, em contraste com as simulações efetuadas neste trabalho.

REFERÊNCIAS BIBLIOGRÁFICAS

ABADI, M. et al. Tensorflow: A system for large-scale machine learning. In: **12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)**. [s.n.], 2016. p. 265–283. Disponível em: <https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf>.

AMANO, H. (Ed.). **Principles and Structures of FPGAs**. [S.l.]: Springer Singapore, 2018.

AMODEI, D. et al. Deep speech 2 : End-to-end speech recognition in english and mandarin. In: BALCAN, M. F.; WEINBERGER, K. Q. (Ed.). **Proceedings of The 33rd International Conference on Machine Learning**. New York, New York, USA: PMLR, 2016. (Proceedings of Machine Learning Research, v. 48), p. 173–182. Disponível em: <http://proceedings.mlr.press/v48/amodei16.html>.

BHATTACHARYA, S.; LANE, N. D. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In: **Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM - SenSys '16**. [S.l.]: ACM Press, 2016.

BISHOP, C. M. **Pattern Recognition and Machine Learning**. [S.l.]: Springer, 2006.

BOJARSKI, M. et al. End to end learning for self-driving cars. 2016.

CHEN, C. et al. DeepDriving: Learning affordance for direct perception in autonomous driving. In: **2015 IEEE International Conference on Computer Vision (ICCV)**. [S.l.]: IEEE, 2015.

CHUNG, E. et al. Serving dnns in real time at datacenter scale with project brainwave. **IEEE Micro**, IEEE, v. 38, p. 8–20, March 2018. Disponível em: <https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/>.

DEAN, J. et al. Large scale distributed deep networks. In: **NIPS**. [S.l.: s.n.], 2012.

FOWERS, J. et al. A configurable cloud-scale dnn processor for real-time ai. In: . ACM, 2018. Disponível em: <https://www.microsoft.com/en-us/research/publication/a-configurable-cloud-scale-dnn-processor-for-real-time-ai/>.

GLOROT, X.; BORDES, A.; BENGIO, Y. Deep sparse rectifier neural networks. In: GORDON, G.; DUNSON, D.; DUDíK, M. (Ed.). **Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics**. Fort Lauderdale, FL, USA: PMLR, 2011. (Proceedings of Machine Learning Research, v. 15), p. 315–323. Disponível em: <http://proceedings.mlr.press/v15/glorot11a.html>.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>.

HAYKIN, S. S. **Neural networks and learning machines**. Third. Upper Saddle River, NJ: Pearson Education, 2009.

HENNESSY, J. **Computer Architecture**. Elsevier LTD, Oxford, 2017. ISBN 0128119055. Disponível em: <https://www.ebook.de/de/product/29865578/john_hennessy_computer_architecture.html>.

HINTON, G. et al. Deep neural networks for acoustic modeling in speech recognition. **Signal Processing Magazine**, 2012.

JORDAN, M. I.; MITCHELL, T. M. Machine learning: Trends, perspectives, and prospects. **Science**, American Association for the Advancement of Science (AAAS), v. 349, n. 6245, p. 255–260, jul 2015.

JOUPPI, N. P. et al. In-datacenter performance analysis of a tensor processing unit. 2017.

KAESLIN, H. **Top-Down Digital VLSI Design: From Architectures to Gate-Level Circuits and FPGAs**. MORGAN KAUFMANN PUBL INC, 2014. ISBN 0128007303. Disponível em: <https://www.ebook.de/de/product/22684952/hubert_kaeslin_top_down_digital_vlsi_design_from_architectures_to_gate_level_circuits_and_fpgas.html>.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. In: PEREIRA, F. et al. (Ed.). **Advances in Neural Information Processing Systems 25**. Curran Associates, Inc., 2012. p. 1097–1105. Disponível em: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. **Nature**, Springer Science and Business Media LLC, v. 521, n. 7553, p. 436–444, may 2015.

LIN, J.; KOLCZ, A. Large-scale machine learning at twitter. In: **Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2012. (SIGMOD '12), p. 793–804. ISBN 978-1-4503-1247-9. Disponível em: <http://doi.acm.org/10.1145/2213836.2213958>.

LIN, W.-Y.; HU, Y.-H.; TSAI, C.-F. Machine learning in financial crisis prediction: A survey. **IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)**, Institute of Electrical and Electronics Engineers (IEEE), v. 42, n. 4, p. 421–436, jul 2012.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. **The Bulletin of Mathematical Biophysics**, Springer Science and Business Media LLC, v. 5, n. 4, p. 115–133, dec 1943.

MNIH, V. et al. Playing atari with deep reinforcement learning. 2013.

MULLER, U. et al. Off-road obstacle avoidance through end-to-end learning. In: WEISS, Y.; SCHöLKOPF, B.; PLATT, J. C. (Ed.). **Advances in Neural Information Processing Systems 18**. MIT Press, 2006. p. 739–746. Disponível em: <http://papers.nips.cc/paper/2847-off-road-obstacle-avoidance-through-end-to-end-learning.pdf>.

NAJAFABADI, M. M. et al. Deep learning applications and challenges in big data analytics. **Journal of Big Data**, Springer Nature, v. 2, n. 1, feb 2015.

NETZER, Y. et al. Reading digits in natural images with unsupervised feature learning. In: **NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011**. [s.n.], 2011. Disponível em: <http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf>.

OVTCHAROV, K. et al. **Accelerating Deep Convolutional Neural Networks Using Specialized Hardware**. Microsoft Research, 2015. Disponível em: <https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware>.

QIU, J. et al. A survey of machine learning for big data processing. **EURASIP Journal on Advances in Signal Processing**, Springer Nature, v. 2016, n. 1, may 2016.

RADFORD, A. et al. Language models are unsupervised multitask learners. 2019.

RANJAN, R. Streaming big data processing in datacenter clouds. **IEEE Cloud Computing**, Institute of Electrical and Electronics Engineers (IEEE), v. 1, n. 1, p. 78–83, may 2014.

ROSENBLATT, F. The perceptron: A probabilistic model for information storage and organization in the brain. **Psychological Review**, American Psychological Association (APA), v. 65, n. 6, p. 386–408, 1958.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. **Nature**, Springer Nature, v. 323, n. 6088, p. 533–536, oct 1986.

SCHMIDHUBER, J. Deep learning in neural networks: An overview. **Neural Networks**, Elsevier BV, v. 61, p. 85–117, jan 2015.

SILVER, D. et al. Mastering the game of go without human knowledge. **Nature**, Springer Nature, v. 550, n. 7676, p. 354–359, oct 2017.

VANHOUCKE, V.; SENIOR, A.; MAO, M. Z. Improving the speed of neural networks on cpus. In: **Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011**. [S.l.: s.n.], 2011.

WESTE, N.; HARRIS, D. **CMOS VLSI Design: A Circuits and Systems Perspective**. 4th. ed. USA: Addison-Wesley Publishing Company, 2010. ISBN 0321547748, 9780321547743.

ANEXO A – CÓDIGO DA REDE NEURAL ARTIFICIAL EM VHDL

codes/types.vhd

```
1   --
        ------------------------------------------------------------------------
2   --! @file
3   --! @brief Custom types definition for ease of use
4   --! @author Gabriel de Jesus Coelho da Silva
5   --
        ------------------------------------------------------------------------
6
7   --! Use standard library with logic elements
8   library ieee;
9   use ieee.std_logic_1164.all;
10  use ieee.numeric_std.all;
11
12  --! Use proposed library with fixed point definition
13  library ieee_proposed;
14  use ieee_proposed.fixed_pkg.all;
15
16  package types is
17
18          --! std_logic_vector type extension --
19
20          --! The width of the bus
21          constant width : integer := 8;
22
23          --! Bus and array of bus definitions
24          subtype std_logic_bus is std_logic_vector(width - 1 downto 0);
25          type std_logic_bus_array is array (integer range <>) of std_logic_bus;
26
27
28          --! sfixed type extension --
29
30          constant int  : integer := 16; --! Size of the integer part
31          constant frac : integer := 16; --! Size of the fractionary part
32
33          --! Bus and array of bus definitions
34          subtype sfixed_bus is sfixed(int - 1 downto -frac);
35          type sfixed_bus_array is array (integer range <>) of sfixed_bus;
36
37
38          --! Real type extension
```

```
39
40          --! Array of real
41          type real_array is array (integer range <>) of real;
42
43          --! Functions that convert to sfixed using the previosly defined sizes of
44          --! integer and fractionary parts
45          function to_sfixed_a(arg : real) return unresolved_sfixed;
46          function to_sfixed_a(arg : integer) return unresolved_sfixed;
47          function to_real(arg : sfixed_bus_array) return real_array;
48
49  end package types;
50
51  --
52  package body types is
53
54          --! Automatically applies indexes on to_sfixed
55          function to_sfixed_a(arg : real) return unresolved_sfixed is
56                  variable result : unresolved_sfixed(int - 1 downto -frac);
57          begin
58                  result := to_sfixed(arg          => arg,
59                                      left_index   => int - 1,
60                                      right_index  => -frac);
61                  return result;
62          end function to_sfixed_a;
63
64          --! Automatically applies indexes on to_sfixed
65          function to_sfixed_a(arg : integer) return unresolved_sfixed is
66                  variable result : unresolved_sfixed(int - 1 downto -frac);
67          begin
68                  result := to_sfixed(arg          => arg,
69                                      left_index   => int - 1,
70                                      right_index  => -frac);
71                  return result;
72          end function to_sfixed_a;
73
74          --! Converts an sfixed_bus_array to a real_array (for easier visualization)
75          function to_real(arg : sfixed_bus_array) return real_array is
76                  variable result : real_array(arg'range);
77          begin
78                  for i in arg'range loop
79                          result(i) := to_real(arg => arg(i));
80                  end loop;
81                  return result;
82          end function to_real;
83
84  end package body types;
```

codes/act_func.vhd

```
1   --
        ------------------------------------------------------------------------
2   --! @file
3   --! @brief Definition of  activation  functions
4   --! @author Gabriel de Jesus Coelho da Silva
5   --
        ------------------------------------------------------------------------
6
7   --! Use standard library with logic  elements
8   library  ieee;
9   use ieee.std_logic_1164.all;
10  use ieee.numeric_std.all;
11
12  --! Use proposed library with fixed point  definition
13  library  ieee_proposed;
14  use ieee_proposed.fixed_pkg.all;
15
16  --! Use custom library for ease of use
17  use work.types.all;
18
19  --! Applies the specified  activation  function  to  the  input
20
21  --! This element takes as input a sfixed_bus (@see @file types.vhd) and outputs
22  --! the activation  function  specified  as the  instantiated  architecture .
23  entity  act_func is
24          port(
25                  clk         :  in   std_logic;  --! Clock, for synchronous implementation
26                  input_i     :  in   sfixed_bus; --! Activation function  input
27                  output_o :  out sfixed_bus :=  (others => '0')  --! Activation function
28                                                                   --! output
29          );
30  end entity act_func;
31
32  -- @brief Threshold
33  -- @details McCulloch-Pitts "all-or-none" activation function (threshold).
34  architecture threshold of  act_func is
35  begin
36          output_o <= to_sfixed_a(1) when input_i >= 0 else to_sfixed_a(0);
37  end architecture threshold;
38
39  -- @brief Rectified Linear Unit (ReLU)
40  -- @details A simple ReLU (f(x) = max(0, x))
41  architecture relu  of  act_func is
42  begin
```

```
43          output_o <= input_i when input_i >= 0 else to_sfixed_a(0);
44  end architecture relu;
```

codes/neuron.vhd

```
 1  --
    ------------------------------------------------------------------------------------------------------------
 2  --! @file
 3  --! @brief Single neuron instance
 4  --! @author Gabriel de Jesus Coelho da Silva
 5  --
    ------------------------------------------------------------------------------------------------------------
 6
 7  --! Use standard library with logic elements
 8  library ieee;
 9  use ieee.std_logic_1164.all;
10  use ieee.numeric_std.all;
11
12  --! Use proposed library with fixed point definition
13  library ieee_proposed;
14  use ieee_proposed.fixed_pkg.all;
15
16  --! Use custom library for ease of use
17  use work.types.all;
18
19  --! A single neuron with generic "N" inputs
20
21  --! This element takes "N" inputs and "N + 1" weights of the type
22  --! sfixed_bus_array (@see @file types.vhd), multiplies the "N"th input by the
23  --! "N+1"th weight, sums the resulting multiplications and outputs the
24  --! activation function defined by the instantiated architecture (@see @file
25  --! act_func.vhd) applied to the sum.
26  entity neuron is
27          generic(
28                  inputs : integer := 3           --! Number of inputs into the neuron
29          );
30          port(
31                  clk      : in std_logic;        --! Clock input
32                  rst      : in std_logic;        --! Reset input
33                  start_i  : in std_logic;        --! Start input, indicates to start the calculation
34                  input_i  : in sfixed_bus_array(inputs - 1 downto 0); --! Neuron inputs
35                  weight_i : in sfixed_bus_array(inputs downto 0);     --! Neuron weights, including bias
36                  output_o : out sfixed_bus := (others => '0');        --! Neuron output
37                  done_o   : out std_logic := '0' --! Done output, indicates completion
38          );
39  end entity neuron;
```

```
40
41 architecture behavioral of neuron is
42         type state is (idle, reg_inputs, mult, sum, act_func);
43         signal current_state, next_state : state;
44
45         signal input_s          : sfixed_bus_array(inputs − 1 downto 0) := (others => (others => '0'));
46         signal sum_s            : sfixed(int downto −frac)               := (others => '0');
47         signal mult_s           : sfixed(2 * int − 1 downto −2 * frac)   := (others => '0');
48         signal index            : integer                                := inputs;
49         signal weight_s         : sfixed_bus_array(inputs downto 0)      := weight_i;
50         signal output_s         : sfixed_bus                             := (others => '0');
51         signal act_func_input : sfixed_bus                               := resize(sum_s, int − 1, −frac);
52         signal done_s           : std_logic                              := '0';
53 begin
54
55         fsm_lower : process(clk, rst) is
56         begin
57                 −− synchronous reset results in better performance
58                 −− because of multiplier blocks and RAM registers inferred
59                 if rising_edge(clk) then
60                         if rst = '1' then
61                                 current_state <= idle;
62                 elsif rising_edge(clk) then
63                         current_state <= next_state;
64                 end if;
65         end process fsm_lower;
66
67         fsm_upper : process(current_state, input_i, input_s, start_i, weight_s, mult_s) is
68                 −− do NOT include "index" or "sum_s" here
69         begin
70                 case current_state is
71                         when idle =>
72                                 done_s <= '0';
73                                 if start_i = '1' then
74                                         next_state <= reg_inputs;
75                                 else
76                                         next_state <= current_state;
77                                 end if;
78
79                         when reg_inputs =>
80                                 input_s <= input_i;
81                                 sum_s  <= resize(weight_s(inputs), int, −frac);
82                                 −− bias is already added to sum
83                                 mult_s  <= (others => '0');
84                                 index   <= inputs;
85
86                                 next_state <= mult;
```

```
87
88                      when mult =>
89                          if index = 0 then
90                              next_state <= act_func;
91                          else
92                              mult_s     <= input_s(index - 1) * weight_s(index - 1);
93                              next_state <= sum;
94                          end if;
95
96                      when sum =>
97                          index      <= index - 1;
98                          sum_s      <= resize(sum_s, int - 1, -frac) + resize(mult_s, int - 1, -
    frac);
99                          next_state <= mult;
100
101                     when act_func =>
102                         done_s     <= '1';
103                         next_state <= idle;
104
105                 end case;
106
107         end process fsm_upper;
108
109         --! Activation function instantiation
110         act_func_inst : entity work.act_func(tanh)
111             port map(
112                 input_i   => act_func_input,
113                 output_o => output_s
114             );
115
116         done_o         <= done_s;
117         output_o       <= output_s;
118         weight_s       <= weight_i;
119         act_func_input <= resize(sum_s, int - 1, -frac);
120
121 end architecture behavioral;
```

codes/network.vhd

```
1 --
    ----------------------------------------------------------------------

2 --! @file
3 --! @brief Neural network instantiation
4 --! @author Gabriel de Jesus Coelho da Silva
5 --
    ----------------------------------------------------------------------
```

```
6
7   --! Use standard library with logic elements
8   library ieee;
9   use ieee.std_logic_1164.all;
10  use ieee.numeric_std.all;
11
12  --! Use proposed library with fixed point definition
13  library ieee_proposed;
14  use ieee_proposed.fixed_pkg.all;
15
16  --! Use custom library for ease of use
17  use work.types.all;
18
19  entity network is
20      generic(
21          inputs  : integer := 2;     --! Network inputs
22          outputs : integer := 1      --! Network outputs
23      );
24      port(
25          clk      : in  std_logic;   --! Clock input
26          rst      : in  std_logic;   --! Reset output
27          start_i  : in  std_logic;   --! Start input, indicates to start the calculation
28          input_i  : in  sfixed_bus_array(inputs - 1 downto 0); --! Network input
29          output_o : out sfixed_bus_array(outputs - 1 downto 0) := (others => (others => '0')); --!
      Network output
30          done_o   : out std_logic                := '0'  --! Done output, indicates
      completion
31      );
32  end entity network;
33
34  architecture n_xor of network is
35      signal output_n1 : sfixed_bus;
36      signal output_n2 : sfixed_bus;
37      signal done_n1   : std_logic;
38      signal done_n2   : std_logic;
39      signal start_n3  : std_logic;
40      signal output_s  : sfixed_bus_array(outputs - 1 downto 0);
41      signal input_n3  : sfixed_bus_array(inputs - 1 downto 0);
42      constant weight_n1 : sfixed_bus_array(inputs downto 0) := (( to_sfixed_a(-1.5) ),
43                                                                 ( to_sfixed_a(1) ),
44                                                                 ( to_sfixed_a(1) ));
45      constant weight_n2 : sfixed_bus_array(inputs downto 0) := (( to_sfixed_a(-0.5) ),
46                                                                 ( to_sfixed_a(1) ),
47                                                                 ( to_sfixed_a(1) ));
48      constant weight_n3 : sfixed_bus_array(inputs downto 0) := (( to_sfixed_a(-0.5) ),
49                                                                 ( to_sfixed_a(-2) ),
50                                                                 ( to_sfixed_a(1) ));
```

```
51  begin
52
53          -- First layer
54
55          n1 : entity work.neuron
56                  generic map(
57                          inputs => inputs
58                  )
59                  port map(
60                          clk      => clk,
61                          rst      => rst,
62                          start_i  => start_i,
63                          input_i  => input_i,
64                          weight_i => weight_n1,
65                          output_o => output_n1,
66                          done_o   => done_n1
67                  );
68
69          n2 : entity work.neuron
70                  generic map(
71                          inputs => inputs
72                  )
73                  port map(
74                          clk      => clk,
75                          rst      => rst,
76                          start_i  => start_i,
77                          input_i  => input_i,
78                          weight_i => weight_n2,
79                          output_o => output_n2,
80                          done_o   => done_n2
81                  );
82
83          -- Second layer
84          n3 : entity work.neuron
85                  generic map(
86                          inputs => inputs
87                  )
88                  port map(
89                          clk      => clk,
90                          rst      => rst,
91                          start_i  => start_n3,
92                          input_i  => input_n3,
93                          weight_i => weight_n3,
94                          output_o => output_s(0),
95                          done_o   => done_o
96                  );
97
```

```
 98         input_n3 <= ((output_n1), (output_n2));
 99         start_n3 <= done_n1 and done_n2;
100         output_o <= output_s;
101
102 end architecture n_xor;
```

codes/network_tb.vhd

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3  use ieee.numeric_std.all;
 4  use work.types.all;
 5
 6  entity network_tb is
 7  end entity network_tb;
 8
 9  architecture n_xor of network_tb is
10
11          constant inputs : integer := 2;
12          constant outputs : integer := 1;
13
14          constant half_period : time := 10 ns;
15
16          signal clk      : std_logic := '0';
17          signal rst      : std_logic := '0';
18          signal input_i  : sfixed_bus_array(inputs - 1 downto 0);
19          signal start_i  : std_logic := '0';
20          signal output_o : sfixed_bus_array(outputs - 1 downto 0);
21          signal done_o   : std_logic;
22          signal input_r  : real_array(inputs - 1 downto 0);
23          signal output_r : real_array(outputs - 1 downto 0);
24
25  begin
26
27          network_inst : entity work.network(n_xor)
28                  generic map(
29                          inputs  => inputs,
30                          outputs => outputs
31                  )
32                  port map(
33                          clk      => clk,
34                          rst      => rst,
35                          start_i  => start_i,
36                          input_i  => input_i,
37                          output_o => output_o,
38                          done_o   => done_o
39                  );
40
```

```
41                  clk  <= not clk  after  half_period;
42
43                   start_stimuli   :  process is
44                  begin
45                           start_i  <=  '0',   '1'  after  15 ns,  '0'  after  30 ns;
46                           wait until  done_o = '1';
47                  end process start_stimuli;
48
49
50                   input_stimuli   :  process is
51                  begin
52                           input_i  <= ((  to_sfixed_a(0)  ),  (  to_sfixed_a(0)  ));
53                           wait until  done_o = '1';
54                           input_i  <= ((  to_sfixed_a(0)  ),  (  to_sfixed_a(1)  ));
55                           wait until  done_o = '1';
56                           input_i  <= ((  to_sfixed_a(1)  ),  (  to_sfixed_a(0)  ));
57                           wait until  done_o = '1';
58                           input_i  <= ((  to_sfixed_a(1)  ),  (  to_sfixed_a(1)  ));
59                           wait until  done_o = '1';
60                  end process input_stimuli;
61
62                  input_r  <= to_real( input_i );
63                  output_r <= to_real(output_o);
64
65   end architecture n_xor;
```

ANEXO B – FORMA DE ONDA COMPLETA

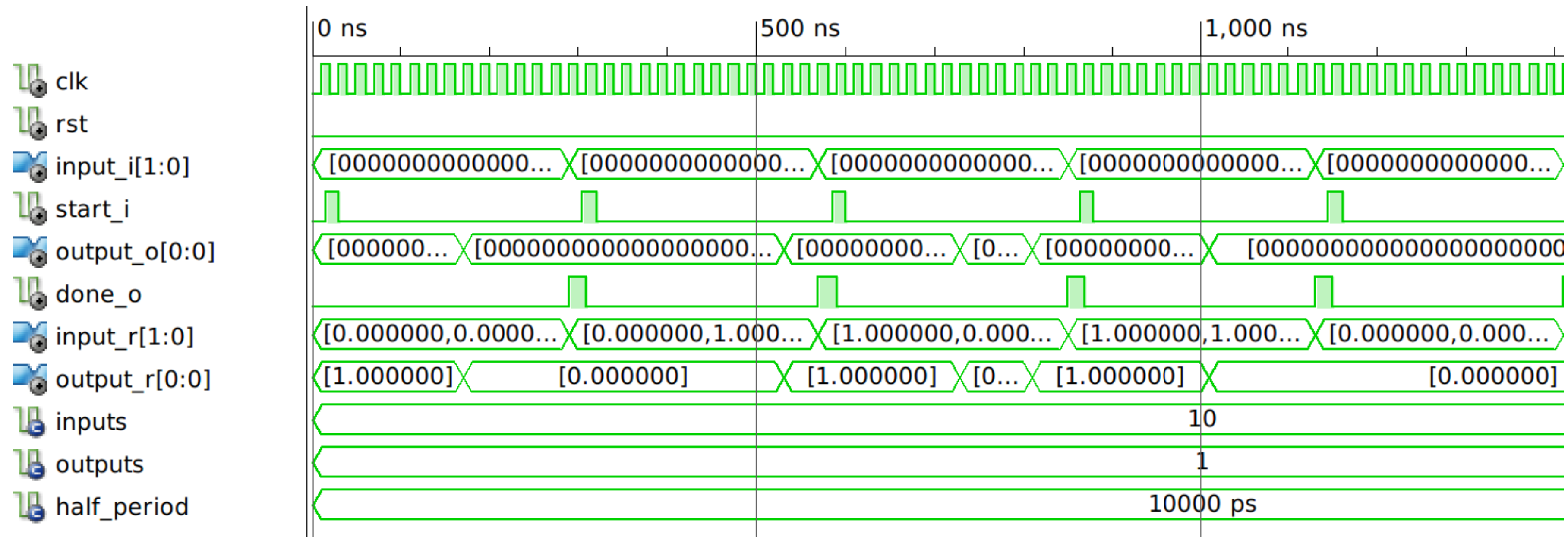

Figura B.1 – Forma de onda completa para o circuito da rede neural para a função XOR.

Fonte: Próprio autor.

ANEXO C – ESQUEMÁTICO DO NEURÔNIO ARTIFICIAL

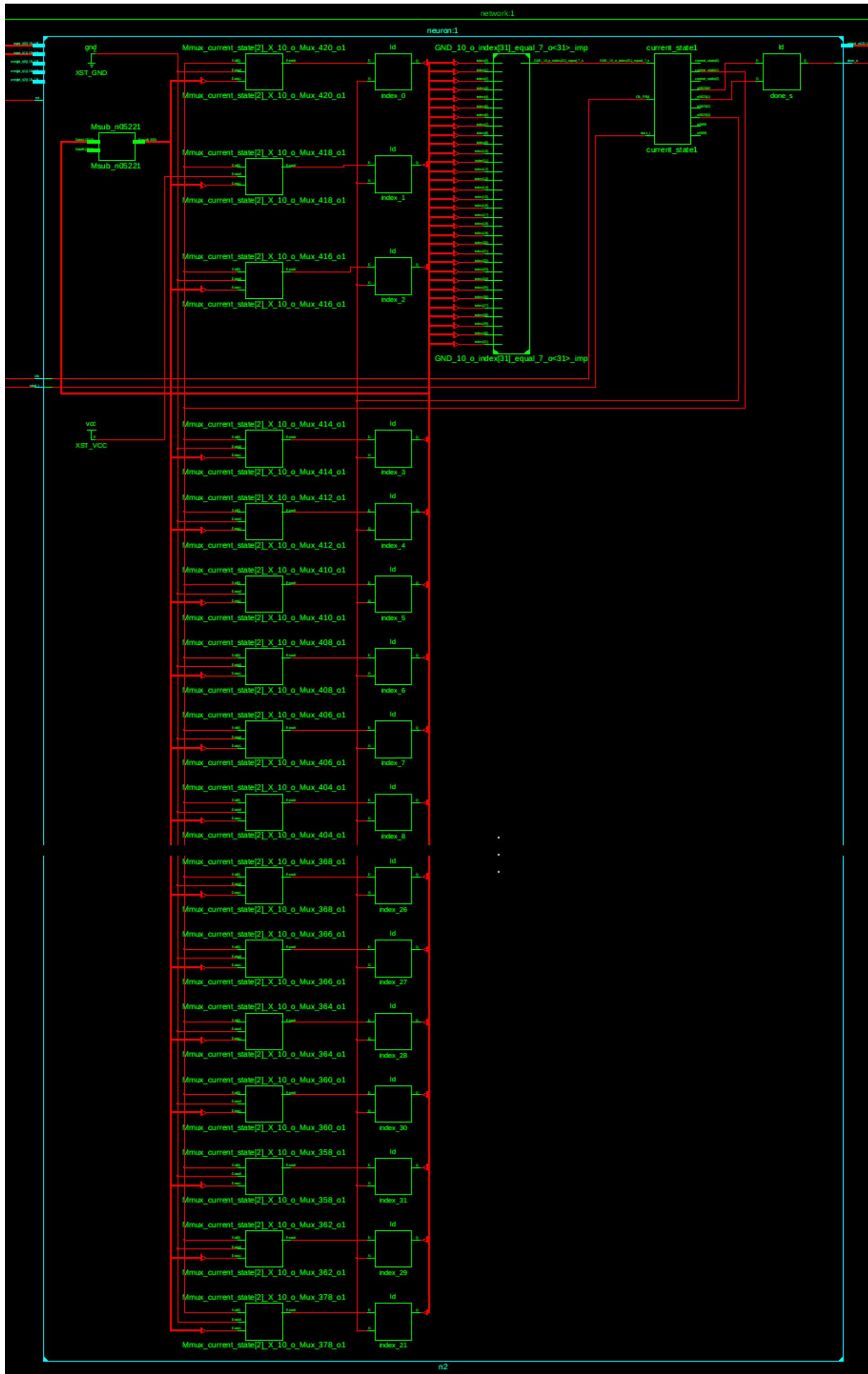

Figura C.1 – Esquemático do neurônio artificial.

Fonte: Próprio autor.