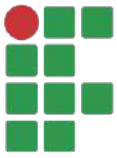


INSTITUTO FEDERAL
Paraíba
Campus
João Pessoa

UNIDADE ACADÊMICA DE CONTROLE E PROCESSOS INDUSTRIAIS
COORDENAÇÃO DE ENGENHARIA ELÉTRICA

VICTOR HERBERT FERREIRA DE SOUSA

ACELERADOR DE HARDWARE EM FPGA PARA INFERÊNCIA EM REDES NEURAIS ARTIFICIAIS

JOÃO PESSOA
2022

VICTOR HERBERT FERREIRA DE SOUSA

ACELERADOR DE HARDWARE EM FPGA PARA INFERÊNCIA EM REDES NEURAIS ARTIFICIAIS

Trabalho de Conclusão de Curso apresentado à Coordenação de Engenharia Elétrica do Instituto Federal da Paraíba, como requisito parcial para a obtenção do título de Engenheiro Eletricista.

Orientador: Prof. Dr. Lincoln Machado de Araújo

JOÃO PESSOA
2022

Dados Internacionais de Catalogação na Publicação (CIP)
Biblioteca Nilo Peçanha do IFPB, *campus* João Pessoa

S725a Sousa, Victor Herbert Ferreira de.

Acelerador de hardware em FPGA para inferência em redes neurais artificiais / Victor Herbert Ferreira de Sousa. – 2022.
76 f. : il.
TCC (Graduação - Curso Superior de Bacharelado em Engenharia Elétrica) -Instituto Federal de Educação da Paraíba / Unidade Acadêmica de Processos Industriais, 2022.
Orientação : Profº D.r Lincoln Machado de Araújo.

1.Redes neurais artificiais. 2. Projeto de hardware digital. 3. Sistemas digitais. 4. FPGAs. 5. Arquitetura de computador. I. Título.

CDU 004.032.26(043)

Lucrecia Camilo de Lima
Bibliotecária - CRB 15/132

VICTOR HERBERT FERREIRA DE SOUSA

ACELERADOR DE HARDWARE EM FPGA PARA INFERÊNCIA EM REDES NEURAIS ARTIFICIAIS

Trabalho de Conclusão de Curso apresentado à Coordenação de Engenharia Elétrica do Instituto Federal da Paraíba, como requisito parcial para a obtenção do título de Engenheiro Eletricista.

BANCA EXAMINADORA

gov.br Documento assinado digitalmente
LINCOLN MACHADO DE ARAUJO
Data: 01/03/2023 16:59:11-0300
Verifique em https://verificador.iti.br

Prof. Dr. Lincoln Machado de Araújo
Orientador — Instituto Federal da Paraíba

gov.br Documento assinado digitalmente
CLEUMAR DA SILVA MOREIRA
Data: 01/03/2023 18:09:51-0300
Verifique em https://verificador.iti.br

Prof. Dr. Cleumar da Silva Moreira
Membro da banca — Instituto Federal da Paraíba

gov.br Documento assinado digitalmente
PATRIC LACOUTH DA SILVA
Data: 01/03/2023 17:35:03-0300
Verifique em https://verificador.iti.br

Prof. Dr. Patric Lacouth da Silva
Membro da banca — Instituto Federal da Paraíba

João Pessoa, 23 de fevereiro de 2023

AGRADECIMENTOS

Acredito que antes mesmo do esforço são necessárias oportunidades, assim sendo, nada mais justo que perante ao trabalho e sacrifícios dos meus pais, Girlene Ferreira e Herbert Cless, essa secção seja dedicada primeiramente a eles. Não obstante, minha trajetória acadêmica foi encorajada por diversos professores: Andréa Lira, Andréa Samara, Dhiego Veloso, Luís Romeu, Lincoln Machado, Valéria Cavalcanti e Thiago Gouveia. Para todos vocês guardo meu afeto e gratidão por toda dedicação em construir um mundo melhor.

RESUMO

A era da informação traz uma série de novas demandas: aumento no poder computacional bruto, computação ubíqua e inteligência artificial são alguns exemplos. A melhoria de desempenho é alcançada por meio de otimizações de hardware para problemas específicos. É nesse sentido que o presente trabalho visa construir um acelerador de hardware para o algoritmo de inferência em redes neurais artificiais. A arquitetura é customizável em parâmetros como número de núcleos de processamento, onde variáveis de customização podem ser facilmente modificados em tempo de síntese alterando os parâmetros definidos com uso da linguagem de descrição de hardware SystemVerilog. O acelerador possui memórias internas que possibilitam salvar pesos sinápticos, entradas, saídas e dimensões das camadas das rede que podem ser escritas por um dispositivo mapeado de memória. As funções de ativação podem ser escolhidas dentre três implementadas em hardware e quatro por aproximação linear em LUT, que podem ser programadas em tempo de execução. O design possui três estágios de pipeline e data forwarding.A arquitetura foi submetida a testes de simulação e acurácia em comparação com um modelo padrão e foi verificada com métricas de cobertura de código. Uma análise sobre a performance em ciclos de clock e frequência máxima para uma tecnologia de referência também foram disponibilizadas. O trabalho termina com sugestões de melhorias não executadas, em especial para implementação do algoritmo de treinamento por gradiente descendente, com poucas modificações na arquitetura.

Palavras-chave: Projeto de hardware digital, Sistemas digitais, Electronic Design Automation - EDA.

ABSTRACT

The information age brings a series of new demands: an increase in raw computational power, ubiquitous computing, and artificial intelligence are some examples. Performance improvement is achieved through hardware optimizations for specific issues. In this sense, the present work aims to build a hardware accelerator for the inference algorithm in artificial neural networks. The architecture is customizable in parameters such as the number of processing cores. Customization variables can be easily modified in synthesis time by changing the defined parameters using the SystemVerilog hardware description language. The accelerator has internal memories that make it possible to save synaptic weights, inputs, outputs, and dimensions of the network layers that a memory-mapped device can write. The activation functions can be chosen among three implemented in hardware and four by linear approximation in LUT, which can be programmed at runtime. The design has three pipeline stages and data forwarding. The architecture was subjected to simulation and accuracy tests against a standard model and was verified with code coverage metrics. An analysis of performance in clock cycles and the maximum frequency for a reference technology is also available. The work ends with suggestions for improvements not implemented, especially for implementing the gradient descent training algorithm, with few changes in the architecture.

Keywords: Digital Hardware Design, Digital Systems, Electronic Design Automation - EDA.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
EDA	Electronic Design Automation
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
GDS	Graphic Design System
HDL	Hardware Description Language
IC	Integrated Circuit
LUT	Lookup Table
MAC	Multiply Accumulate
MSE	Mean Square Error
ReLU	Rectified Linear Unit
RTL	Register Transfer Level
VLSI	Very Large Scale Integration

LISTA DE SÍMBOLOS

$\neg$	Operador lógico de negação (*Not*)
$\vee$	Operador lógico Ou (*Or*)
$\wedge$	Operador lógico E (*And*)
$\otimes$	Operador lógico Ou Exclusivo (*Xor*)
X	Vetor de entrada
W	Matriz de pesos
B	Vetor de bias
V	Vetor produto
φ	Função de ativação
Y	Vetor de saída
η	Taxa de Aprendizagem

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

A ideia de usar aceleradores de hardware ou circuitos integrados dedicados para otimização de operações em redes neurais artificais não é exatamente nova. Philips' L-Neuro, Intel's Ni1000, Siemens' MA16 e SYNAPSE-1 são só alguns exemplos de chips produzidos no século passado projetados para a aplicação específica de processamento neural (IENNE et al., 1995). Da mesma data também apareciam implementações de coprocessadores em FPGA (RYBARCZYK; SZULC, 2002) e em processadores de propósito geral (KIM; KANG; LEE, 1998). Uma guia compreensivo de implementações mais modernas pode ser encontrado em (DIAS; ANTUNES; MOTA, 2004).

A era da informação traz uma série de novas demandas: aumento no poder computacional bruto, computação ubíqua e inteligência artificial são alguns exemplos. Treinamento e inferência, os dois principais de algoritmos de redes neurais têm subproblemas como operações matriciais e cálculo de função de ativação, que apresentam um obstáculo significativo para processadores de propósito geral. Por essa razão, muito do processamento desses problemas já é realizado em hardware especializado que tira maior proveito de estruturas paralelas. Nessa classe de hardware, o estado da arte engloba as GPUs - *Graphical Processing Units* (OH; JUNG, 2004) e as TPUs - *Tensor Processing Units* (SUN; KIST, 2021). A primeira é projetada para processamento de gráficos e renderização 3D, que são tarefas de caráter paralelo, logo pode ser utilizada. TPUs, por sua vez, são dispositivos especialmente projetados para cálculos com tensores[1]

2 FUNDAMENTAÇÃO TEÓRICA

2.1 REDES NEURAIS

No contexto de redes neurais artificiais, muitos conceitos são usados erroneamente, especialmente quando há interesses de *marketing* envolvidos. Termos como Aprendizado de Máquina, Inteligência artificial são usados como sinônimos de redes neurais quando não o são. Se faz necessária distinção destes termos.

(GEITGEY, 2018) associa a ideia de Aprendizado de Máquina como um algoritmo genérico, que não é projetado para resolução de um problema específico, mas que consegue dizer algo interessante sobre um conjunto de dados de entrada, possivelmente com a ajuda de um processo chamado treinamento, em que o algoritmo aprende algum padrão. Inteligência artificial, por outro lado, é uma ideia muito mais ampla e abstrata, que envolve capacidade de deduções lógicas (ERTEL, 2018).

Rede neural artificial, doravante chamada somente de rede neural, é um modelo que assimila a organização da estrutura biológica nervosa de propagação de informação. Sua principal característica é a sua generalidade e adaptabilidade (HAYKIN, 2009). Uma rede neural pode ser treinada a detectar certos padrões e inferir informações a partir de uma entrada. Segundo a definição dada, o algoritmo de inferência pode ser considerado um algoritmo de Aprendizado de Máquina que adquire informação através do processo de treinamento.

2.1.1 Perceptron

O bloco fundamental das redes neurais é chamado de perceptron (figura 1). Essa estrutura emula o comportamento de um nerônio biológico ao promover a propagação de sinais de entrada v_i associados a pesos sinápticos w_i, que indicam a influência desse cada entrada na saída.

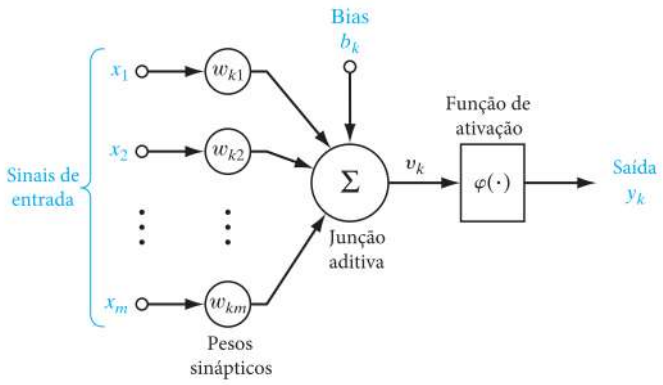


Figura 1 – Modelo de perceptron. Adaptado de (HAYKIN, 2009).

Em termos matemáticos, podemos definir v como combinação linear das entradas e pesos. Existe ainda um valor b, chamado de *bias*, que serve para deslocar o sinal da combinação linear, mudando o alcance da operação.

$$v = w_1x_1 + w_2x_2 + ... + w_nx_n + b \tag{2.1.1}$$

No entanto, admitir v como saída no neurônio traz um sério problema de divergência: seus valores crescem linearmente conforme o tamanho da entrada. Para corrigir tal situação, adiciona-se uma funcão de ativação, que limita a saída do dispositivo a um intervalo finito I, tipicamente $[0, 1]$ ou $[-1,1]$ (HAYKIN, 2009). Um outro problema é que a linearidade, inerente à equação 2.1.1 , é ineficiente para lidar com comportamentos complexos. Com efeito, o neurônio orgânico, o qual o perceptron imita, também se utiliza de mecanismos de propagação não lineares. (GEITGEY, 2018)

Usemos então uma função $\varphi : \mathbb{R} \to I$, chamada de função de ativação para tal efeito. Por fim, denotemos y como a saída do neurônio após função de ativação.

$$y = \varphi(v) \tag{2.1.2}$$

2.1.2 Inferência

O segundo nível de abstração que podemos adicionar é o de utilizar vários neurônios com diferentes pesos conforme a figura 2. O número de neurônios não necessita ser igual ao número de entradas como no exemplo. A essa estrutura é dado o nome de camada.

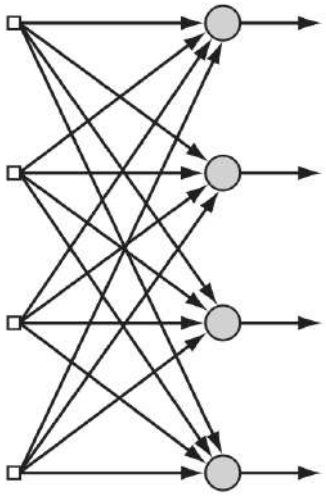


Figura 2 – Camada de uma rede neural. (HAYKIN, 2009)

Para as nossas intenções, é útil utilizar uma notação matricial para definição das operações de camada. Assim sendo, dada uma camada, podemos definir entrada X, combinação linear $V = W \times X$, saída Y e pesos sinápticos de todos os neurônios na camada W.

$$X = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \quad (2.1.3) \qquad V = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_m \end{bmatrix} \quad (2.1.4) \qquad Y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{bmatrix} \quad (2.1.5)$$

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix} \quad (2.1.6)$$

De agora em diante, vamos utilizar maiúsculas W, X, V, Φ e Y para referir às versões vetoriais e minúsculas para suas versões escalares definidas na subseção 2.1.1.

Observe que o vetor de bias pode ser embutido em W ao adicionar mais uma coluna $w_{i,n+1} = b_i$ à matriz e o elemento $x_{n+1} = 1$ ao vetor X. Vamos utilizar a matriz W como único depósito de pesos, ignorando o vetor de bias, dada a propriedade acima.

Por conseguinte,

$$V = W \times X \quad (2.1.7)$$

$$Y = \Phi(V) \quad (2.1.8)$$

Podemos então empregar uma camada como uma caixa preta, que dada uma entrada X, nos entrega uma saída Y (figura 3), conforme algum padrão de treinamento.

Figura 3 – Propagação direta

Numa rede neural direta multicamadas, as saídas de uma camada servem de entradas para a próxima camada, gerando padrões cada vez mais complexos. Por exemplo, em um fluxo típico de uma redes convolucional(GOODFELLOW; BENGIO; COURVILLE, 2017) encarregada de detectar rostos, uma camada poderia identificar bordas, outra formas, em seguida uma camada acusa a presença de características intricadas, como nariz e olhos.

Tomemos outro exemplo, o da figura 4. Ela consiste em um rede neural para solucionar um problema de classificação. Dado um ponto X, Y do plano como entrada, a rede deve dizer

se ele possui a cor azul ou laranja através de uma saída que pode variar de $[-1,1]$. As imagens em cada nó revelam a sua saída para cada ponto do plano. A saída do neurônio da última camada foi ampliado para melhor visualização. Observe que os padrões emergidos em cada uma das três camadas são gradativamente mais complexos.

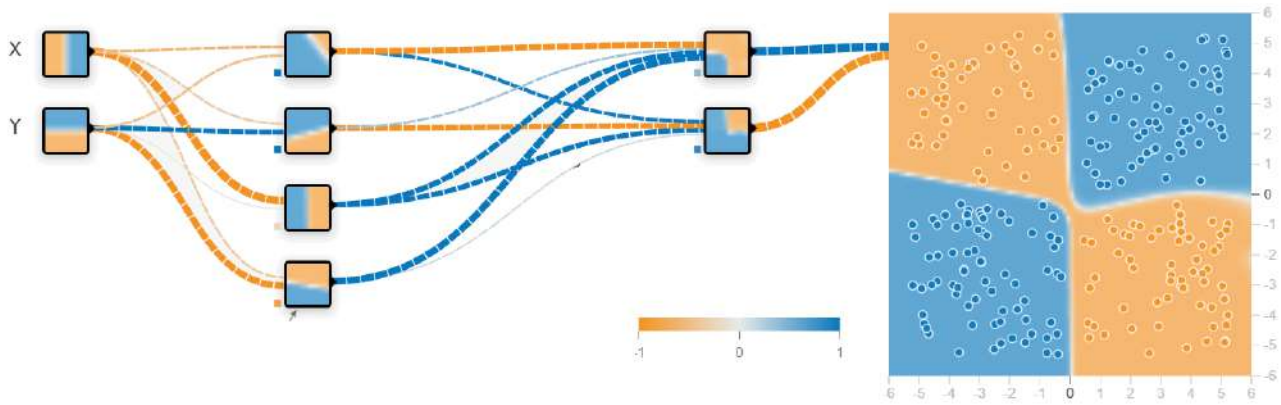


Figura 4 – Problema de classificação com rede multicamadas

Ao processo de obtenção da saída de uma rede neural a partir de sua entrada é dado o nome de **inferência**.

Existem ainda outras topologias de rede neural, como por exemplo as redes neurais recorrentes (HAYKIN, 2009), que não possuem uma topologia puramente direta. O estudo aprofundado de tais redes foge ao escopo desse trabalho.

2.1.3 Funções de ativação

Para os nossos propósitos, vamos classificar as funções de ativação em lineares e não-lineares. Funções lineares são particularmente simples de serem implementadas em hardware, enquanto não-lineares exigem técnicas mais intricadas (OLIVEIRA, 2017). Dentre as funções não-lineares, uma classe recebe atenção especial para redes neurais: as sigmóides.

2.1.3.1 Funções lineares

A essa categoria pertencem as funções que podem ser definidas por composições de funções lineares em seus subintervalos. Matematicamente,

$$\varphi(v) = a_i v + b_i, \text{ se } v \in D_i \tag{2.1.9}$$

donde

$$D_0 \cup D_1 \cup D_2 ... \cup D_n = \mathbb{R}$$

$$D_i \cap D_j = \emptyset, \forall i, j$$

Prontamente, a função linear (equação 2.1.10) é a função que aparece naturalmente dessa definição. Usualmente, se usa a função identidade, com $a = 1$ e $b = 0$, o que lhe dá a aparência disposta na figura 5.

$$\varphi(v) = av + b \tag{2.1.10}$$

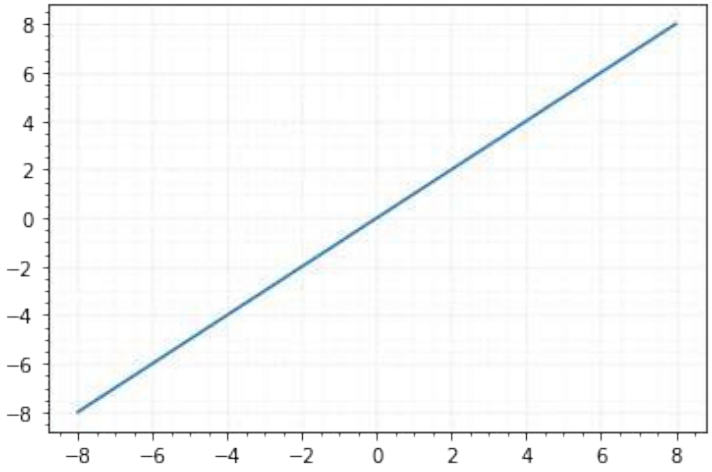


Figura 5 – Função Linear

Um outro tipo de função. de ativação linear é a Função Degrau de Heaviside, também chamada de função limiar. Definida como na equação 2.1.11, essa formulação descreve uma propriedade binária, isto é, o neurônio só pode se encontrar nos estados ativado ou desativado (HAYKIN, 2009). A sua aparência é tal como na figura 6

$$\varphi(v) = \begin{cases} 1, \text{ se } v \geq 0 \\ 0, \text{ se } v < 0 \end{cases} \tag{2.1.11}$$

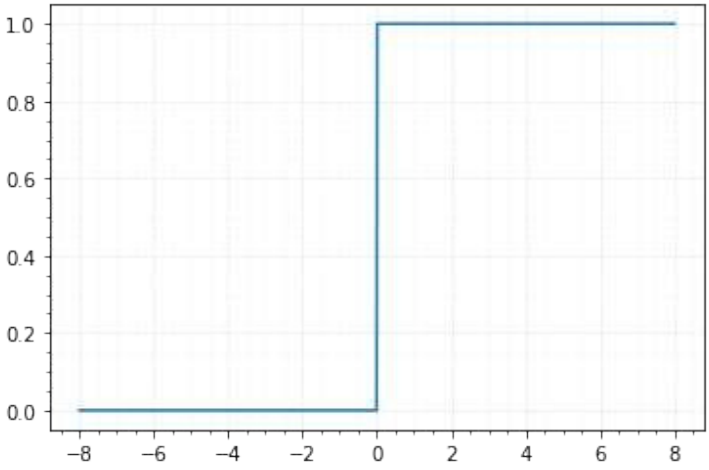


Figura 6 – Função Degrau

Um outro tipo comum de função de ativação é a Unidade linear retificada, ou ReLU - *Rectified Linear Unit*. Ela é um misto da função linear pura e da limiar (equação 2.1.12), o que a torna muito computacionalmente fácil de calcular e enquanto ainda apresenta propriedades interessantes (GOODFELLOW; BENGIO; COURVILLE, 2017). Observa-se que a classificação enquanto linear é pela definição aqui proposta na equação 2.1.9, grande parte da literatura não a classifica como linear (GOODFELLOW; BENGIO; COURVILLE, 2017). Sua aparência é a da figura 7.

$$\varphi(v) = \begin{cases} v, \text{ se } v \geq 0 \\ 0, \text{ se } v < 0 \end{cases} \tag{2.1.12}$$

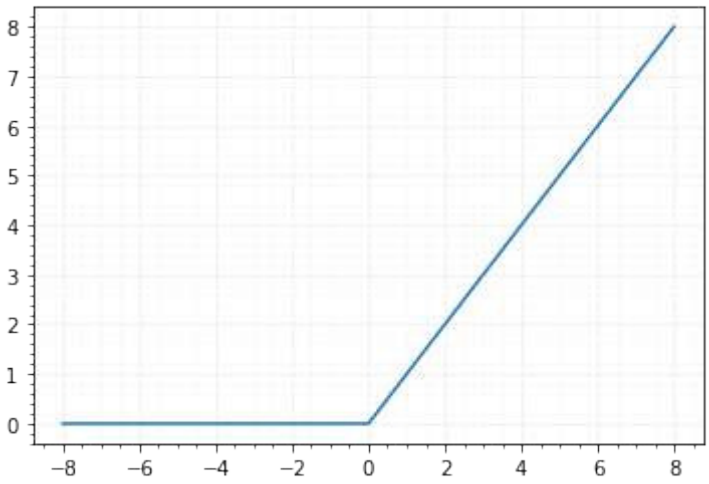


Figura 7 – Função ReLU

2.1.3.2 Sigmóides

Essa classe de função tem esse nome por seu gráfico ser em forma de S, sendo estritamente crescentes e exibindo um equilíbrio entre comportamento linear e não-linear (HAYKIN, 2009). O exemplo mais comum de função sigmoide é a função logística (equação 2.1.13. Seu formato em S é claramente visto na figura 8

$$\Phi(x) = \frac{1}{1 + e^{-x}} \tag{2.1.13}$$

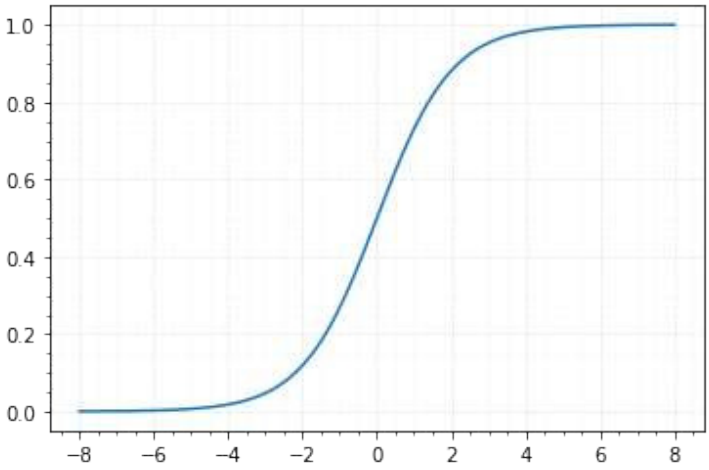


Figura 8 – Função Logística

A tangente hiperbólica é uma outra função muito utilizada como função de ativação em redes neurais. Possui um comportamento e definição algébrica (equação 2.1.14) muito próxima da função logística, possuindo uma imagem no intervalo $[-1,1]$ (figura 9).

$$\Phi(x) = \frac{1 - e^{-x}}{1 + e^{-x}} \tag{2.1.14}$$

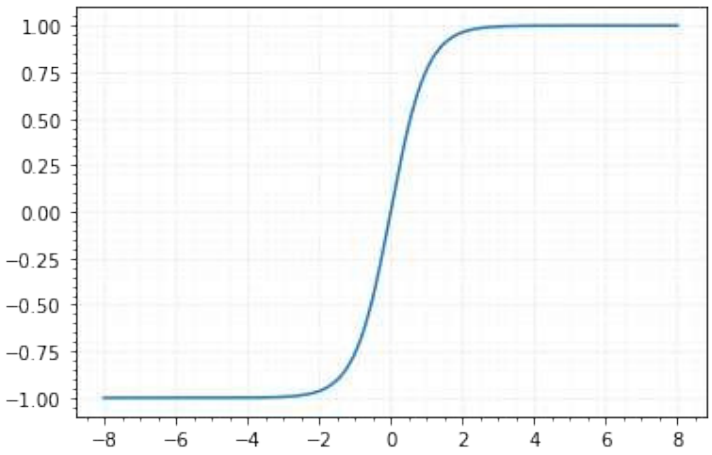


Figura 9 – Função Tangente Hiperbólica

2.1.4 Treinamento

Como citado na seção 2.1, redes neurais são algoritmos genéricos, de modo que é necessário algum tipo de treinamento a fim de adaptar a rede, isto é, seus pesos, a um problema específico. Por sorte, existe uma área dedicada da matemática dedicada a isto, a otimização numérica (GEITGEY, 2018). Existem várias técnicas para treinamento, alguns dos mais famosos sendo algoritmos genéricos (GOLBERG, 1989) e o gradiente descendente (HAYKIN, 2009). Aqui focaremos na última.

2.1.4.1 Gradiente Descendente

Para o gradiente descendente e alguns outros algoritmos de otimização, uma função de erro é uma função que visa quantificar o quão errado está uma aproximação de um valor de referência. Uma função de erro comum em redes neurais é o erro quadrático médio, ou MSE do inglês (Mean Square Error) (HAYKIN, 2009). Dado um vetor de referência $Y' = [y'_0\ y'_1\ \dots\ y'_n]$ e outro vetor palpite $Y = [y_0\ y_1\ \dots\ y_n]$ ambos de tamanho n define-se MSE como na equação 2.1.15.

$$E(Y, Y') = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - y'_i)^2 \tag{2.1.15}$$

Seja E uma função de erro dependente de parâmetros p_i, o algoritmo do gradiente descendente minimiza o valor de E com a atualização de cada parâmetro através da equação (2.1.17). O parâmetro η é uma constante denominada taxa de aprendizagem, que pode ser constante ou variar de acordo com a iteração. Esse parâmetro é extremamente relevante no processo de treinamento, e dele depende a convergência do algoritmo, que pode carecer de diversas iterações até a convergência. Valores pequenos de η podem tornar a otimização lenta ou estagná-la de vez. Do outro lado, valores grande demais podem provocar divergência (HAYKIN, 2009).

$$p \leftarrow p - \eta\frac{\partial E}{\partial p} \tag{2.1.16}$$

$$p \leftarrow p - \eta\frac{\partial E}{\partial p} \tag{2.1.17}$$

2.1.4.2 Propagação retroativa

Vejamos como podemos aplicar o algoritmo numa camada.

$$W \leftarrow W - \eta\frac{\partial E}{\partial W} \tag{2.1.18}$$

Expandindo

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial w_{1,1}} & \frac{\partial E}{\partial w_{1,2}} & \cdots & \frac{\partial E}{\partial w_{1,n}} \\ \frac{\partial E}{\partial w_{2,1}} & \frac{\partial E}{\partial w_{2,2}} & \cdots & \frac{\partial E}{\partial w_{2,n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial w_{m,1}} & \frac{\partial E}{\partial w_{m,2}} & \cdots & \frac{\partial E}{\partial w_{m,n}} \end{bmatrix}$$

Aplicando a regra da cadeia (GUIDORIZZI, 2000),

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial v_1}\frac{\partial v_1}{\partial w_{i,j}} + \frac{\partial E}{\partial v_2}\frac{\partial v_2}{\partial w_{i,j}} + \cdots + \frac{\partial E}{\partial v_m}\frac{\partial v_m}{\partial w_{i,j}}$$

Mas acontece que $v_i = w_{i,1}x_1 + w_{i,2}x_2 + \cdots w_{i,n}x_n + b_i$, de modo que $\frac{\partial v_n}{\partial w_{i,j}}$ só não é nulo quando $n = i$, logo

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial v_i}\frac{\partial v_i}{\partial w_{i,j}} = \frac{\partial E}{\partial v_i}x_j$$

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial v_1}x_1 & \frac{\partial E}{\partial v_1}x_2 & \cdots & \frac{\partial E}{\partial v_1}x_n & \frac{\partial E}{\partial v_1} \\ \frac{\partial E}{\partial v_2}x_1 & \frac{\partial E}{\partial v_2}x_2 & \cdots & \frac{\partial E}{\partial v_2}x_n & \frac{\partial E}{\partial v_2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{\partial E}{\partial v_m}x_1 & \frac{\partial E}{\partial v_m}x_2 & \cdots & \frac{\partial E}{\partial v_m}x_n & \frac{\partial E}{\partial v_m} \end{bmatrix}$$

$$\frac{\partial E}{\partial W} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \begin{bmatrix} \frac{\partial E}{\partial v_1} & \frac{\partial E}{\partial v_2} & \cdots & \frac{\partial E}{\partial v_m} \end{bmatrix}$$

$$\frac{\partial E}{\partial W} = X\left(\frac{\partial E}{\partial V}\right)^T \tag{2.1.19}$$

Mas agora observe que essa camada precisou do vetor $\frac{\partial E}{\partial V}$, que recebe da camada anterior. Com efeito, a camada seguinte também necessita do vetor $\frac{\partial E}{\partial V}$ análogo, de modo que a entrada da camada anterior é saída da camada atual (Figura 10). Deste modo, recíproco ao comportamento de propagação direta da figura 3. Por tal comportamento que algoritmo recebe o nome de propagação retroativa. É possível ver também o fluxo de propagagação do erro a partir de um nerônio individual.

Figura 10 – Propagação retroativa

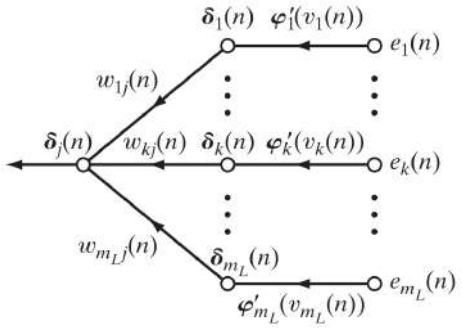


Figura 11 – Propagação retroativa vista por neurônio. Adaptado de (HAYKIN, 2009)

Assim sendo, precisamos também calcular ainda $\frac{\partial E}{\partial V^*}$, para isso, tomemos

$$\begin{aligned}\frac{\partial E}{\partial x_i} &= \frac{\partial E}{\partial v_1}\frac{\partial v_1}{\partial x_i} + \frac{\partial E}{\partial v_2}\frac{\partial v_2}{\partial x_i} + \cdots + \frac{\partial E}{\partial v_m}\frac{\partial v_m}{\partial x_i} \\ &= \frac{\partial E}{\partial v_1}w_{1,i} + \frac{\partial E}{\partial v_2}w_{2,i} + \cdots + \frac{\partial E}{\partial v_m}w_{m,i}\end{aligned}$$

De modo que

$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial v_1}w_{1,1} + \frac{\partial E}{\partial v_2}w_{2,1} + \cdots + \frac{\partial E}{\partial v_m}w_{m,1} \\ \frac{\partial E}{\partial v_1}w_{1,2} + \frac{\partial E}{\partial v_2}w_{2,2} + \cdots + \frac{\partial E}{\partial v_m}w_{m,2} \\ \vdots \\ \frac{\partial E}{\partial v_1}w_{1,n} + \frac{\partial E}{\partial v_2}w_{2,n} + \cdots + \frac{\partial E}{\partial v_m}w_{m,n} \end{bmatrix}$$

$$\frac{\partial E}{\partial X} = \begin{bmatrix} w_{1,1} & w_{2,1} & \cdots & w_{n,1} \\ w_{1,2} & w_{2,2} & \cdots & w_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,m} & w_{2,m} & \cdots & w_{n,m} \end{bmatrix} \begin{bmatrix} \frac{\partial E}{\partial v_1} \\ \frac{\partial E}{\partial v_2} \\ \vdots \\ \frac{\partial E}{\partial v_n} \end{bmatrix}$$

$$\frac{\partial E}{\partial X} = W^T\frac{\partial E}{\partial V} \tag{2.1.20}$$

Por fim, podemos determinar $\frac{\partial E}{\partial V}$ por

$$\frac{\partial E}{\partial V} = \frac{\partial E}{\partial Y}\cdot\frac{\partial Y}{\partial V}$$

$$\frac{\partial E}{\partial V} = \frac{\partial E}{\partial Y}\Phi'(V) \tag{2.1.21}$$

2.2 SISTEMAS DIGITAIS

Embora houvessem tentativas de construir computadores eletrônicos em outras bases de numeração, foi somente na base binária que eles realmente foram bem sucedidos e cresceram em escala. A superioridade da base binária não é de natureza matemática, mas sim natureza elétrica. Componentes que trabalham com somente dois estados oferecem uma imunidade maior ao ruído e possuem propriedades que garantem confiabilidade ao se trabalhar com ferramentas de EDA - *Electronic Design Automation* (RABAEY; CHANDRAKASAN; NIKOLIC, 2002).

O fato da menor complexidade de um bloco funcional permite também a abstração de propósitos. Transistores formam portas lógicas, portas lógicas compõem registradores e operadores aritméticos, esses últimos são capazes de intricados fluxos de dados. Cada uma dessas etapas pode ser pensada com certa independência das outras. Assim sendo, nesta seção veremos alguns tópicos de sistema digitais relevantes ao desenvolvimento do trabalho.

2.2.1 Sistemas de numeração

Sem perda de generalidade, um número inteiro não negativo b, pode ser representado de forma única pela equação 2.2.1, com $b_i \in [0,1]$. Por exemplo, adotando a notação subscrita para denotar a base e omitindo os coeficientes, $27 = 2^4 + 2^3 + 2^1 + 2^0 \Leftrightarrow 27_{10} = 11011_2$.

$$b = b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \cdots + b_1 2^1 + b_0 2^0 \tag{2.2.1}$$

Admitindo um sistema de representação finito, pode-se fixar um $N \geq \lceil log_2(b) \rceil$. Os coeficientes b_i são chamados bits, e b pode ser representando pela lista $[b_{N-1}, b_{N-2}, \ldots, b_0]$ ou $(b_{N-1}b_{N-2}\ldots b_0)_2$. Tal sistema de numeração de N bits representa os 2^N inteiros no intervalo $[0, 2^N - 1]$.

As principais técnicas de representação de números negativos são a utilização de um bit para indicação de sinal ou complemento de dois. A segunda é muito mais utilizada, uma vez que simplifica as operações de adição e subtração.

$$C2(b) = \begin{cases} b, \text{ se } b \geq 0 \\ 2^N + b, \text{ se } b < 0 \end{cases} \tag{2.2.2}$$

Da equação 2.2.1 é possível determinar sua inversa.

$$C2^{-1}(b) = \begin{cases} b, \text{ se } b < 2^{N-1} \\ b - 2^N, \text{ se } b \geq 2^{N-1} \end{cases} \tag{2.2.3}$$

Uma revisão da bibliografia de aritmética modular (COUTINHO, 1997) dá as ferramentas para provar a equação 2.2.4. Acontece que combinadas, as equações 2.2.3 e a 2.2.4 são suficientes para que as operações de adição e subtração sejam realizadas da maneira convencional e a qualquer momento se possa obter o valor original.

$$C2(a+b) = C2(a) + C2(b) \tag{2.2.4}$$

Um corolário da equação é que o bit mais significativo tem a propriedade de indicar o sinal do número, como se vê na tabela 1.

Inteiro	Complemento de dois	Binário
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Tabela 1 – Representação em complemento de dois para $N = 3$

Se faz conveniente também estender essa definição para incluir potencias de dois com expoentes negativos e assim aproximar números racionais. A representação da equação 2.2.5 é chamada de representação em ponto fixo. Por exemplo $6.25 = 2^2+2^1+2^{-2} \Leftrightarrow 6.25_{10} = 110.01_2$. Dados limites I e F, denota-se o sistema por $QI.F$. Tal sistema de numeração de $I + F$ bits representa os 2^N inteiros no intervalo $[0, 2^N - 1]$.

$$b = b_{I-1}2^{I-1} + \cdots + b_1 2^1 + b_0 2^0 + b_{-1}2^{-1} + \cdots + b_{-F}2^{-F} \tag{2.2.5}$$

A técnica de complemento de dois vale também para esta representação, como se pode ver na tabela 2.

Inteiro	Comp. de dois	Binário	Ponto Fixo	Inteiro	Comp. de dois	Binário	Ponto Fixo
0	0	0000	0.00	-8	8	1000	-2.00
1	1	0001	0.25	-7	9	1001	-1.75
2	2	0010	0.50	-6	10	1010	-1.50
3	3	0011	0.75	-5	11	1011	-1.25
4	4	0100	1.00	-4	12	1100	-1.00
5	5	0101	1.25	-3	13	1101	-0.75
6	6	0110	1.50	-2	14	1110	-0.50
7	7	0111	1.75	-1	15	1111	-0.25

Tabela 2 – Representação em complemento de dois para $Q = 2.2$

Existe também a representação em ponto flutuante, sendo sua versão mais famosa a sistematizada pelo padrão IEEE 754 (IEEE, 2019a). Semelhante à notação científica, a

representação em ponto flutuante reserva alguns bits para armazenamento de um expoente. Sendo M bits dedicados ao expoente e N à chamada mantissa, tem-se a relação da equação 2.2.6.

$$b = 2^e(1 + b_{N-1}2^{-1} + b_{N-2}2^{-2} \cdots + b_0 2^{-N}) \tag{2.2.6}$$

Com

$$e = b_{N+M-1}2^{N+M-1} + e_{N+M-2}2^{N+M-2} + \cdots + e_N 2^N$$

A quantidade de recursos de hardware para realização de operações com números flutuantes é substancialmente maior que para com ponto fixo. Com efeito, operações com pontos flutuantes exigem unidades especiais de processamento, as FPUs (*Floating Point Units*) (LIANG; TESSIER; MENCER, 2003).

2.2.2 Operações Aritméticas

O mais simples somador é o *Ripple Carry*, que possui a estrutura iterativa da figura 13. Cada um dos subcomponentes (Figura 12) mostrados realiza a soma de dois bits, recebendo um bit de carry, ou "vai um"em português, e enviando o seu carry para o próximo bloco, em uma estrutura idêntica ao algoritmo de soma convencional.

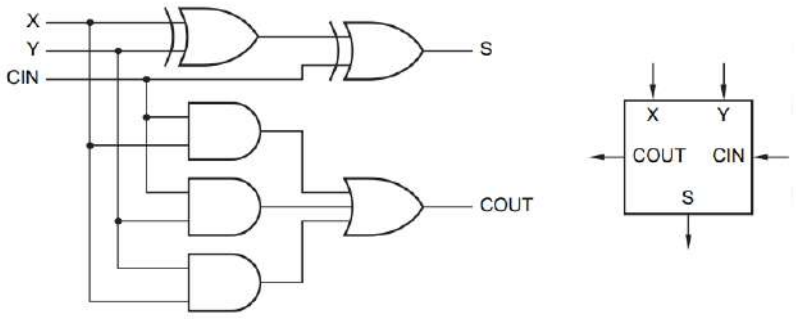


Figura 12 – Somador completo de 1 bit. Adaptado de (WAKERLY, 2008)

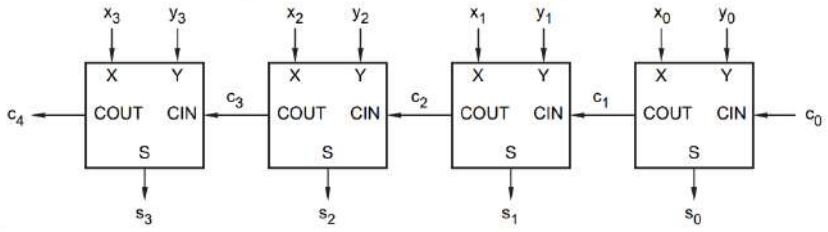


Figura 13 – Somador Ripple Carry. (WAKERLY, 2008)

Esse somador em específico possui um caminho crítico longo, de modo que foram criadas alternativas que mitigam esse problema. Existem muitas arquiteturas para somadores,

podemos citar Ripple Carry, Carry Skip, Carry Increment, Carry look-ahead, dentre muitas outras, cada uma possuindo um compromisso entre caminho crítico, potência e utilização de recursos (KAUR; SOOD, 2015).

Quando utilizando a notação de complemento de dois, subtratores conseguem reutilizar o hardware de adição substituindo a operação $a-b$ por $a+(-b)$. Para tanto, basta no diagrama da figura 13 que $x_i = a_i$, $y_i = \neg b_i$ e $C_0 = 1$ (WAKERLY, 2008).

Assim como para os somadores, existem diversas arquiteturas de multiplicadores. Multiplicação Ingênua, Karatsuba e Vedic são alguns exemplos (BRENT; ZIMMERMANN, 2010). A versão mais simples, por isso chamada ingênua é exposta na figura 14. Mais uma vez, esta arquitetura imita o algoritmo de multiplicação convencional, com n somadores de m bits para um multiplicador de $a[n-1:0] \times b[m-1:0]$.

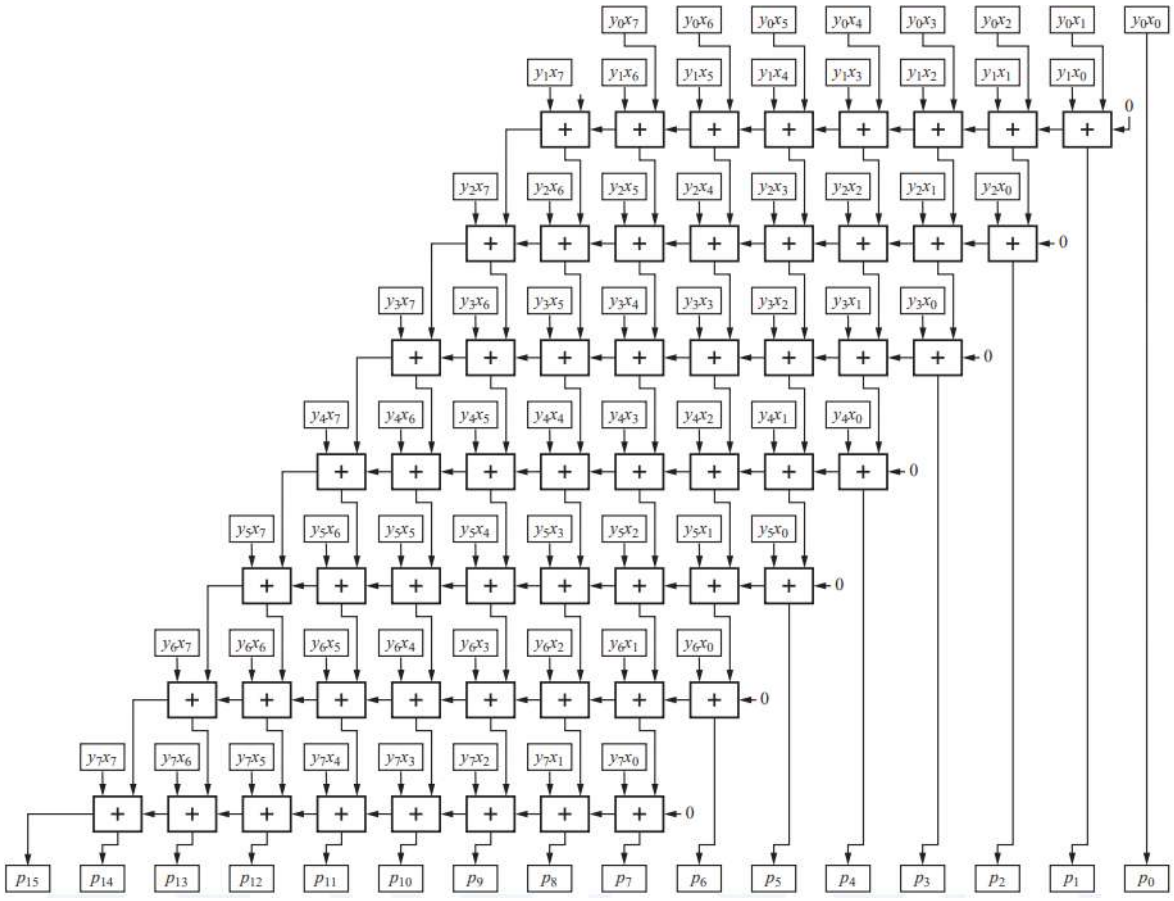


Figura 14 – Multiplicador ingênuo. (WAKERLY, 2008)

Recíproca da multiplicação, a divisão, é de fato a mais complicada e lenta operação básica. Uma exploração detalhada do tópico foge ao escopo do trabalho, (TYANEV; PETKOVA, 2018) oferece uma referência adicional para o leitor interessado.

As linguagens de descrição de hardware oferecem um nível de abstração a ponto de deixarem para a ferramenta de síntese lógica a escolha de qual arquitetura de operador será usada (STEFAN, 2016). Existem ainda modelos de FPGAs, contendo unidades DSP que englobam somadores e multiplicadores, que podem ser usados diretamente, ao invés de usar

blocos lógicos para sintetizar o operador (ALTERA, 2016).

2.2.3 Temporização

Circuitos assíncronos possuem uma série de problemas com respeito a determinação de tempos de propagação pelas ferramentas de síntese, de modo que o design de circuitos digitais a nível VLSI - *Very Large Scale INtegration* é quase todo feito usando técnicas síncronas (ALTERA, 2016). Assim sendo, síntese de loops combinacionais ou circuitos como ripple counter são desencorajados.

A medida que os circuitos digitais se miniaturizaram, cada vez mais o efeito da propagação dos sinais nos componentes se torna relevante. O efeito do delay entre entradas e saídas de portas lógicas, o tempo que a entrada de um flip flop precisa estar estável antes e depois da transição de clock e até mesmo o comprimento das trilhas de condutores devem ser levadas em conta (Figura 15). Para designs síncronos, a frequência máxima pode ser dada pelo maior tempo de propagação entre dois elementos sequenciais. O conjunto de vias e componentes que geram tal tempo é denominado de caminho crítico (ARORA, 2011).

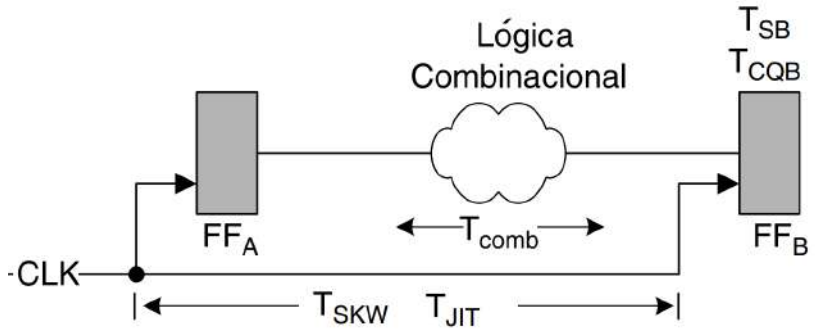


Figura 15 – Diagrama com as principais formas consumo de tempo. Adaptado de (ARORA, 2011)

Por (ARORA, 2011), a frequência máxima é dada por:

$$F_{Max} = (T_{Disponvel} + T_{Combinacional} + T_{Roteamento} + T_{Setup} + T_{Skew} + T_{Jitter})^{-1} \quad (2.2.7)$$

Onde

T_Disponível: tempo de transição de clock até a disponibilidade na saída do flip flop
T_Combinacional: é a propagação da lógica combinacional
T_Roteamento: o tempo devido a propagação em condutores
T_Setup: tempo que o sinal precisa estar estável antes da transição de clock
T_Skew: é a diferença de tempo que o clock leva para chegar em dois componentes

T_Jitter: é o tempo devido à variações de precisão na borda de transição, que afetam o período efetivo do clock. (Figura 16)

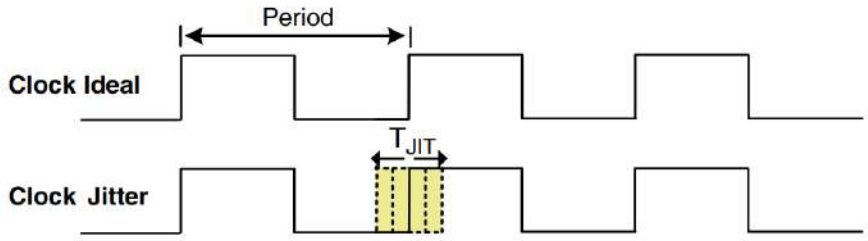


Figura 16 – Clock com Jitter. (ARORA, 2011)

O cálculo de F_{Max} é feito pelas ferramentas de EDA apropriadas. A falha em respeitar a restrição do tempo de propagação leva a uma condição chamada metastabilidade. Nessa condição, a saída de um flip flop é não determinística e não garantida de ocupar um dos dois estados lógicos e potencialmente pode se propagar aos sinais que dependem dele, gerando instabilidade generalizada. Embora existam técnicas para mitigar seus efeitos, quando não possa ser cancelado, é preemptivo de eliminação. (ARORA, 2011), (ALTERA, 2019).

2.2.3.1 Máquinas de Estado Finitas

Máquina de estados finita é um modelo matemático composto de um conjunto Q finito de estados, dos quais um é o inicial e um conjunto $T \supset Q^2$ de transições entre eles. Por brevidade, vamos nos referir a tais construtos por apenas máquinas de estados ou FSM, do inglês Finite State Machine. Um maneira de visualizar uma máquina de estados é através de um grafo direcionado, a qual chamamos de diagrama de estados (Figura 17). Poderíamos modelar por exemplo o problema do movimento de um elevador, sendo cada estado um andar e como condições as configurações de pressionamento de botões nos andares. Como técnica de projeto, cada estado adicionalmente entrega uma saída $s(q)$, $\forall q \in Q$. Ainda no exemplo anterior, a saída poderia ser um acionamento de motor para movimentação do elevador.

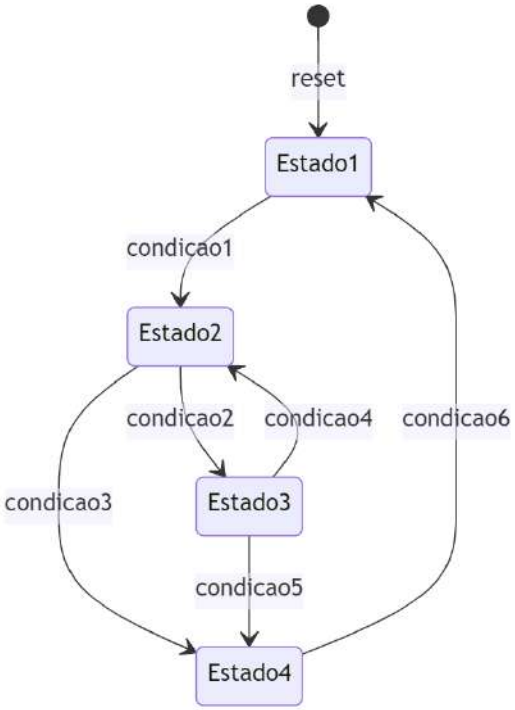


Figura 17 – Diagrama de estados

Em hardware digital, uma máquina de estados possui três componentes, um registrador de estados, um circuito lógico combinacional que retorna o próximo estado a ser armazenado e uma lógica de saída (Figura 18).A lógica de saída pode depender exclusivamente do estado atual, nesse caso sendo chamada de Máquina de Estados Moore, caso contrário é chamada de Mealy (PEDRONI, 2013). O primeiro tipo diminui o delay no caminho que leva a saída, no entanto, possui uma maior quantidade de estados, que se reflete em maior área e ciclos de clock.

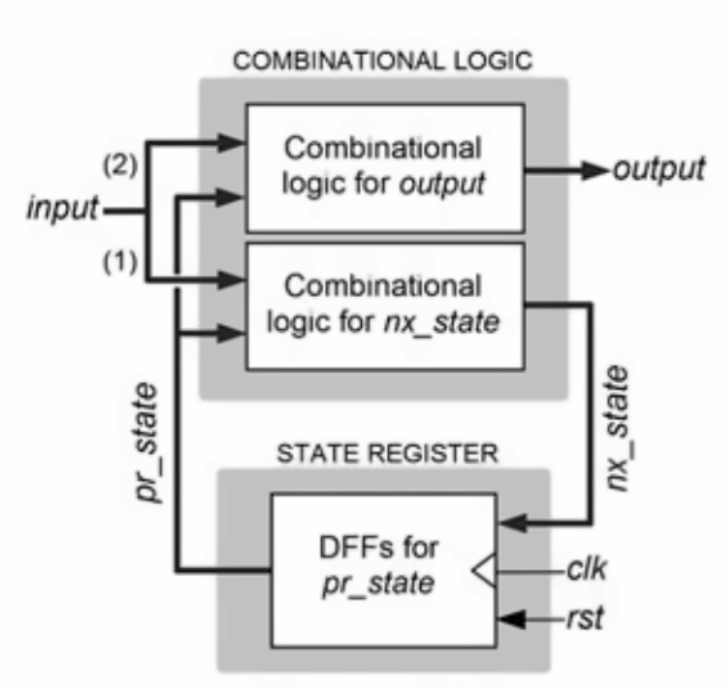


Figura 18 – Máquina de estados. (PEDRONI, 2013)

2.2.4 FPGAs

É acrônimo de Field Programmable Gate Array, em português Arranjo de Portas Programáveis em Campo. É uma arquitetura de dispositivos, introduzida na década de 1980 que veio a substituir tecnologias baseadas em fusíveis integrados (memória ROM) e outros de baixa capacidade (AMANO, 2018). A expressão programável em campo se refere a capacidade de mimetizar diferentes circuitos lógicos, podendo ser alterado a qualquer momento, característica que difere de circuitos integrados digitais tradicionais (CHU, 2012).

O bloco elementar do FPGA (Figura 19) é o bloco lógico, ou elemento lógico, nomenclatura essa que depende do fabricante. Um bloco dessa natureza possui, em essência, uma Look Up Table, que é uma estrutura de multiplexação para obter determinada saída a partir da memória; e um Flip-Flop, responsável pela execução de tarefas síncronas. Com efeito, esses blocos são capazes de gerar qualquer função que seriam possíveis com portas lógicas e flip-flops discretos (CHU, 2012).

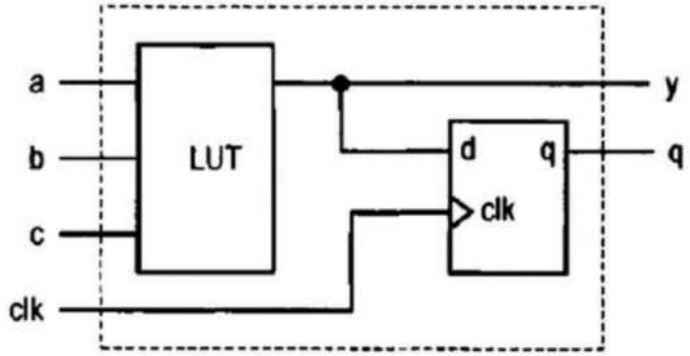


Figura 19 – Estrutura de elemento lógico simples. (CHU, 2012)

Os principais componentes de um FPGA são (AMANO, 2018)

- Elementos lógicos
- Elementos de entrada e saída, responsáveis por interfacear os circuitos internos e os externos, normalmente de níveis de tensão e impedância de saída incompatíveis
- Elementos de conexão (blocos de chaveamento e conexão e canal de interconexão) ligam os elementos lógicos para criar atribuições complexas. Segundo (INTEL, 2023), a maioria do custo de fabricação, consequentemente espaço em chip, de um FPGA se dá pelo roteamento.

Nas arquiteturas atuais, diversos elementos de uso específicos e de alta performance são adicionados: transceivers para comunicação com barramentos de altíssima velocidade, memória dedicada, PLLs (phase locked loops) para geração de clock arbitrário em frequência e fase, Conversores A/D, etc.

Após a aquisição de grandes empresas fabricantes de FPGA por parte de importantes fabricantes de processadores novos produtos voltados para aceleração de processamento em servidores vem surgindo no mercado. Esses novos produtos incorporam em um único chip um processador com set de instruções compatível com grande quantidade de programas comerciais e compiladores, junto com um FPGA capaz de trazer flexibilidade e paralelismo.

2.2.5 Técnicas de otimização de design

Existem técnicas para melhoria de potência, área e desempenho. Dada uma arquitetura, para uma dada tecnologia, a melhoria de um aspecto se dá em detrimento de outro. O aumento de desempenho se dá através do aumento da área e consequentemente potência. Reciprocamente, a diminuição da consumo de potência pode impactar a performance. Lista-se algumas dessas técnicas abaixo, focando-se em FPGAs.

2.2.5.1 Potência

- **Clock Gating:** Essa técnica consiste em inserir uma porta and entre o driver de clock e o destino em um grupo de flip flops (Figura 20). A inserção da porta permite que o clock

seja virtualmente desativado nos flip flops por um sinal, suprimindo o consumo dinâmico de potência (ALTERA, 2016).

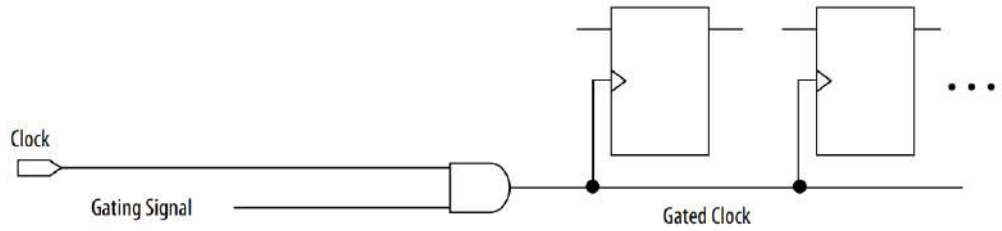


Figura 20 – Clock Gating. (ALTERA, 2016)

- **Clock Enable Síncrono:** tem o mesmo fim da anterior, diminuição de consumo dinâmico, no entanto, ela se utiliza de um multiplexador para admitir novos dados no flip flop (Figura 21). O benefício deste em relação à anterior é que é puramente síncrona, sendo mais eficiente na verificação de tempo, ao custo de uma menor diminuição de consumo (ALTERA, 2016).

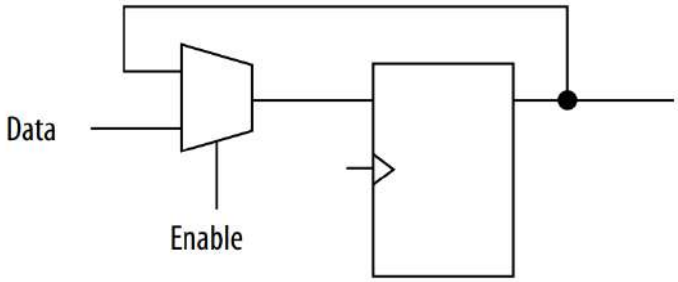


Figura 21 – Clock Enable Síncrono. (ALTERA, 2016)

- **Diminuição de área:** a potência é crescente com relação à área, portanto uma menor área reflete também na potência do design. Sendo assim, algumas técnicas de diminuição de área, como reaproveitamento de hardware, também resultam em diminuição de potência.

2.2.5.2 Desempenho

- **Paralelismo:** ao reorganizar estruturas sequenciais em puramente combinacionais ao invés de seriais (KILTS, 2007).
- **Pipeline:** é uma técnica que visa aumentar a frequência máxima de operação de um design ao inserir um ou mais flip flops em um caminho combinacional, criando caminhos adicionais, mas menores delays combinacionais, isto é, reduzindo o termo $T_{Combinacional}$ da equação 2.2.7. Caso a divisão do caminho seja homogênea, a aceleração de performance pode ser tão grande quanto o número de estágios de pipeline (HENNESSY; PATTERSON, 2017).

2.2.5.3 Área

- **Reaproveitamento harware:** exatamente o oposto do paralelismo, transformar estruturas paralelas em sequenciais, reaproveitando harware (KILTS, 2007).
- **Não uso de reset:** utilizar reset, especialmente em escopo global de FPGAs, pode evitar a inferência de componentes dedicados já disponíveis (KILTS, 2007).
- **Uso de recursos dedicados:** os FPGAs modernos contam com uma série de recursos como memórias, registradores de deslocamento e afins de modo muito mais eficiente que usando LUTs (KILTS, 2007).

3 MATERIAIS E METÓDOS

Os principais materiais usados neste trabalho foram as ferramentas de EDA, usadas para síntese, simulação e verificação do design; as linguagens de descrição de hardware, operando a caracterização do arquitetura; e as linguagens de programação, que comandaram as ferramentas de EDA e geraram a interface com o design e os casos de teste para verificação. A metodologia utilizada segue os passos abaixo:

1. Revisão de bibliográfica de arquiteturas e componentes existentes
2. Definição da arquitetura
3. Codificação em SystemVerilog
4. Elaboração de um modelo de referência para testes
5. Definição dos testes e métricas de verificação
6. Execução dos testes

3.1 FERRAMENTAS DE EDA

Hoje em dia, o projeto de circuitos, integrados ou discretos, está intimamente ligado às ferramentas de EDA, sigla para *Electronic Design Automation*, especialmente com a escala de integração cada vez maior. Simulação, layout, verificação são todas tarefas realizadas por esse tipo de software. A existência de chips VLSI (Very Large Scale Integration) com bilhões de transistores e a própria validade da Lei de Moore (MOORE, 1998) dependem do desenvolvimento das ferramentas de EDA (LAVAGNO et al., 2016).

Atualmente, as duas maiores empresas que fornecem ferramentas de EDA comercialmente são a Synopsys (SYNOPSYS, 2022), Cadence (CADENCE, 2022). Algumas outras empresas conseguem fatias como mercado com produtos focados em partes específicas do fluxo digital. Ferramentas para FPGAs são disponibilizadas pelos fabricantes, Quartus para a Intel (INTEL, 2022) e Vivado para Xilinx (XILINX, 2022), as duas maiores fabricantes de FPGA no mercado. Ferramentas Open Source vem cada vez tomando mais espaço no mercado, a dupla Icarus Verilog (ICARUS, 2022) para simulação e GTKWave (GTKWAVE, 2022) para visualização de formas de onda com grande presença na área. Já o OpenROAD é um projeto da University of California San Diego que visa a democratização da geração de layout digital para ASIC, do RTL ao GDS (OPENROAD, 2022).

Para nossos propósitos, é necessária uma ferramenta de síntese e simulação que seja gratuita para uso acadêmico. Foi escolhida a ferramenta QuestaSim (SIEMENS, 2022), da Siemens, pela presença de recursos como code coverage e suporte a properties do SystemVerilog, ausentes das ferramentas abertas. A ferramenta foi obtida em versão limitada com licença gratuita.

3.2 LINGUAGENS DE DESCRIÇÃO DE HARDWARE (HDLs)

Linguagens de descrição de hardware (HDLs) tem o objetivo de descrever univocamente um circuito. Diferentemente das linguagens de programação, que descrevem instruções sequenciais de execução para um tipo de processador, as linguagens de descrição de hardware descrevem as conexões entre componentes de um circuito, dentre eles flip flops e portas lógicas.

Atualmente as mais utilizadas são o VHDL (IEEE, 2019b), o Verilog (IEEE, 2006) e sua variante atualizada o SystemVerilog (IEEE, 2018). Na vanguarda das HDL temos o SystemC (IEEE, 2012) e Bluespec (NIKHIL, 2004), que buscam oferecer um fluxo de trabalho mais próximo dos paradigmas de codificação modernos.

Para esse trabalho, foi utilizada a linguagem SystemVerilog, robusta e já estabelecida na literatura. As linguagens mais recentes apresentam uma curva de aprendizado íngreme devido a seu pouco tempo de estabelecimento. Das linguagens consagradas o Verilog apresenta uma sintaxe muito menos prolixa que o VHDL, simplificando o esforço de codificação. O SystemVerilog espontaneamente apresenta uma série de melhorias em relação a seu predecessor, foi uma escolha natural para o projeto.

3.3 LINGUAGENS DE PROGRAMAÇÃO

3.3.1 Python

Python é uma linguagem de programação de proposito geral (PYTHON, 2022), foi criada por Guido Van Hossum com o objetivo de criar uma linguagem de programação muito parecida com inglês, otimizando a compreensão do código, e por conseguinte, a produtividade (LUTZ, 2014). É uma linguagem de alto nível, presuposto que se aproxima do inglês. É também fracamente tipada, isto é, por padrão não é exigido que operações e funções determinem tipos aceitos em tempo de declaração. O rendimento da linguagem a fez presente em diversas áreas: inteligência artificial e machine learning, cálculo numérico, automação de tarefas, dentre outras. Para machine learning e redes neurais, o estado da arte é o framework Tensorflow (TENSORFLOW, 2022). A produtividade da linguagem foi o fator preponderante na hora da sua escolha como linguagem usada nesse projeto, junto a presença de bibliotecas de computação numérica prontas para uso.

3.3.2 TCL

TCL é uma linguagem de programação interpretada para controle e extensão de aplicações. Essa linguagem foi criada com o objetivo de fornecer interfaces de comunicação em script para aplicações, daí seu nome Tool Command Language (OUSTERHOUT; JONES, 2009). Por esse motivo, TCL é o padrão de comandos e script com ferramentas de EDA, com a grande maioria das ferramentas comerciais a utilizando. Dentre essas ferramentas está o

Questasim, de maneira que a utilização do TCL se tornou imperativa na utilização de scripts para automatização de rotinas nesta plataforma.

4 ARQUITETURA

Grande parte do desafio da inferência em redes neurais é computação eficiente do produto entre matriz e vetor (equação 2.1.7).

Algoritmo 1: Produto Matriz x Vetor

Entrada: X, W
Saída: V
início
  $V[j] \leftarrow 0$
  $i \leftarrow 0$
  repita
    $j \leftarrow 0$
    repita
      $V[i] \leftarrow V[i] + X[j] \cdot W[i][j]$
      $j \leftarrow j + 1$
    até $j = n$;
    $i \leftarrow i + 1$
  até $i = n$;
fim

A cada iteração do laço de repetição, 4 operações são feitas: um produto, a soma no acumulador $v[j]$, a soma de índice e a verificação de saída laço. Sem nenhuma otimização de hardware ou software, pode-se estimar que tal algoritmo leve $4nm$ ciclos de clock para ser executado.

Todavia, esse algoritmo é facilmente paralelizável. Começando que o laço da variável i pode ser desenrolado em N_{cores} núcleos, reduzindo a constante m em N_{cores} vezes. As operações são também prontamente desenroladas. O produto e a soma podem ser implementados sequencialmente por hardware em um só ciclo. Em paralelo, um somador dedicado para o índice pode ser compartilhado por todas as unidades, reduzindo em mais um o número de ciclos.

As principais abordagens para implementação de redes neurais artificiais em hardware digital são através de neurônios e interconexões fixas ou por multiplexação de camadas (OLIVEIRA, 2017). Essa última implementa somente uma camada, que pode ser reutilizada nos diversos níveis de uma rede. Assim sendo, é muito mais flexível e gasta menos recursos que a primeira implementação para uma mesma dimensão de rede.

As implementações com multiplexação de camadas seguem uma estrutura comum e bem listada pela literatura (YOUSSEF; MOHAMMED; NASAR, 2014). Sem perda de generalidade, a arquitetura proposta por (ACOSTA; TOSINI, 2002) nos fornece um bom ponto de partida para implementação.

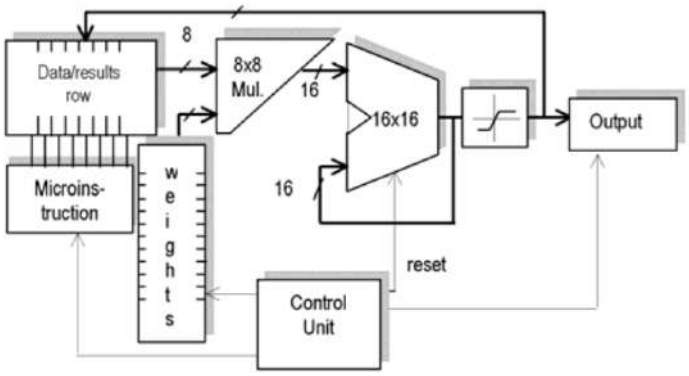


Figura 22 – Arquitetura da literatura. (ACOSTA; TOSINI, 2002)

Podemos modificá-la, inserindo um número arbitrário de núcleos de processamento de modo a alcançar a paralelização pretendida. Observe que alguns elementos foram duplicados, no entanto, outros podem ser compartilhados, como o módulo de função de ativação e uma memória de resultados.

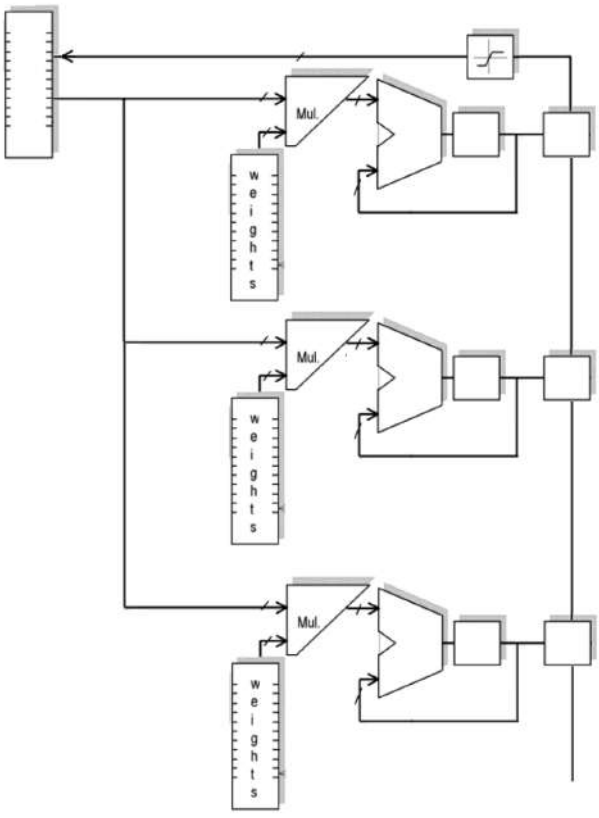


Figura 23 – Arquitetura modificada. Adaptado de (ACOSTA; TOSINI, 2002)

Utilizaremos então a arquitetura da figura 23 como base de implementação, inserindo algumas melhorias. Se faz saber que tal paradigma, de uso de unidades de processamento estando as conexões determinadas pela memória de instrução, é muito mais eficiente e flexível quando comparado a neurônios fixos com interconexões estáticas.

4.1 ARQUITETURA PROPOSTA

Propõe-se um arquitetura com três estágios de pipeline, conforme a figura 24, que exemplifica uma arquitetura com quatro núcleos de processamento. O pipeline carrega consigo também estruturas de *data forwading*. O módulo controlador foi omitido na figura a fim de reduzir a densidade de informações, ele se relaciona com todos os outros módulos. Foi escolhida a representação binária em ponto fixo, uma vez que é muito menos dispendiosa que a representação em ponto flutuante.

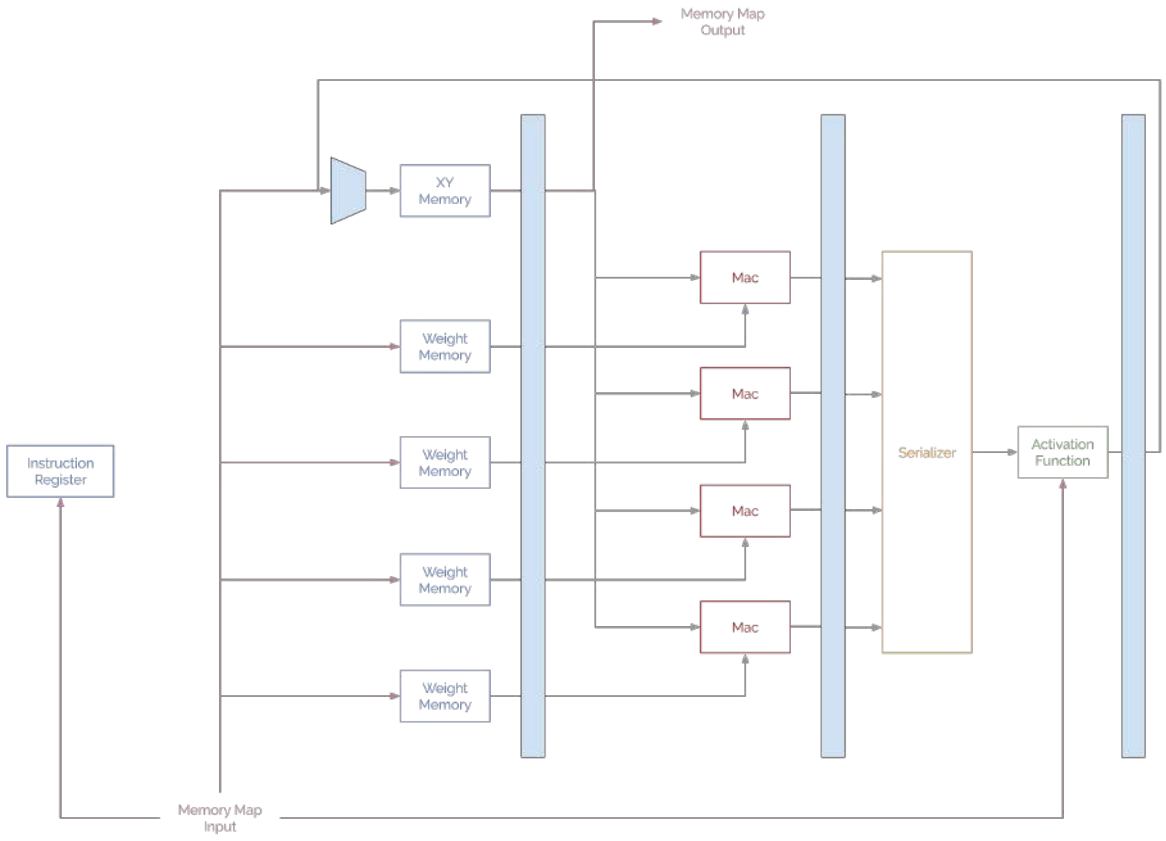


Figura 24 – Arquitetura proposta.

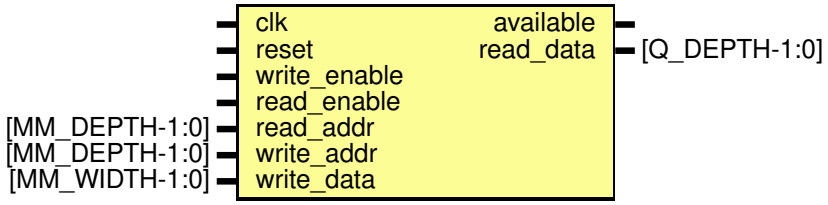


Figura 25 – Diagrama de bloco do acelerador de redes neurais

Podemos listar os parâmetros livres pelos quais a a arquitetura proposta pode ser customizada na tabela 4. A descrição do hardware em SystemVerilog deverá contê-los e o design instanciado conforme mudança nestes fatores.

Parâmetro	Descrição
N_{cores}	Quantidade de unidades MAC em paralelo
M_d	Profundidade do endereço de memória mapeada
M_w	Tamanho da palavra de dados da memória mapeada
I_d	Profundidade do endereço de memória de instruções
I_w	Tamanho da palavra de dados da memória de instruções
X_d	Profundidade do endereço de memória de entradas e saídas
W_d	Profundidade do endereço de memória de pesos
F_d	Profundidade do endereço de memória de funções
F_w	Tamanho da palavra de dados da memória de funções
Q_i	Tamanho da parte inteira da palavra aritmética
Q_f	Tamanho da parte fracionária da palavra aritmética
A_i	Tamanho da parte inteira do coeficiente a
A_f	Tamanho da parte fracionária do coeficiente a
B_i	Tamanho da parte inteira do coeficiente b
B_f	Tamanho da parte fracionária do coeficiente b

Tabela 3 – Parâmetros da arquitetura

E por completude, alguns outros parâmetros dependentes, que não podem ser alterados, uma vez que são função dos anteriores.

Parâmetros	Descrição	Definição
Q_l	Tamanho da palavra aritmética	$Q_i + Q_f$
F_w	Tamanho da palavra de dados da memória de funções	$A_l + B_l$
A_l	Tamanho do coeficiente a	$A_i + A_f$
B_l	Tamanho do coeficiente b	$B_i + B_f$

Tabela 4 – Parâmetros dependentes da arquitetura

Para fins de verificação e visualização, foi escolhido o conjunto de parâmetros da tabela 5. O FPGA 5CSXFC6D6F31C6Nda linha Cyclone V foi usado como referência para métricas dependentes de tecnologia.

Parâmetro	Valor
N_{cores}	4
M_d	16
M_w	16
I_d	9
I_w	64
X_d	11
W_d	12
F_d	6
F_w	32
Q_i	4
Q_f	12
A_i	4
A_f	12
B_i	4
B_f	12
Q_l	16
A_l	16
B_l	16

Tabela 5 – Parâmetros da arquitetura de teste

Por fim, a presença de uma memória de instruções permite que várias camadas sejam computadas, e cada instrução diz ao acelerador quais as propriedades da camada atual, como dimensões e sua localização nas memórias. Cada espaço na memória de instruções é decodificado conforme a tabela 6, caso uma instrução tenha o bit *reset* em nível 1, o dispositivo volta à instrução de posição 0 e espera uma nova requisição de atividade (Figura 26). Não confundir o bit *reset* do pacote de instrução com o bit *reset* global do módulo.

Variável	Fim	Início	Tamanho
reset	63	63	1
x_offset	62	52	11
w_offset	51	40	12
unused	39	39	1
y_offset	38	28	11
x_length	27	16	12
y_length	15	4	12
act_mask	3	0	4

Tabela 6 – Estrutura da instrução de camada

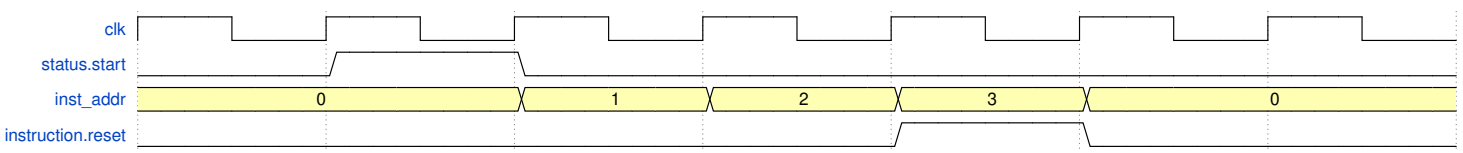


Figura 26 – Funcionamento do bit de reset

4.2 UNIDADE MAC

O módulo pode ser pensado como uma caixa preta segundo o diagrama de bloco da figura 27.

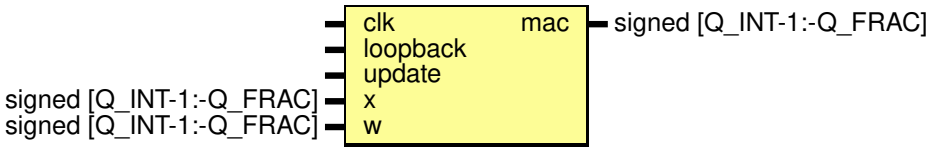


Figura 27 – Diagrama de bloco do MAC

A unidade MAC (Figura 28) é o componente responsável por realizar a operação da equação 4.2.1, cerne do produto $W \times V$ e do algoritmo 1.

$$Acc \leftarrow Acc + x \cdot w \tag{4.2.1}$$

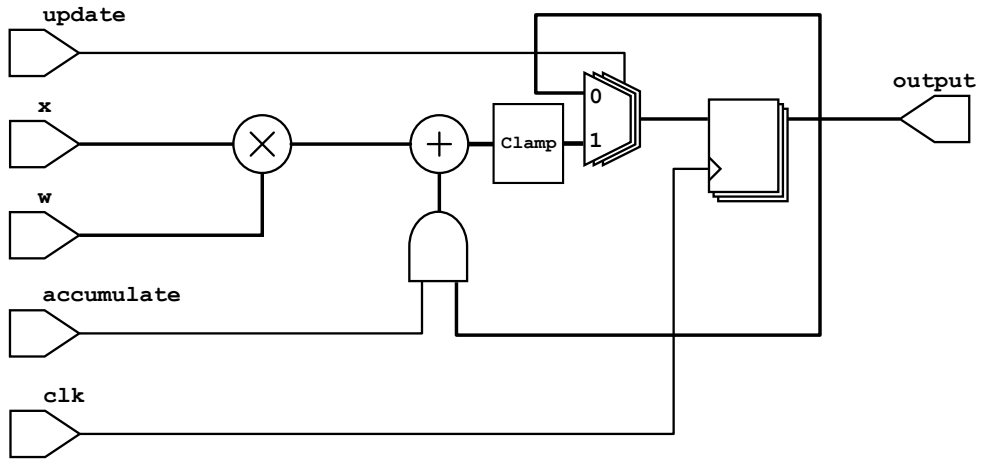


Figura 28 – Unidade Mac proposta

O seu funcionamento é tal como mostrado na tabela 7, de modo que no final das iterações, sua saída é a combinação linear exposta na equação 2.1.1.

iteração	update	accumulate	Operação
0	0	-	-
1	1	0	$Acc \leftarrow x_0 \cdot w_0$
2	1	1	$Acc \leftarrow Acc + x_1 \cdot w_1$
⋮	⋮	⋮	⋮
n-1	1	1	$Acc \leftarrow Acc + x_{n-1} \cdot w_{n-1}$

Tabela 7 – Funcionamento da unidade MAC

Em uma operação de adição, pode ocorrer do valor da soma de dois números ser maior que os valores suportados pelo tamaho da palavra aritmética (HENNESSY; PATTERSON, 2017). O submódulo *Clamp* garante que a soma será grampeada em um valor positivo máximo

ou negativo mínimo, evitando que a ocasião de overflow gere impecilhos maiores nos cálculos. Por sorte, condições de overflow podem ser naturalmente detectáveis examinando as paridades dos operandos e da saída da soma. Sendo $s = a + b$, ambos números em complemento de dois e ponto fixo $Q_i.Q_f$.

$a[Q_i]$	$b[Q_i]$	$s[Q_i]$	Overflow Positivo	Overflow Negativo	Saída
0	0	0	não	não	s
0	0	1	sim	não	$2^{Q_i-1} + 2^{Q_i-2} + \cdots + 2^{Q_f}$
0	1	0	não	não	s
0	1	1	não	não	s
1	0	0	não	não	s
1	0	1	não	não	s
1	1	0	não	sim	-2^{Q_i}
1	1	1	não	não	s

Tabela 8 – Condições de Overflow e saída do submódulo de Clamp. Adaptado de (HENNESSY; PATTERSON, 2017)

4.3 MEMÓRIAS

Operacionalmente, podemos pensar uma memória como um vetor $M = [m_0, m_1, \ldots, m_n]$, com as operações de escrita $M[write\_addr] \leftarrow v$ e leitura $v \leftarrow M[read\_addr]$. Assim sendo, uma memória precisa das portas na tabela 9.

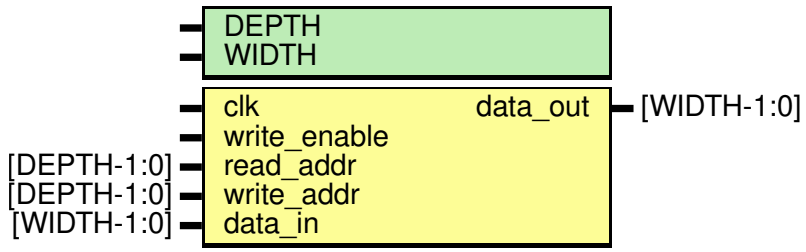


Figura 29 – Diagrama de bloco da memória

Sinal	Tamanho	Entrada/Saída	Descrição
clk	1	Entrada	Clock
write_enable	1	Entrada	Habilita a escrita
read_addr	READ_DEPTH	Entrada	Endereço de escrita
write_addr	WRITE_DEPTH	Entrada	Endereço de leitura
data_out	READ_WIDTH	Saída	Saída de dados
data_in	WRITE_WIDTH	Entrada	Entrada de dados

Tabela 9 – Portas das memórias

Foram implementados dois tipos de memória. O primeiro tipo, é uma memória simples de duas portas para cada uma das memórias de pesos (*w_mem*) e a memória de entrada e saída (*xy_mem*). Essa memória é tal que $READ\_DEPTH = WRITE\_DEPTH$

e $READ\_WIDTH = WRITE\_WIDTH$. Um problema que surge naturalmente de tal restrição é o barramento de entrada cresce em tamanho com o de saída. Para evitar isto, uma variante da memória com diferentes tamanhos dos barramentos de entrada e saída foi utilizada para as memórias de instrução (*inst_mem*) e a LUT do módulo de função de ativação (*act_mem*). Tal façanha permite que por exemplo tenhamos uma instrução de 64 bits mesmo mantendo o barramento de entrada em 16 bits. A estratégia de implementação foi o uso de várias memórias do primeiro tipo, chamadas de bancos (Figura 52). Por conseguinte, suas dimensões são tais que

$$WRITE\_DEPTH = READ\_DEPTH + \lceil log_2(BANKS) \rceil$$

$$WRITE\_WIDTH = \frac{READ\_WIDTH}{BANKS}$$

Em que $BANKS$ é a quantidade de blocos de memória da primeira variante utilizados, na figura de exemplo 52, tem-se $BANKS = 4$.

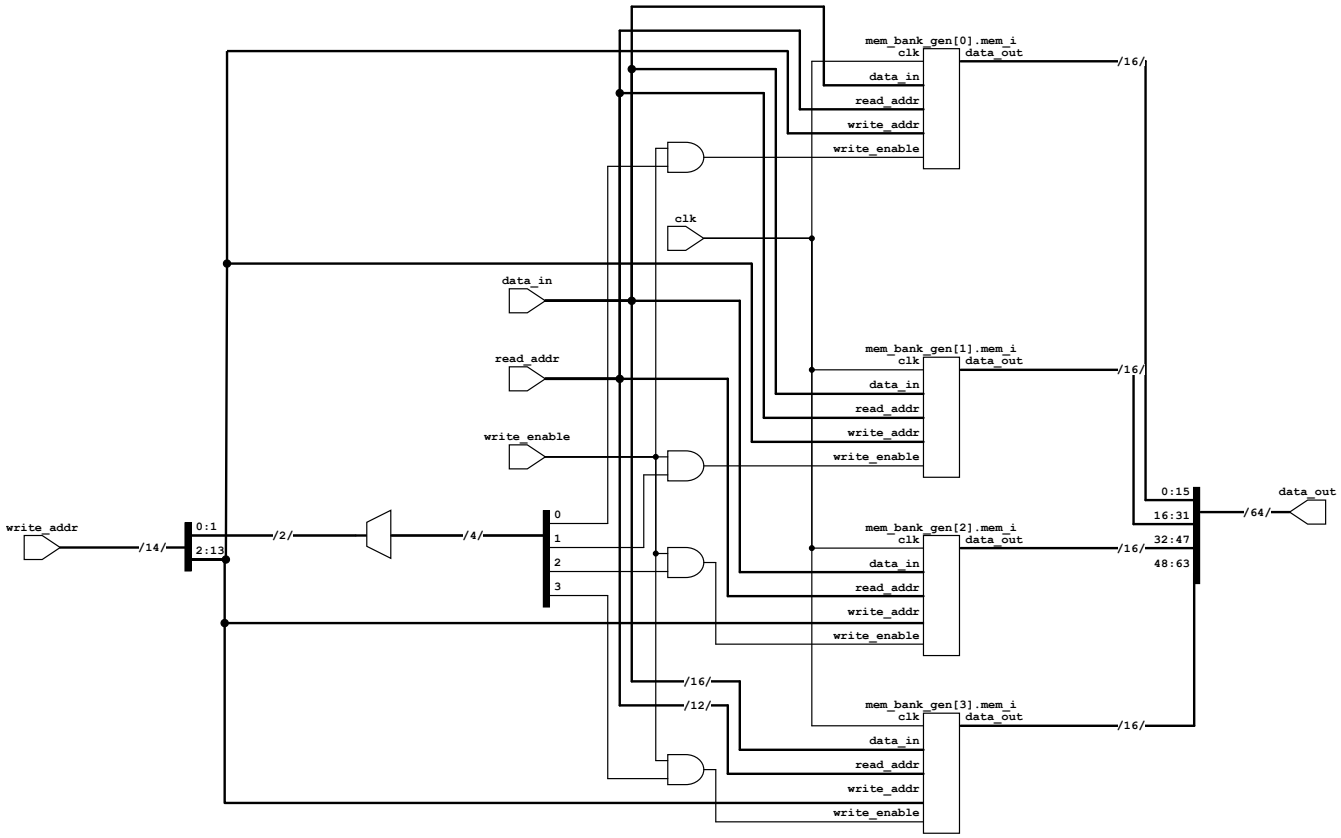


Figura 30 – Diagrama interno do banco de memórias

4.4 SERIALIZADOR

O módulo pode ser pensado como uma caixa preta segundo o diagrama de bloco da figura 31.

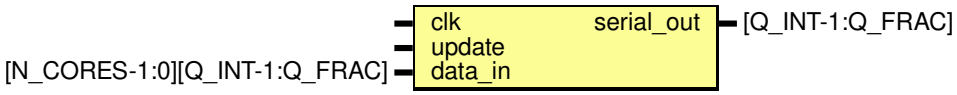


Figura 31 – Interface do módulo serializador

O módulo serializador é tão somente um registrador de deslocamento e um multiplexador que permite o data forwarding (Figura 32).

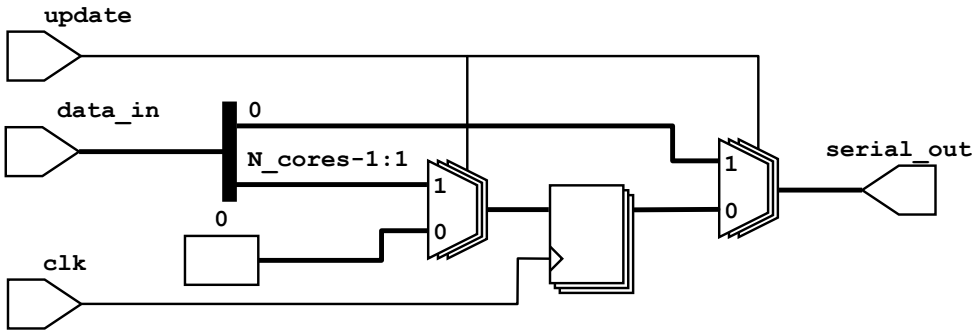


Figura 32 – Diagrama interno do serializador

Sua forma de onda (Figura 33)é similar a de um registrador de deslocamento comum, no entanto, possui uma latência menor, o que beneficia o pipeline.

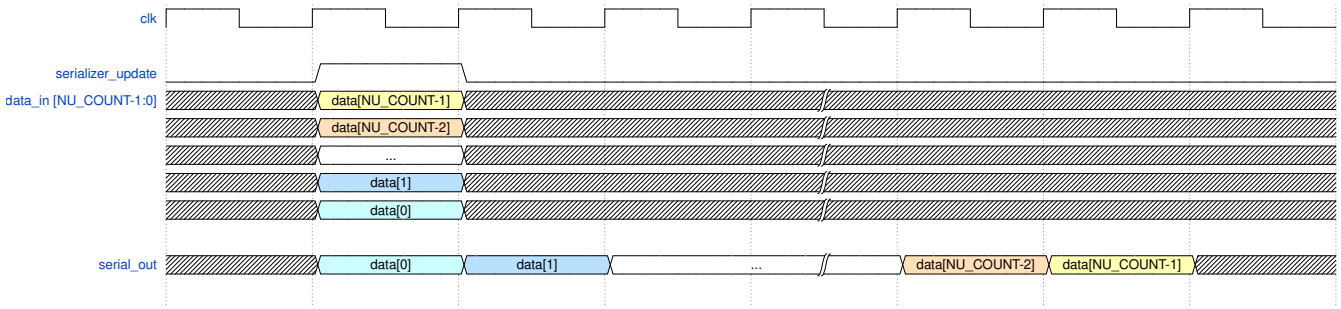


Figura 33 – Forma de onda do serializador

4.5 FUNÇÃO DE ATIVAÇÃO

Em princípio, o módulo função de ativação necessita somente de uma entrada numérica x para retornar uma saída $f(x)$. No entanto, para maior flexibilidade, uma entrada *mask* permite que sejam selecionadas dentre vários tipos de função de ativação, selecionadas por multiplexadores (Tabela 10 e Figura 34). Mesmo que possuam aplicações específicas, como a ReLU especialmente adequada para redes profundas (GOODFELLOW; BENGIO; COURVILLE, 2017), é por flexibilidade que foram escolhidas tais funções de ativação.

Máscara	$f(x)$
00	x
01	$Step(x)$
10	$ReLU(x)$
11	$LUT(x)$

Tabela 10 – Máscara e funções de ativação

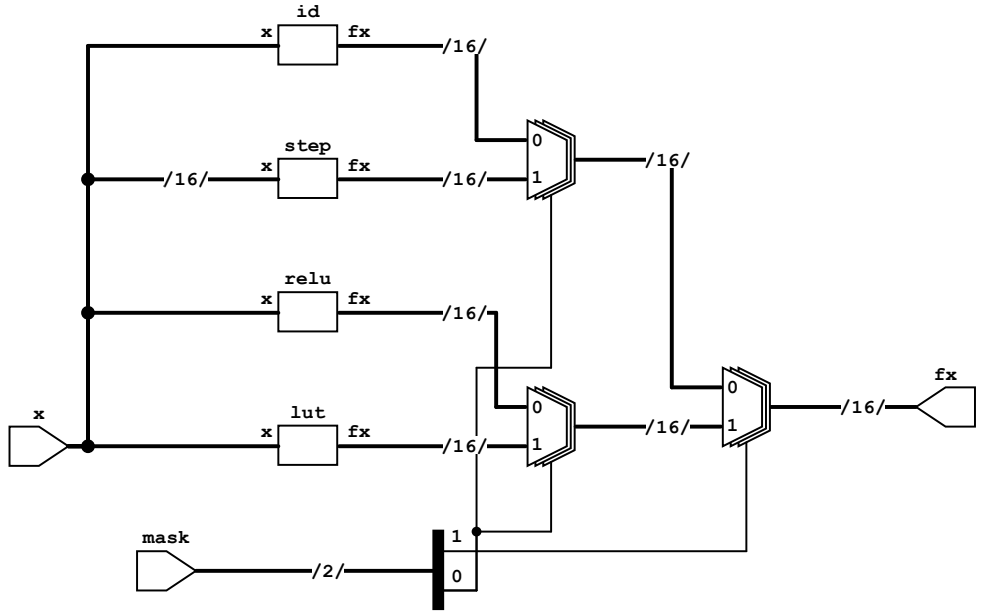


Figura 34 – Diagrama interno do módulo função de ativação

Do ponto de vista de caixa preta, o módulo pode ser visto como na figura 35. Os sinais *write_addr* e *write_data* cumprem um papel especial para LUTs internas.

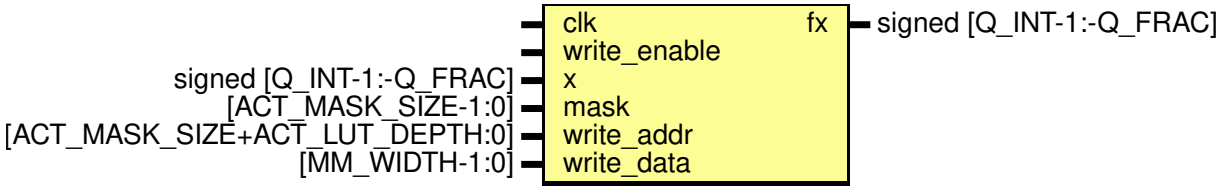


Figura 35 – Diagrama de bloco da função de ativação

4.5.0.1 Funções lineares

Seja b um número binário em complemento de dois e ponto fixo $Q_i.Q_f$, e $b_i, Q_i \geq i \geq Q_f$ seus dígitos e $y = f(b)$ a função de ativação. Função lineares são facilmente implementadas por hardware. A implementação da função identidade é trivial: a repetição da entrada. Para as funções Degrau e ReLU, pode-se usar o bit de sinal b_{N-1} como sinal de controle.

A implementação da função degrau pode ser simplificada como

$$\begin{cases} y_0 = \neg b_{N-1} \\ y_i = 0, \forall i \neq 0 \end{cases} \tag{4.5.1}$$

uma vez que

$$b \geq 0 \Rightarrow b_{N-1} = 0 \Rightarrow \neg b_{N-1} = 1$$
$$\Rightarrow y_0 = 1$$
$$\Rightarrow y = 2^0 = 1$$

e

$$b < 0 \Rightarrow b_{N-1} = 1 \Rightarrow \neg b_{N-1} = 0$$
$$\Rightarrow y_i = 0$$
$$\Rightarrow y = 0$$

Para uma arquitetura $Q4.12$, o circuito é tal como na figura 36.

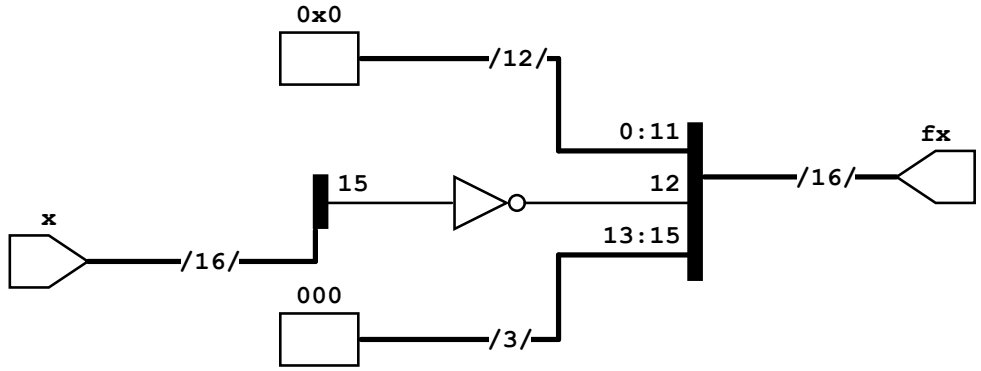


Figura 36 – Diagrama interno da função de ativação degrau

Para ReLU,

$$y_i = \neg b_{N-1} \wedge x_i \tag{4.5.2}$$

que se equivale à equação 2.1.12 por

$$b \geq 0 \Rightarrow b_{N-1} = 0 \Rightarrow \neg b_{N-1} = 1$$
$$\Rightarrow y_i = 1 \wedge x_i = x_i$$
$$\Rightarrow y = x$$

e

$$b < 0 \Rightarrow b_{N-1} = 1 \Rightarrow \neg b_{N-1} = 0$$

$$\Rightarrow y_i = 0 \wedge x_i = 0$$

$$\Rightarrow y = 0$$

Para uma arquitetura $Q4.12$, o circuito é tal como na figura 37.

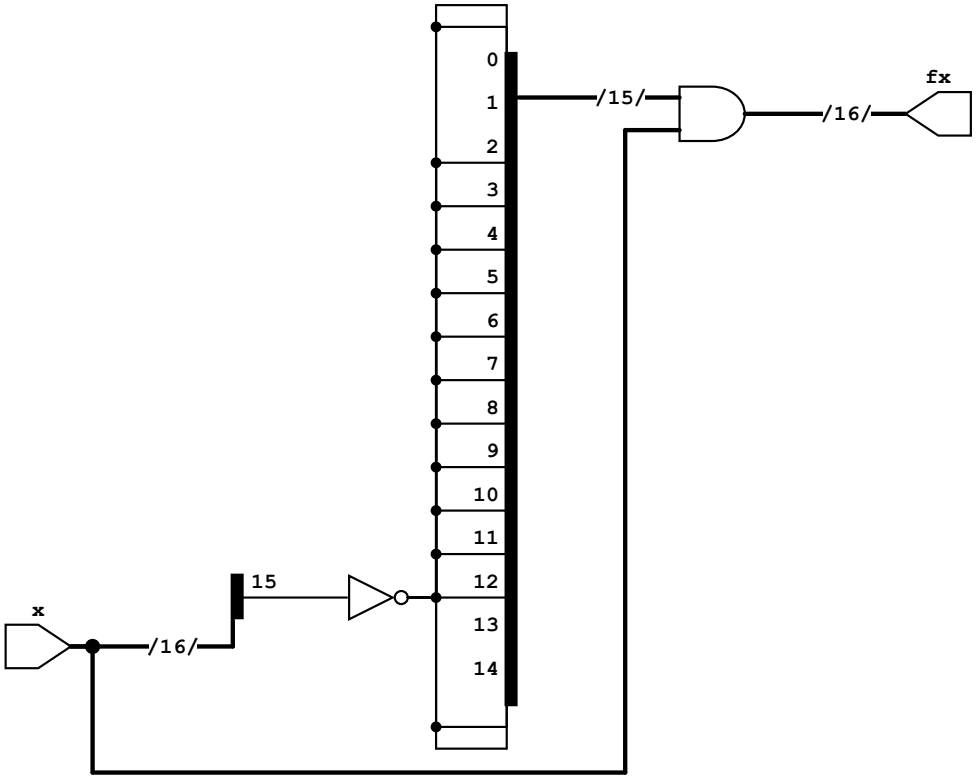


Figura 37 – Diagrama interno da função de ativação ReLU

4.5.0.2 Funções não lineares

A quantidade de recursos usada para implementação de operações matemáticas mais complicadas cresce bastante com o a complexidade da função resultante. Pensando nisso, uma técnica muito utilizada para implementação de funções matemáticas mais complexas é lidando com LUTs (*Lookup Tables*). Essa técnica consiste em armazenar valores chave em uma memória e realizar operações mais complexas a partir disso. Um célebre algoritmo que usa LUT é o CORDIC (VOLDER, 1959), e o faz para calcular funções trigonométricas.

Uma aproximação particularmente simples, mas eficiente é a interpolação por funções lineares. Para tanto, podemos simplesmente armazenar em memória os coeficientes a_i e b_i da equação 2.1.9 e uma maneira de definir os intervalos disjuntos I_i. Esta última tarefa pode ser alcançada armazenando os limites de cada intervalo. Uma alternativa de muito mais fácil implementação é no entanto, fazer os limites constantes e iguais a potências de dois, de modo que uma simples checagem de igualdade entre os bits mais significativos atesta o pertencimento a um intervalo. Uma outra melhoria que se pode fazer é o uso de uma máscara a fim de dividir a memória em subregiões, aumentando a quantidade de lookup tables armazenadas. Em sumário, o design é tal como apresentado na figura 38.

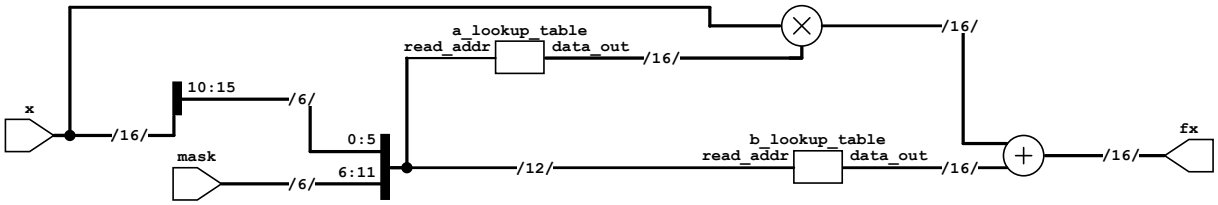


Figura 38 – Diagrama interno da função de ativação por aproximação linear e LUTs

Vários parâmetros interferem no erro de aproximação por interpolação linear em LUT : a quantidade de intervalos (Figura 39), a precisão da memória e do próprio comportamento da função aproximada. Decidiu-se manter o mesmo tamanho de palavra $Q4.12$ para os coeficientes a e b, e uma quantidade de intervalos de 64.

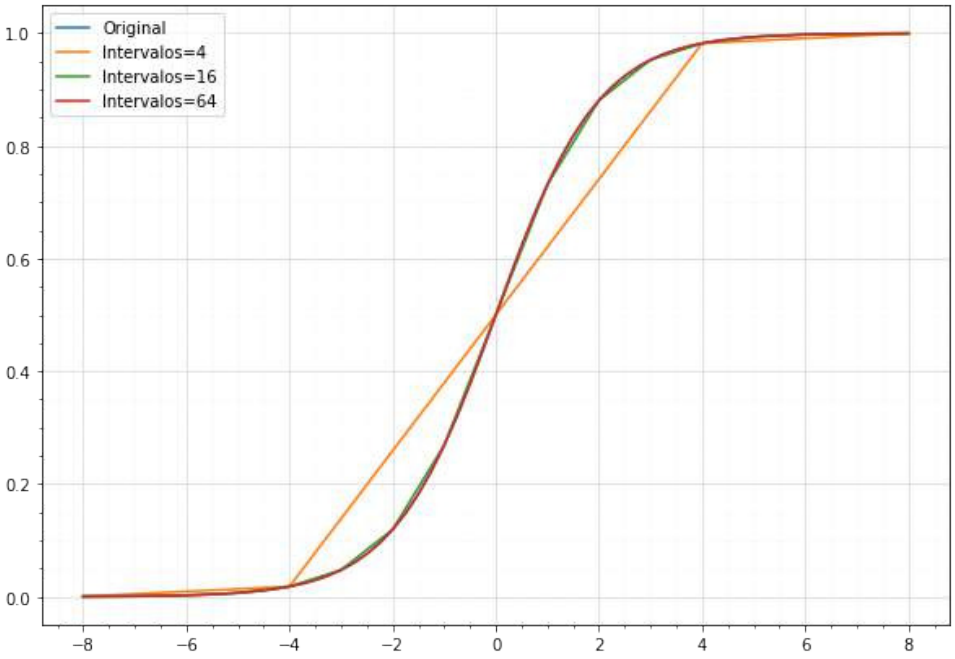


Figura 39 – Interpolação da Função sigmóide para diferentes intervalos

4.6 CONTROLADOR

Este módulo é responsável pela condução de todos os sinais de controle do caminho de dados do design. Ele possui como entrada uma instrução que codifica uma camada da rede, advinda da memória de instruções, além dos sinais de reset e start (Figura 40).

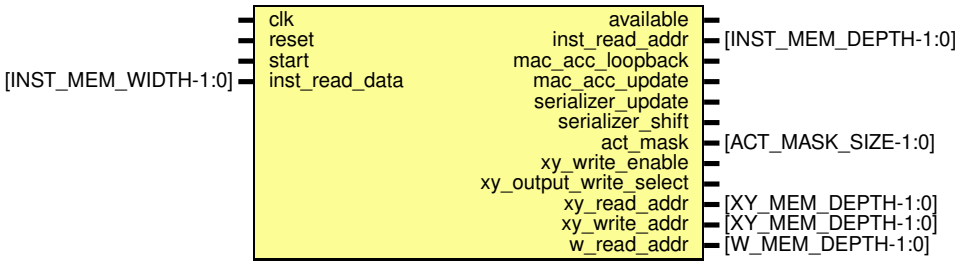


Figura 40 – Diagrama de bloco do controlador

O seu design foi baseado em uma FSM do tipo Mealy, ilustrada na figura 41. Pela sua complexidade, as condições de transição foram ocultas.

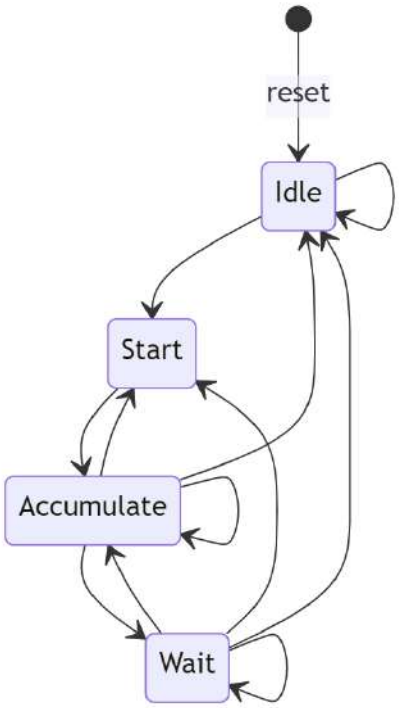


Figura 41 – Máquina de estados do controlador

Cada estado desempenha um papel no fluxo de cálculo como segue, acionando os sinais de saída correspondentes.

- **Idle:** espera até que o sinal *start* do registrador status seja acionado para iniciar os trabalhos. O sinal em questão é o bit menos significativo do registrador *status_reg*, mencionado na tabela 11.
- **Start:** inicializa os registradores internos com os dados de camada recebidos pela instrução
- **Accumulate:** realiza a operação de acumulação nos MACs
- **Wait:** caso o serializador esteja ocupado, espera até que esteja disponível para enviar os resultados dos MACs

4.7 MAPA DE MEMÓRIA

Um dispositivo de mapeamento em memória é um componente que emula uma memória para a interface externa de jeito que a escrita e leitura em determinada região de memória corresponde à escrita em um componente específico. Tomemos como exemplo o mapa de memória adotado na arquitetura (Tabela 11), ler no endereço 0x0000, 0x0001, ..., 0x1FFF faz com que o valor seja repassado a memória xy_mem, da mesma forma, escrever no endereço 0xE000, 0xE001, 0xE002, ..., 0xEFFF faz com que o dados sejam escritos na memória de instruções. Nota-se que da maneira que a escolha dos valores torna fácil a lógica de seleção, basta atentar-se para os primeiros quatro bits do endereço.

Memória	Offset de endereço
xy_mem	0x0000
act_mem	0x2000
w_mem	0x4000
status_reg	0xC000
inst_mem	0xE000

Tabela 11 – Mapa de memória

A leitura (Figura 42) e escrita (Figura 43) funcionam da mesma forma como funcionariam para uma memória convencional. As formas de onda apresentadas representam os sinais da arquitetura com a nomenclatura da figura 25.

Figura 42 – Forma de onda para leitura de dados

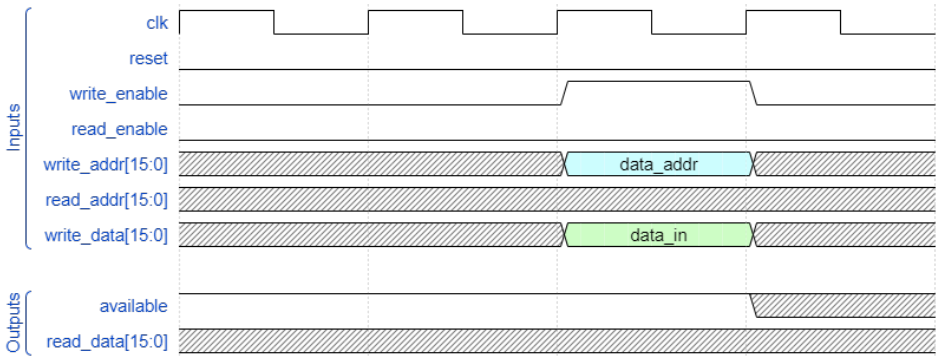


Figura 43 – Forma de onda para escrita de dados

5 VERIFICAÇÃO

A maior parte do tempo desenvolvimento é gasto com verificação, com efeito, estima-se que 20% do tempo seja gasto com RTL e os 80% restante com verificação (VERRYCKEN, 2020). Não é a toa, o processo de fabricação de um circuito digital pode custar milhões, e um erro pode ser o suficiente para inutilizar um lote inteiro e obrigar o gasto de mais alguns milhões para refazer o fluxo para reparar o erro.

Hoje em dia existem várias técnicas de verificação, aperfeiçoadas ao longo dos anos, verificação formal e funcional, dentro desta última ainda temos metodologias com várias camadas de abstração como UVM, VMM, OVM (CERNY et al., 2015). Aqui optamos pela verificação funcional utilizando as funções nativas da linguagem pela relativa falta de complexidade quando comparada com as outras alternativas.

5.1 MODELO DE REFERÊNCIA

No contexto da verificação de hardware digital, o Golden Model, também chamado de Golden Reference ou afins, é uma referência de comportamento para o design a ser testado que se comunica com o ambiente de verificação (CERNY et al., 2015). No nosso caso, precisa-se de uma peça de software capaz de realizar inferência, as principais alternativas são:

- **Em código Verilog:** a linguagem possui as estruturas de uma linguagem de programação convecional (condicionais, loops, vetores, ...) para implementação de redes neurais e do algoritmo de inferência.
- **C/C++ e PLI:** em Verilog e SystemVerilog é possível chamar diretamente código em C/C++ através da interface PLI (Programming Language Interface) (SUTHERLAND, 2013).
- **Programa externo e arquivos:** o verilog permite a leitura de arquivos, de modo os casos de teste podem ser pré-computados e salvos ao invés de gerados dinamicamente

As duas primeiras opções apresentam um grande nível de complexidade e consequentemente maior tempo de desenvolvimento. Com a terceira opção, é possível usar soluções em alto nível e a linguagem Python para gerar os estímulos necessários. Posto isto, nos atentemos ao desenvolvimento do golden model em python. Uma solução codificada do príncipio com o auxílio da biblioteca de computação númerica Numpy (HARRIS et al., 2020) oferece mais flexibilidade para debug e visualização dos vetores X, W, V e Y de cada camada que o uso de frameworks consagrados como o tensorflow (TENSORFLOW, 2022). Com efeito, bastaram as classes *Layer*, *NeuralNetwork* e *NeuralInterface* empreender o golden model.

Na classe *Layer* a função de propagação direta do código 5.1 implementa as equações 2.1.7 e 2.1.8. As equações 2.1.21, 2.1.19 e 2.1.20 que implementam o gradiente descendente são sintetizadas na função de propagação retroativa do código 5.2.

```
def forward_propagate(self, X : np.array) -> np.array:
    self.X = X
    self.V = np.dot(self.W, self.X)
    self.Y = self.func.fx(self.V)
    return self.Y
```

Código 5.1 – Função de propagação direta

```
def backward_propagate(self, dE_dY: np.array, learning_rate : float) ->
    np.array:
    dE_dV = dE_dY*self.func.dfx(self.V)
    dE_dW = np.dot(self.X, dE_dV.T).T
    dE_dX = np.dot(self.W.T, dE_dV)

    self.W -= learning_rate * dE_dW

    return dE_dX
```

Código 5.2 – Função de propagação retroativa

A classe *NeuralNetwork* implementa os algoritmos de inferência (Código 5.3) e treinamento (Código 5.4).

```
def predict(self, x):
    for layer in self.layers:
        x = layer.forward_propagate(x)
    return x
```

Código 5.3 – Função de inferência

```
def fit(self, x_train, y_train, epochs = 100, learning_rate = 0.1, verbose =
    False):
    error_list = []
    for epoch_index in range(epochs):
        error = 0
        for i in range(x_train.shape[1]):
            y_pred = self.predict(x)
            error += self.loss(y_pred, y)[0]

            dE_dY = self.d_loss(y_pred, y)
            for layer in reversed(self.layers):
                dE_dY = layer.backward_propagate(dE_dY,
                    learning_rate[epoch_index])

```

```
13          error /= len(x_train)
14          if verbose == True:
15              print(f'Epoch {epoch_index} error={error}')
16
17          error_list.append(error)
18      return error_list
```

Código 5.4 – Função de treinamento

Para avaliação do próprio golden model, podemos utilizá-lo em um problema conhecido e verificar sua conformidade. O dataset Iris (DUA; GRAFF, 2017) possui 3 classes de flores com 50 instâncias cada. Cada instância possui comprimento e largura de sépalas e pétalas e sua classificação em uma das três espécies, respectivamente (Tabela 12).

S_c	S_l	P_c	P_l	Espécie
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
6.0	2.9	4.5	1.5	versicolor
5.7	2.6	3.5	1.0	versicolor
6.7	3.0	5.2	2.3	virginica
6.3	2.5	5.0	1.9	virginica
6.5	3.0	5.2	2.0	virginica
⋮	⋮	⋮	⋮	⋮

Tabela 12 – Dataset iris

Podemos treinar um modelo de camadas $[4, 6, 3]$ (Figura 44) com função de ativação logística em um problema de classificação cuja entrada é o vetor com as quatro primeiras colunas da tabela 12 e a saída é um vetor tridimensional com a probabilidade de um vetor arbitrário pertencer a cada espécie.

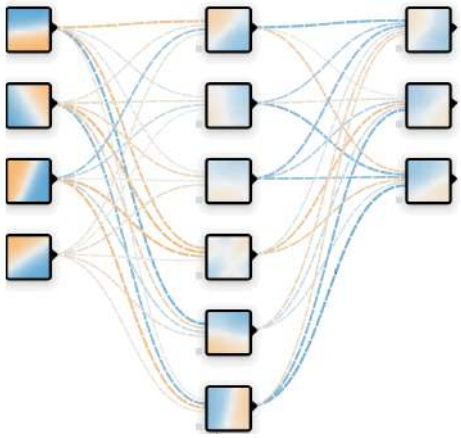


Figura 44 – Rede neural usada com o dataset iris

Uma vez que a função de treino chama todas as outras mostradas anteriormente, podemos simplesmente usá-la para testar o sistema por completo. Usando uma taxa de

aprendizado de $\eta(i) = 0.2 - 0.001i$, onde i é a atual iteração e a função de erro quadrático (2.1.15) como métrica de se obtém a curva decrescente da figura 45, como esperado.

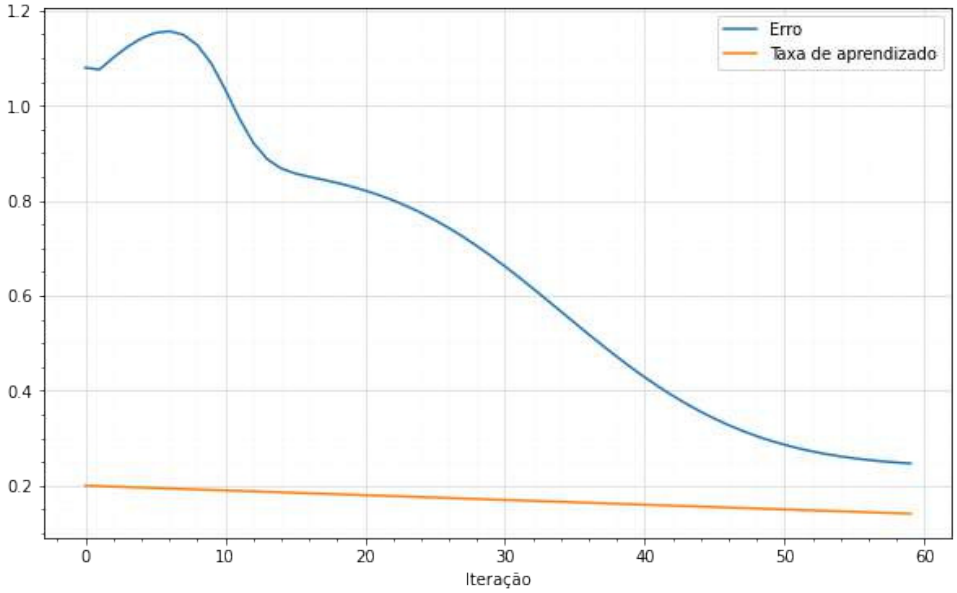


Figura 45 – Treino com o dataset Iris

Por fim, uma classe *NeuralInterface* serve de interface entre a rede neural e o acelerador, gerando a escrita de um arquivo teste de caso conforme o padrão do código 5.5. É também desta classe a responsabilidade de organizar a rede no formato das memórias do acelerador.

5.2 TESTBENCH

Testbench é um padrão de verificação funcional orientado a simulação, em que um módulo externo gera estímulos para o design a ser testado, DUT do inglês Design Under Test, e verifica a correspondência com o modelo de referência 46.(SPEAR, 2008)

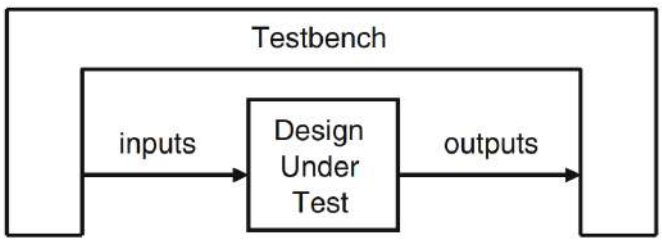


Figura 46 – Diagrama do fluxo de um testbench. (SPEAR, 2008)

O testbench lê o arquivo conforme o padrão do código 5.5. No exemplo, ele escreve seis valores nas posições de memória informadas, que por sua vez inicializam as memórias de pesos e a LUT da função de ativação com a rede gerada. Para o teste ser considerado bem sucedido, os valores numéricos das posições de memória *0x003* e *0x004* devem coincidir com

310 e 1000, respectivas representações decimais de 0.075 e 0.224 em $Q4.12$. As últimas duas linhas têm propósito de debug, mostrando a quantidade de neurônios em cada camada e o resultado da inferência em base decimal.

```
6              --> WriteCount
0xE000 20      --> Addr Value
0xE001 4097
0xE002 128
0xE003 32768
0x0000 1014
0x4000 1251

2              --> ReadCount
0x003 310      --> Addr Value
0x004 1000

[1, 3, 2]      --> Layers
[0.075, 0.224] --> OutputVector
```

Código 5.5 – Arquivo gerado

Foram gerados 1000 redes aleatórias, de 2 a 5 níveis de profundidade. Cada uma das camadas pôde ter de 1 a 7 neurônios e sua função de ativação poderia ser Sigmóide, Tangente hiperbólica, Linear ou ReLU. A quantidade de camadas e a função de ativação foi escolhida de maneira uniforme dentre as possibilidades, no entanto para a quantidade de neurônios por camada, foi tido 44.44% de probabilidade de ter 1 neurônio e 9.26% de probabilidade de possuir cada um dos níveis restantes.

Observe que as redes foram geradas aleatoriamente e não fruto de treino com problemas reais, uma vez que o que queremos testar é a conformidade com o modelo de referência.

5.3 COBERTURA DE CÓDIGO

Cobertura de código é um conceito de verificação que visa entregar métricas de quão bem explorado o espaço de busca está sendo, estabelecendo casos que devem ser cobertos e métricas de qualidade do teste (CERNY et al., 2015).

Esta ferramenta comumente disponível em produtos de EDA de acesso aberto, esse foi um fator relevante na escolha do Questasim em detrimento das outras apresentadas na seção 3.1. Após o uso das flags adequadas nos comandos de compilação e simulação, o QuestaSim entra em modo de Coverage, onde é possível visualizar as métricas na interface do programa ou em reports dedicados. As opções de cobertura na ferramenta utilizada são:

- **Branch:** testa se cada bloco de bifurcação (if/else/case/ternário) é alcançado
- **Condition:** testa os sinais usados como condições nos blocos são atendidos

- **Expression:** cobertura por cada expressão
- **Statement:** cobertura por linha de atribuição em sinais
- **Toggle:** testa transições de bit (0 -> 1 e 1 -> 0)
- **FSM State:** checa se todos os estados da FSM são atingidos
- **FSM Transition:** cobertura de transições em máquinas de estado

5.4 AVALIAÇÃO DE DESEMPENHO

Podemos avaliar a arquitetura em quantidade de ciclos de clock para realizar a inferência em uma camada e frequência máxima alcançada. A quantidade de recursos usados será uma métrica secundária.

6 RESULTADOS

6.1 ARQUITETURA

Uma boa maneira de analisar a arquitetura é pelo diagrama em nível RTL gerado pelo Quartus. O primeiro resultado a mostrar é a etapa de desenvolvimento antes da inserção do pipeline (Figura 47), que mantém os mesmos traços da arquitetura completa. A arquitetura completa pode ser vista na figura 48, porém, sua visualização é emaranhada devido aos registradores e elementos de dataforwarding inerentes ao pipeline.

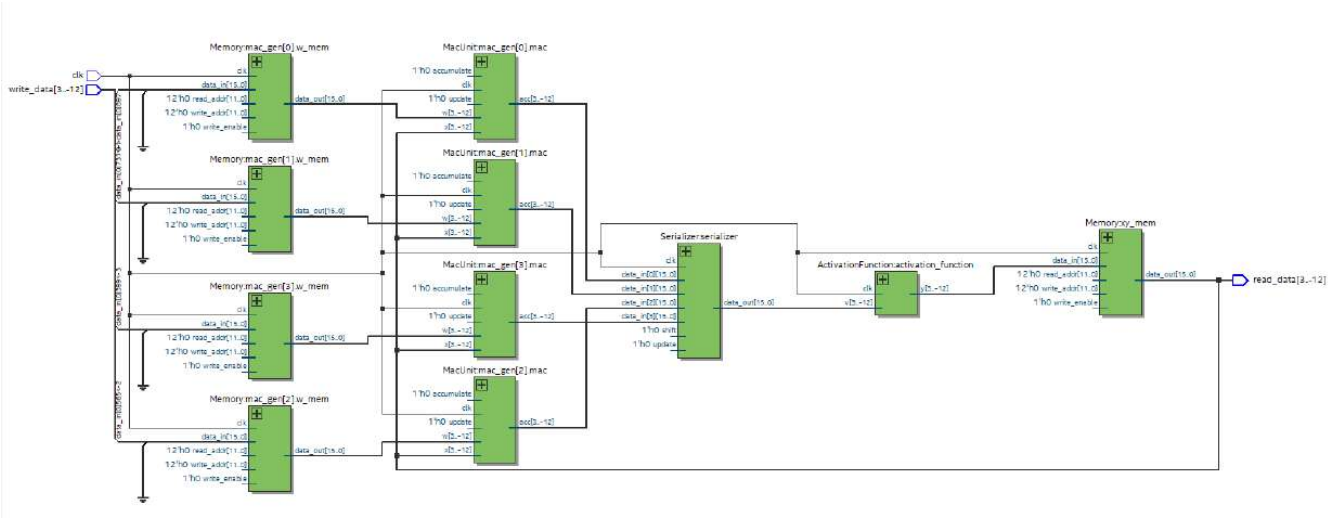


Figura 47 – Diagrama RTL do datapath e sem pipeline

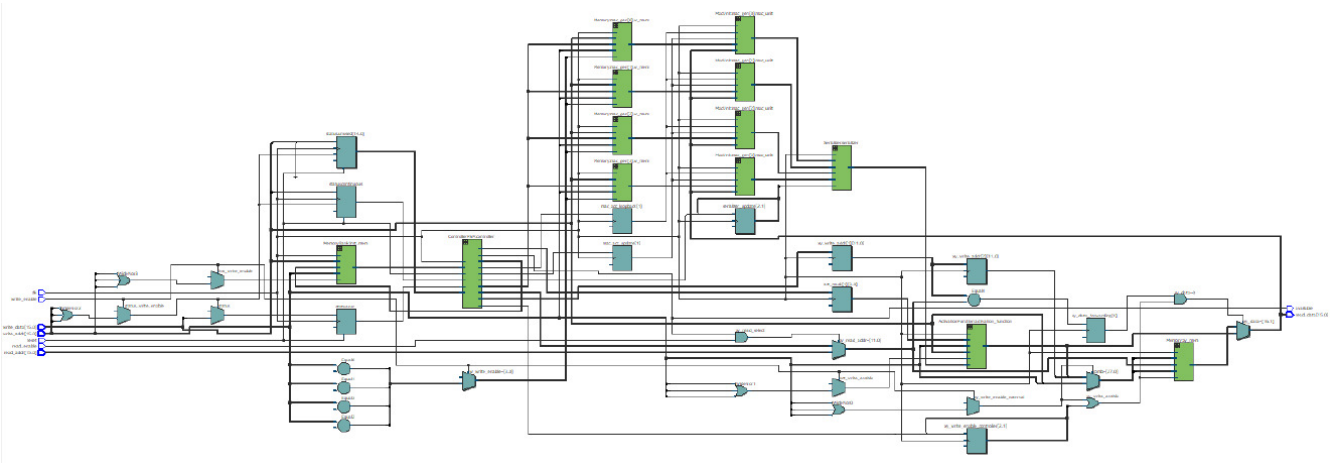


Figura 48 – Diagrama RTL da arquitetura

Pode-se também visualizar os diagramas RTL para os componentes utilizados, como segue abaixo.

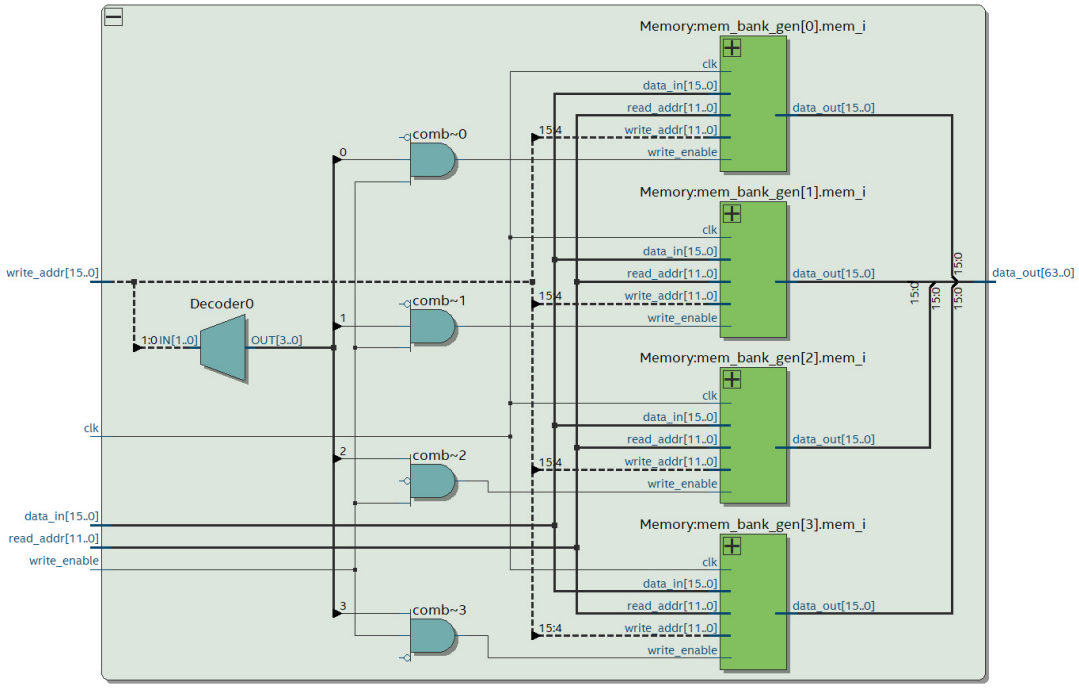


Figura 49 – Diagrama RTL do banco de memórias

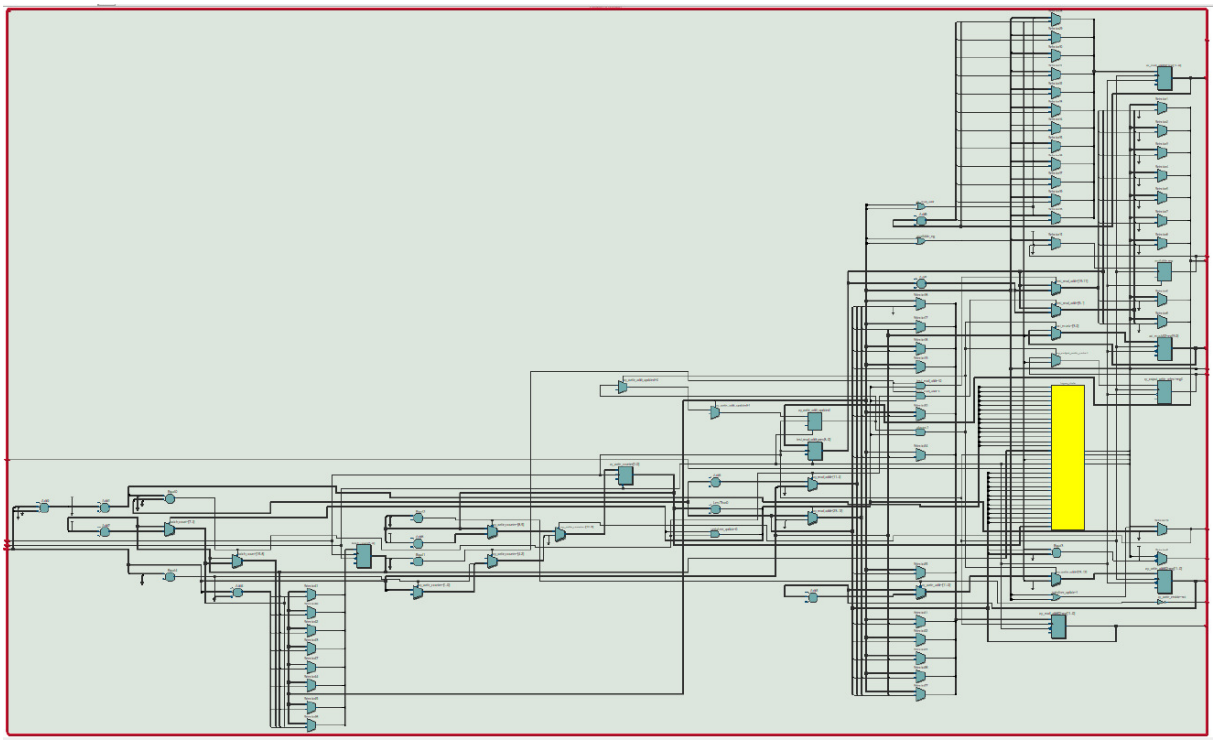


Figura 50 – Diagrama RTL do controlador

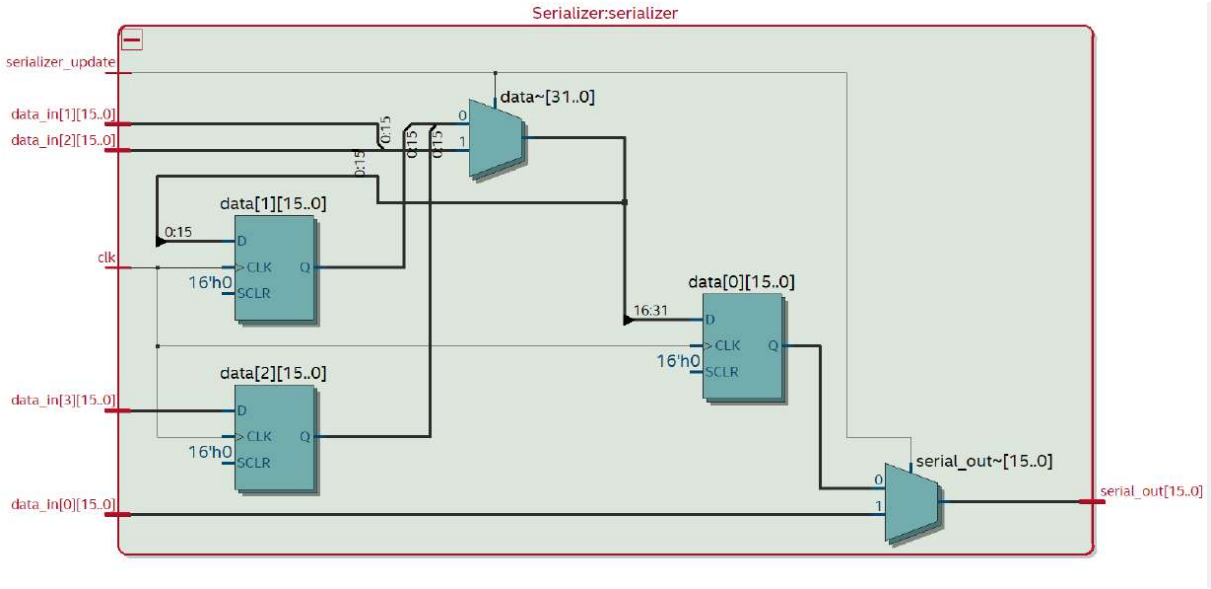


Figura 51 – Diagrama RTL do serializador

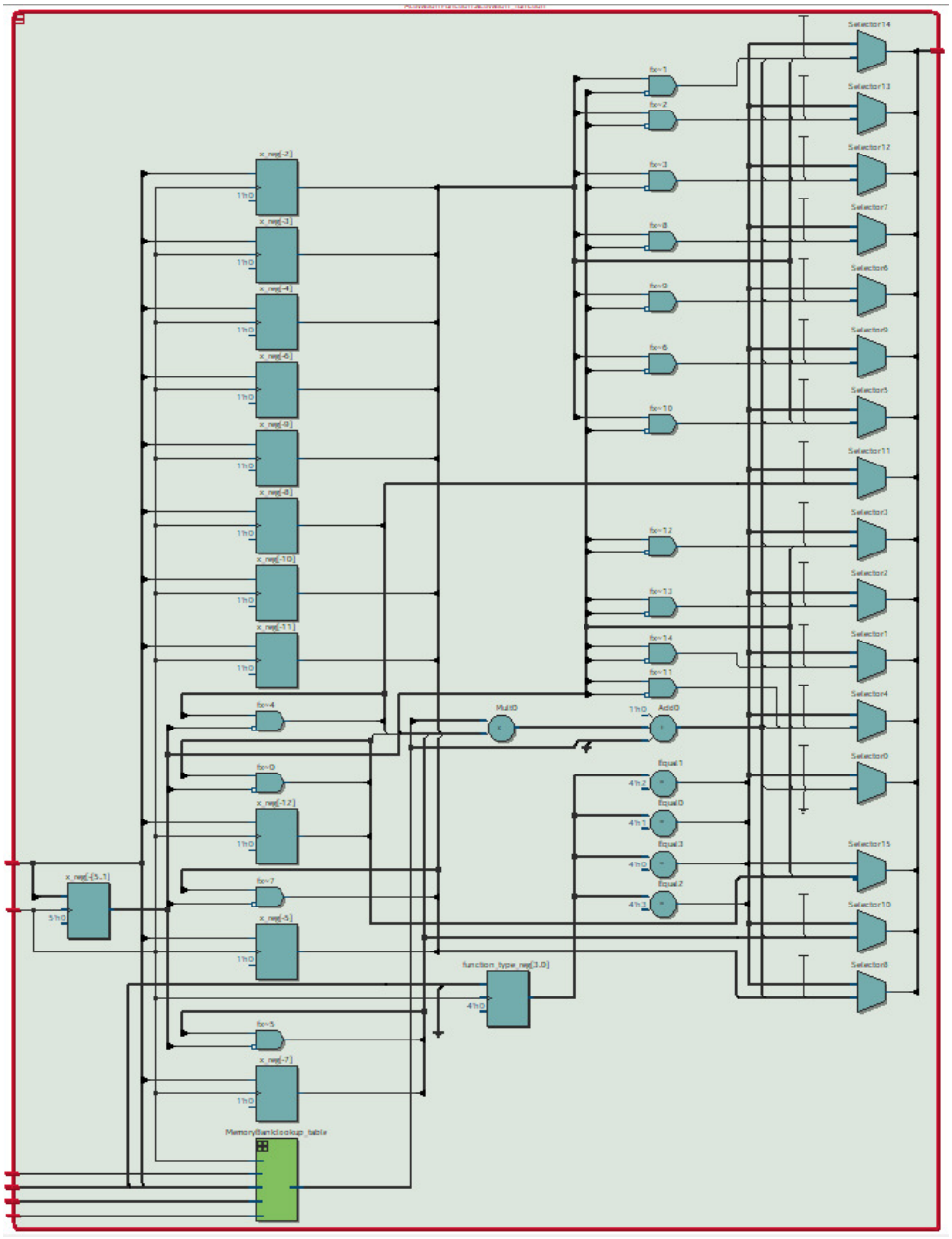


Figura 52 – Diagrama RTL da função de ativação

O código tanto em descrição de hardware da arquitetura tanto quanto das interfaces de programação usadas para testá-la estão disponíveis segundo o paradigma open-source no repositório (SOUSA, 2023).

6.2 SIMULAÇÃO

Através da ferramenta de EDA Questasim e da metodologia indicada foi possível obter os resultados de simulação, essenciais no processo de debug da arquitetura. Com os estímulos adequados do testbench, a ferramenta projeta em sua tela as formas de onda da simulação feita. Podemos ver nas figuras 53 e 54 os respectivos processos de escrita e leitura no dispositivo de memória mapeado, conforme previsto nas formas de onda das figuras 43 e 42. A figura 55

mostra o comportamento interno da arquitetura durante o cálculo da inferência. Nas simulações foram usados os casos de teste descritos na seção 5.

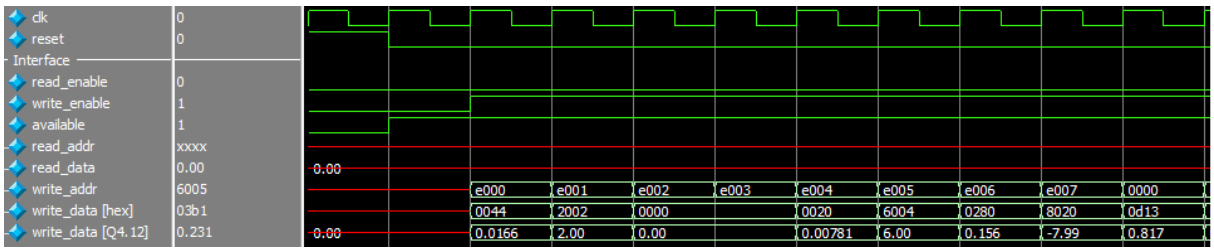


Figura 53 – Escrita em memória

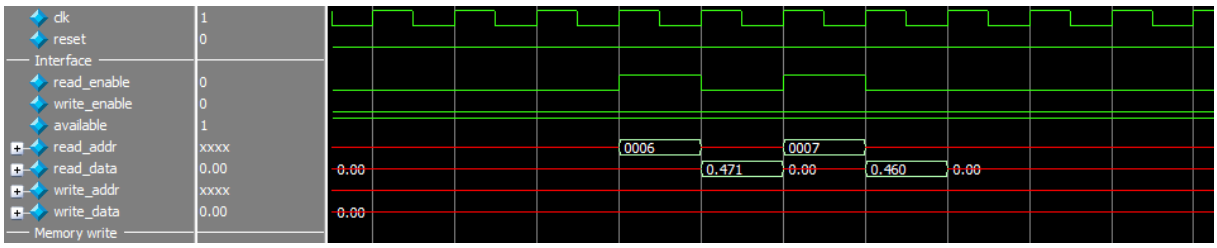


Figura 54 – Leitura em memória

Figura 55 – Sinais internos

6.3 ACURÁCIA

Através da metodologia indicada no capítulo 5, foi possível obter as saídas das redes calculadas pelo design, e compará-las com os valores de referência do golden model usando a métrica de MSE. Podemos agrupar os dados obtidos numa distribuição de frequências em lotes de acordo com a tabela 13.

Quantidade	Limite inferior	Limite superior
917	0.0000000	0.0118164
37	0.0118164	0.0236328
16	0.0236328	0.0354492
11	0.0354492	0.0472650
4	0.0472650	0.0590820
8	0.0590820	0.0708984
4	0.0708984	0.0827148
0	0.0827148	0.0945312
1	0.0945312	0.1063476
2	0.1063476	0.1181640

Tabela 13 – Distribuição da métrica de MSE para os casos de teste

Observe que 91.7%, quase a totatilidade dos casos de teste, obteve um MSE de até 0.0118164 e em nenhum caso a métrica ultrapassou o valor de 0.1181640. Apesar de aceitáveis, valores melhores podem ser alcançados com o aumento do comprimento da palavra, atualmente de 16 bits.

6.4 COBERTURA DE CÓDIGO

O QuestaSim permite a geração de um relatório que mostra a cobertura por módulo e instância no design. A figura 56 expõe a tela do relatório para o módulo controlador, onde é possível ver a cobertura para cada um dos tipo mencionados.

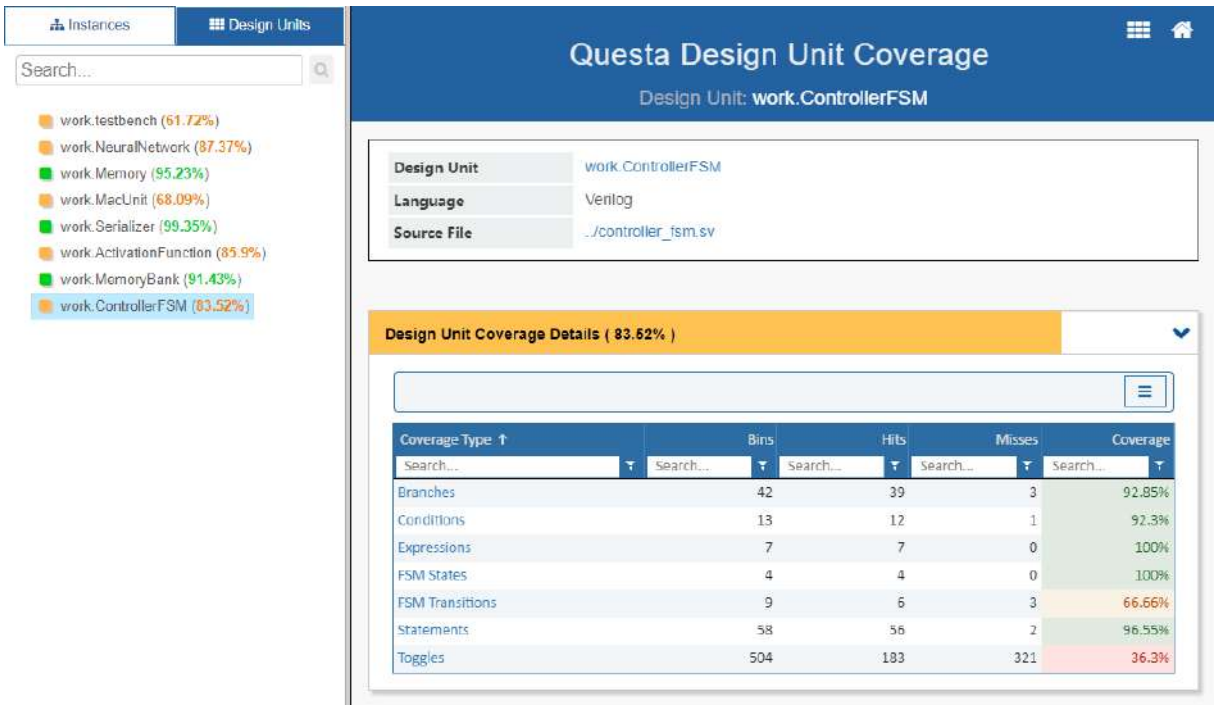


Figura 56 – Cobertura da função de ativação

A transcrição da cobertura por módulo por ser vista na tabela 14.

Design	Branches	Conditions	Expressions	Statements	Toggles
NeuralNetwork	95.65	100	80	100	61.2
Memory	100	-	-	100	85.71
MacUnit	66.66	100	0	76.47	97.35
Serializer	100	-	-	-	98.07
ActivationFunction	80	-	-	90.9	86.8
MemoryBank	100	-	-	-	74.29
ControllerFSM	92.85	92.3	100	96.55	36.3

Tabela 14 – Cobertura dos módulos em %

As máquinas de estado obtém um tratamento especial na cobertura de código, já que precisam ser testadas tanto em estados quanto em transições, a tabela 15 mostra os resultados para a FSM do controlador. Na tabela 16 vemos a presença do teste para todas as transições com exceção de Start -> Idle e Wait -> Wait.

Design	FSM States	FSM Transitions
ControllerFSM	100%	66.66%

Tabela 15 – Cobertura da máquina de estados

Estado	Idle	Start	Acc	Wait
Idle	s	s	-	-
Start	n	-	s	-
Acc	s	s	s	s
Wait	n	s	n	s

Tabela 16 – Cobertura das transições da máquina de estados

Observa-se que falha de cobertura não necessariamente indica falha no design, mas falha de geração de um caso de teste que permita o teste de uma funcionalidade ou sinal específico. A análise de cobertura permite também verificar a origem da falha de cobertura. Adentrando na hierarquia do design ActivationFunction, por exemplo (figura 57) podemos perceber que existe uma falha de cobertura uma vez que o caso FUNC_STEP não é alcançado. Na prática, isso significa que não testamos a função degrau nesse tesbench.

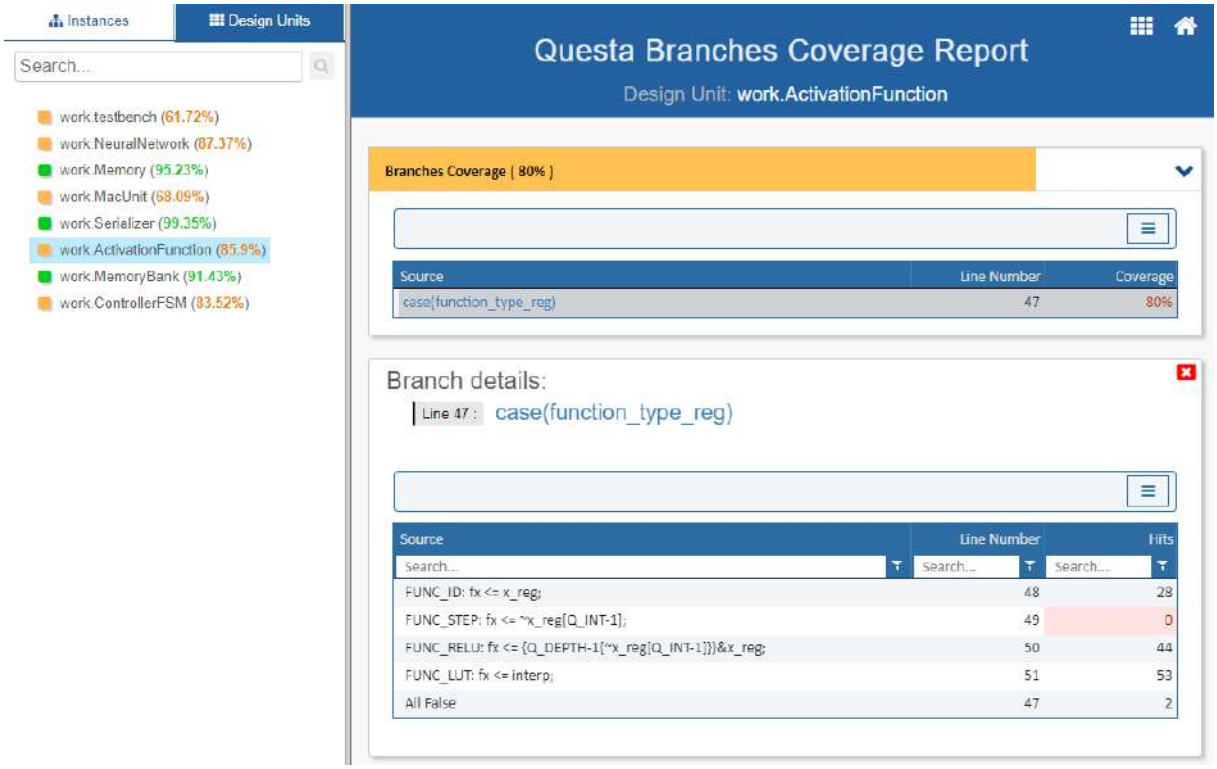


Figura 57 – Coverage de um branch da função de ativação

Certamente, para uma maior cobertura é necessário: a) um maior número de casos de testes randômicos; b) um testbench que conscientemente englobe *corner cases*. O primeiro fator é facilmente alcançável pela mudança do valor de parâmetros no testbench, no entanto, o segundo requer um trabalho maior de verificação. O testbench passou por várias iterações melhorando os fatores a) e b), contudo, por limitações de tempo, nem todos os casos foram atingidos.

6.5 DESEMPENHO

Seja uma matriz de pesos W de dimensões m x n e o vetor de entrada X de dimensão n. Uma arquitetura com N_{cores} unidades MAC consegue realizar as operações de produto (Equação 2.1.7) e aplicação da função de ativação (Equação 2.1.8) em

$$1 + \left\lceil \frac{m}{N_{cores}} \right\rceil \cdot n \text{ ciclos de clock} \tag{6.5.1}$$

E para uma rede neural de D camadas,

$$\sum_{i=0}^{D} 1 + \left\lceil \frac{m_i}{N_{cores}} \right\rceil \cdot n_i \text{ ciclos de clock} \tag{6.5.2}$$

Podemos usar as equações 6.5.1 e 6.5.2 para determinar a performance da inferência para arquiteturas de diferentes quantidades de núcleos. A figura 58 mostra a contribuição de cada camada para uma rede de dimensões $[10, 4, 23, 7, 14]$. A menor quantidade de ciclos,

48, é obtida quando a quantidade de núcleos é igual ao maior número de neurônios em uma camada da rede, isto é, 23, de modo que a inserção de mais núcleos é redundante. Para efeitos de comparação, utilizando a estimativa de $4nm$ ciclos por camada em um processador não otimizado para tarefa, esta mesma rede gastaria 1564 ciclos de clock, 4 vezes mais lenta em comparação com a arquitetura de somente um núcleo.

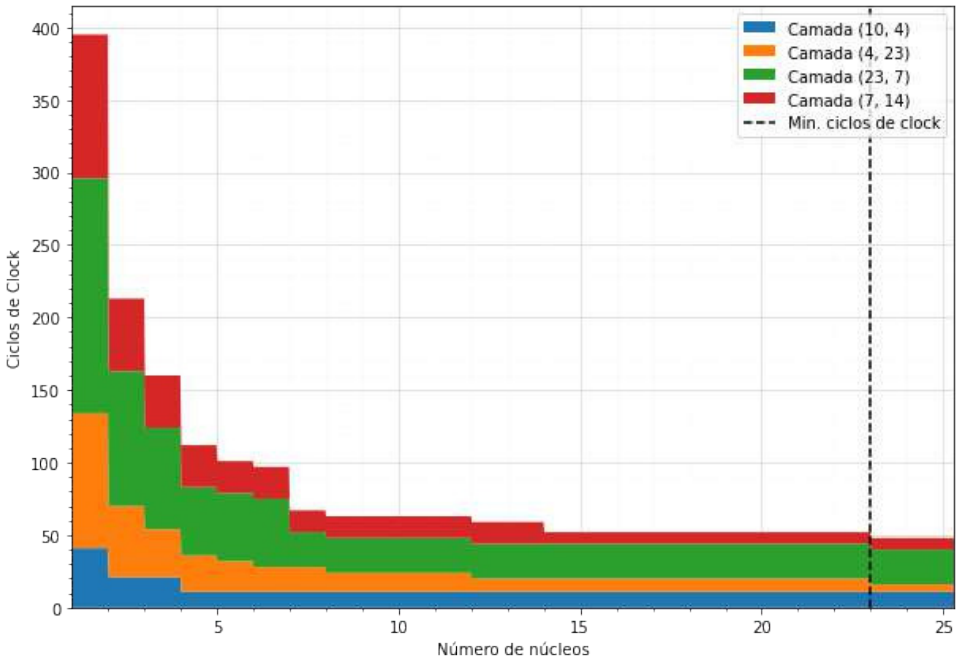


Figura 58 – Quantidade de ciclos de clock por camada

6.6 RESULTADOS DEPENDENTES DE TECNOLOGIA

São resultados dependentes de tecnologia a quantidade de recursos usados, frequências máximas em pontos de operação específicos, dentre outros. Esses resultados podem variar muito de acordo com o modelo de FPGA utilizado, e a geração de fabricação. A ferramente de EDA Quartus aponta relatórios com esses dados referentes à compilação para o FPGA 5CSXFC6D6F31C6N, tomado como padrão na seção 4.1, na tabela 18.

Recurso	Usado	Máximo	Percentual
Blocos lógicos	418	41,910	< 1 %
Registradores	651	não reportado	não reportado
Pinos	69	499	14 %
Blocos de memória	391,680	5,662,720	7 %
Blocos de RAM	48	553	9 %
Blocos de DSP	5	112	4 %

Tabela 17 – Recursos usados

É fácil ver que o uso de recursos se manteve baixo em todos os aspectos. O parâmetro mais relevante talvez seja a frequência alcançada, que pode ser vista na tabela 18. Como é

sabido pela literatura, alguns parâmetros de propagação (seção 2.2.3) dependem das condições de operação, assim sendo, a ferramenta entrega no relatório a frequência máxima suportada, simulada em diferentes condições de temperatura e tensão de alimentação.

Tensão	Temperatura	F_{max}
1100mV	85C	59.34 MHz
1100mV	0^{o}C	58.9 MHz

Tabela 18 – Frequência máxima

7 CONCLUSÕES

O uso de hardware digital dedicado à computação de redes neurais não é algo novo, no entanto, é cada vez necessário, vista a necessidade por crescente demanda de poder computacional nesta área. Nesse sentido, o trabalho apresentou uma solução com fim de atender a esse problema, se aproveitando da paralelização para acelerar o algoritmo de inferência para redes neurais.

A arquitetura aqui implementada é altamente flexível, bastando a mudança de alguns parâmetros para mudança de número de núcleos de processamento, tamanho da palavra com consequente aumento de acurácia, e até mesmo tamanho das memórias. A possibilidade de armazenar diversas funções na LUT da função de ativação acrescenta à flexibilidade. Essa característica a permite um balanceamento entre recursos usados e aumento de performance. O seu paradigma similar a um processador de programa armazenado permite que qualquer rede neural direta seja implementada, no limite do tamanho das memórias. Algumas outras arquiteturas são também suportadas desde que possam devidamente separadas em camadas.

Os resultados de simulação de performance mostraram que a arquitetura é altamente otimizada para compactar operações em um mesmo ciclo de clock, alcançando uma melhoria de performance da ordem de grandeza da quantidade de núcleos de processamento usados. Reutilização de recursos foi sempre utilizado quando não impactasse a performance, de modo que os componentes de função de ativação e memória xy foram compartilhados por todos os núcleos. Os testes aplicados foram em redes de pesos aleatórios, contudo, o mesmo nível de acurácia deve ser visto para redes treinadas de mesmas dimensões. Os resultados de cobertura de código também revelam que grande parte do código está verificado.

7.1 TRABALHOS FUTUROS

Alguns pontos foram deixados de lado deste trabalho por questões de prazos e escopo, estes devem ser os próximos a serem resolvidos:

- Refatoração do código para conformidade com padrões de estilo e organização
- Testes com diferentes de parâmetros, em especial de quantidade de núcleos.
- Testes em FPGA
- Testes com Datasets
- Maior cobertura de código
- Inserção ou reorganização de estágios de pipeline, em especial no caminho crítico

Além de pontos de reparação, existem ainda pontos de melhoria que podem ser implementados em próximas iterações:

- Inserção de uma fila para escrita na memória xy, a fim de aproveitar os ciclos de clock usados para escrita das entradas nesta memória

- Utilização de vários aceleradores em série, o que permite mais uma dimensão de paralelismo. Para tanto se faz necessário a inserção de uma lógica para permitir o *stall* de escrita quando o próximo acelerador estiver ocupado
- Implementação em ASIC
- Modificação que permita o treinamento, tratada com mais detalhes abaixo

Uma grande funcionalidade com o potencial de ser adiciona em trabalhos futuros é o treinamento em hardware, o que é possível com ligeiras modificações na arquitetura, nas unidades MAC e controlador. Como última sugestão deste trabalho, aponta-se uma modificação na unidade MAC para implementação de tal funcionalidade (Figura 59).

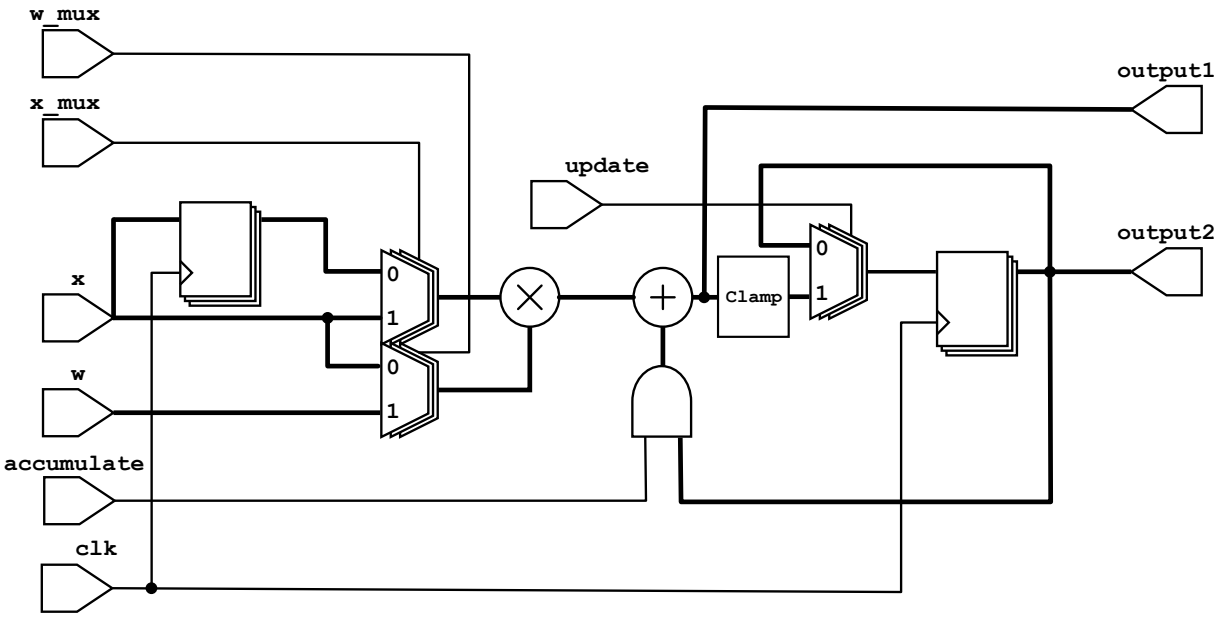


Figura 59 – Unidade MAC adaptada para treinamento

Com a correta utilização dos sinais x_mux e w_mux é possível implantar todas as equações necessárias para realização do treinamento por propagação retroativa, como é possível ver na tabela 19.

x_mux	w_mux	Equação
0	0	-
0	1	2.1.20, 2.1.21
1	0	2.1.19
1	1	2.1.7

Tabela 19 – Operações realizadas na unidade MAC adaptado para treinamento

A possibilidade de treinamento em FPGA além da possível melhoria em performance proposta, permite também o treinamento em campo, na ocasião da aquisição de dados em tempo real.

REFERÊNCIAS

ACOSTA, N.; TOSINI, M. A. Custom architectures for fuzzy and neural networks controllers. **Journal of Computer Science & Technology**, v. 2, 2002. Citado 2 vezes nas páginas 26 e 27.

ALTERA. **Quartus Prime Standard Edition Handbook Volume 1: Design and Synthesis**. 2016. Disponível em: <https://www.intel.com/content/dam/support/us/en/programmable/support-resources/bulk-container/pdfs/literature/hb/qts/archives/qts-qps-handbook-16.0.pdf>. Citado 2 vezes nas páginas 16 e 21.

ALTERA. **Understanding Metastability in FPGAs**. 2019. Disponível em: <https://cdrdv2-public.intel.com/650346/wp-01082-quartus-ii-metastability.pdf>. Acesso em: 15/01/2023. Citado na página 17.

AMANO, H. **Principles and Structures of FPGAs**. [S.l.]: Springer, 2018. Citado 2 vezes nas páginas 19 e 20.

ARORA, M. **The art of hardware architecture: Design methods and techniques for digital circuits**. [S.l.]: Springer Science & Business Media, 2011. Citado 2 vezes nas páginas 16 e 17.

BRENT, R. P.; ZIMMERMANN, P. **Modern computer arithmetic**. [S.l.]: Cambridge University Press, 2010. v. 18. Citado na página 15.

CADENCE. **Cadence Design Systems**. 2022. Disponível em: https://www.cadence.com/. Acesso em: 11/12/2022. Citado na página 23.

CERNY, E. et al. **SVA: the power of assertions in systemVerilog**. [S.l.]: Springer, 2015. Citado 2 vezes nas páginas 41 e 45.

CHU, P. P. **Embedded SoPC design with NIOS II processor and Verilog examples**. [S.l.]: John Wiley & Sons, 2012. Citado 2 vezes nas páginas 19 e 20.

COUTINHO, S. C. **Números inteiros e criptografia RSA**. [S.l.]: IMPA, 1997. Citado na página 13.

DIAS, F. M.; ANTUNES, A.; MOTA, A. M. Artificial neural networks: a review of commercial hardware. **Engineering Applications of Artificial Intelligence**, Elsevier, v. 17, n. 8, p. 945–952, 2004. Citado na página 1.

DUA, D.; GRAFF, C. **Iris Dataset**. 2017. UCI Machine Learning Repository. Disponível em: <http://archive.ics.uci.edu/ml>. Citado na página 43.

ERTEL, W. **Introduction to artificial intelligence**. [S.l.]: Springer, 2018. Citado na página 2.

GEITGEY, A. **Machine learning is fun**. [S.l.]: Geitgey, Adam, 2018. Citado 3 vezes nas páginas 2, 3 e 8.

GOLBERG, D. E. Genetic algorithms in search, optimization, and machine learning. **Addion**

wesley, v. 1989, n. 102, p. 36, 1989. Citado na página 8.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. Deep learning (adaptive computation and machine learning series). **Cambridge Massachusetts**, p. 321–359, 2017. Citado 3 vezes nas páginas 4, 7 e 34.

GTKWAVE. **GTKWave**. 2022. Disponível em: https://gtkwave.sourceforge.net/. Acesso em: 11/12/2022. Citado na página 23.

GUIDORIZZI, H. L. **Um curso de cálculo: Volume 2**. [S.l.]: Grupo Gen-LTC, 2000. Citado na página 9.

HARRIS, C. R. et al. Array programming with NumPy. **Nature**, Springer Science and Business Media LLC, v. 585, n. 7825, p. 357–362, set. 2020. Disponível em: <https://doi.org/10.1038/s41586-020-2649-2>. Citado na página 41.

HAYKIN, S. **Neural networks and learning machines, 3/E**. [S.l.]: Pearson Education India, 2009. Citado 8 vezes nas páginas 2, 3, 5, 6, 7, 8, 9 e 11.

HENNESSY, J. L.; PATTERSON, D. A. **Computer organization and design risc-v edition: The hardware software interface**. [S.l.]: Elsevier Science & Technology Books, 2017. Citado 3 vezes nas páginas 21, 31 e 32.

ICARUS, V. **Icarus Verilog**. 2022. Disponível em: http://iverilog.icarus.com/. Acesso em: 11/12/2022. Citado na página 23.

IEEE. Ieee standard for verilog hardware description language. **IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)**, p. 1–590, 2006. Citado na página 24.

IEEE. Ieee standard for standard systemc language reference manual. **IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)**, p. 1–638, 2012. Citado na página 24.

IEEE. Ieee standard for systemverilog–unified hardware design, specification, and verification language. **IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)**, p. 1–1315, 2018. Citado na página 24.

IEEE. Ieee standard for floating-point arithmetic. **IEEE Std 754-2019 (Revision of IEEE 754-2008)**, p. 1–84, 2019. Citado na página 13.

IEEE. Ieee standard for vhdl language reference manual. **IEEE Std 1076-2019**, p. 1–673, 2019. Citado na página 24.

IENNE, P. et al. Digital systems for neural networks. 01 1995. Citado na página 1.

INTEL. **Software Intel Quartus Prime**. 2022. Disponível em: https://www.intel.com.br/content/www/br/pt/products/details/fpga/development-tools/quartus-prime.html. Acesso em: 11/12/2022. Citado na página 23.

INTEL. **Architecture All Access: Modern FPGA Architecture**. 2023. Disponível em: <https://www.youtube.com/watch?v=EVy4KEj9kZg>. Acesso em: 08/02/2023. Citado na página 20.

KAUR, J.; SOOD, L. Comparison between various types of adder topologies. **IJCST**, v. 6, n. 1,

p. 62–66, 2015. Citado na página 15.

KILTS, S. **Advanced FPGA design: architecture, implementation, and optimization**. [S.l.]: John Wiley & Sons, 2007. Citado 2 vezes nas páginas 21 e 22.

KIM, Y.-C.; KANG, D.-K.; LEE, T.-W. Risc-based coprocessor with a dedicated vlsi neural network. In: IEEE. **1998 IEEE International Conference on Electronics, Circuits and Systems. Surfing the Waves of Science and Technology (Cat. No. 98EX196)**. [S.l.], 1998. v. 3, p. 281–283. Citado na página 1.

LAVAGNO, L. et al. **Electronic design automation for IC implementation, circuit design, and process technology: circuit design, and process technology**. [S.l.]: CRC Press, 2016. Citado na página 23.

LIANG, J.; TESSIER, R.; MENCER, O. Floating point unit generation and evaluation for fpgas. In: IEEE. **11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.** [S.l.], 2003. p. 185–194. Citado na página 14.

LUTZ, M. **Python Pocket Reference: Python In Your Pocket**. [S.l.]: "O'Reilly Media, Inc.", 2014. Citado na página 24.

MOORE, G. E. Cramming more components onto integrated circuits. **Proceedings of the IEEE**, Ieee, v. 86, n. 1, p. 82–85, 1998. Citado na página 23.

NIKHIL, R. Bluespec system verilog: efficient, correct rtl from high level specifications. In: **Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.** [S.l.: s.n.], 2004. p. 69–70. Citado na página 24.

OH, K.-S.; JUNG, K. Gpu implementation of neural networks. **Pattern Recognition**, Elsevier, v. 37, n. 6, p. 1311–1314, 2004. Citado na página 1.

OLIVEIRA, J. d. G. M. d. **Uma arquitetura reconfigurável de Rede Neural Artificial utilizando FPGA.** Dissertação (Mestrado) — Universidade Federal de Itajubá, 2017. Citado 2 vezes nas páginas 5 e 26.

OPENROAD. **The OpenROAD Projet**: Foundations and realization of open and accessible design. 2022. Disponível em: https://theopenroadproject.org/. Acesso em: 11/12/2022. Citado na página 23.

OUSTERHOUT, J. K.; JONES, K. **Tcl and the Tk Toolkit Second Edition**. [S.l.: s.n.], 2009. Citado na página 24.

PEDRONI, V. A. **Finite state machines in hardware: theory and design (with VHDL and SystemVerilog)**. [S.l.]: MIT press, 2013. Citado 2 vezes nas páginas 18 e 19.

PYTHON. **Python 3 Documentation**. 2022. Disponível em: https://www.python.org/doc/. Acesso em: 08/12/2022. Citado na página 24.

RABAEY, J. M.; CHANDRAKASAN, A. P.; NIKOLIC, B. **Digital integrated circuits**. [S.l.]: Prentice hall Englewood Cliffs, 2002. v. 2. Citado na página 12.

RYBARCZYK, A.; SZULC, M. The concept of a microcontroller with neural-matrix coprocessor

for control systems that exploits reconfigurable fpgas. In: IEEE. **Proceedings of the Third International Workshop on Robot Motion and Control, 2002. RoMoCo'02.** [S.l.], 2002. p. 123–132. Citado na página 1.

SIEMENS. **Questa advanced simulator**. 2022. Disponível em: https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/. Acesso em: 11/12/2022. Citado na página 23.

SOUSA, V. H. F. de. **FPGA Neural Network Accelerator**. 2023. Disponível em: <https://github.com/VictorHerbert/FPGANeuralNetworkAccelerator>. Acesso em: 08/02/2023. Citado na página 50.

SPEAR, C. **SystemVerilog for verification: a guide to learning the testbench language features**. [S.l.]: Springer Science & Business Media, 2008. Citado na página 44.

STEFAN, G. Loops & complexity in digital systems. **Lecture Notes on Digital Design**, 2016. Citado na página 15.

SUN, Y.; KIST, A. M. Deep learning on edge tpus. **arXiv preprint arXiv:2108.13732**, 2021. Citado na página 1.

SUTHERLAND, S. **The Verilog PLI Handbook: a user's guide and comprehensive reference on the Verilog programming language interface**. [S.l.]: Springer Science & Business Media, 2013. Citado na página 41.

SYNOPSYS. **Synopsys**. 2022. Disponível em: https://www.synopsys.com/. Acesso em: 11/12/2022. Citado na página 23.

TENSORFLOW. **About Tensor Flow**. 2022. Disponível em: https://www.tensorflow.org/about. Acesso em: 08/12/2022. Citado 2 vezes nas páginas 24 e 41.

TYANEV, D.; PETKOVA, Y. Hardware divider. In: **Proceedings of the 19th International Conference on Computer Systems and Technologies**. [S.l.: s.n.], 2018. p. 139–143. Citado na página 15.

VERRYCKEN, B. **Which field of dVLSI has more demand: RTL or verification?** 2020. Disponível em: <https://www.linkedin.com/pulse/which-field-dvlsi-has-more-demand-rtl-verification-verrycken-/>. Citado na página 41.

VOLDER, J. E. The cordic trigonometric computing technique. **IRE Transactions on Electronic Computers**, EC-8, n. 3, p. 330–334, 1959. Citado na página 37.

WAKERLY, J. F. **Digital Design: Principles and Practices, 4/E**. [S.l.]: Pearson Education India, 2008. Citado 2 vezes nas páginas 14 e 15.

XILINX, A. **Vivado**. 2022. Disponível em: https://www.xilinx.com/support/university/vivado.html. Acesso em: 11/12/2022. Citado na página 23.

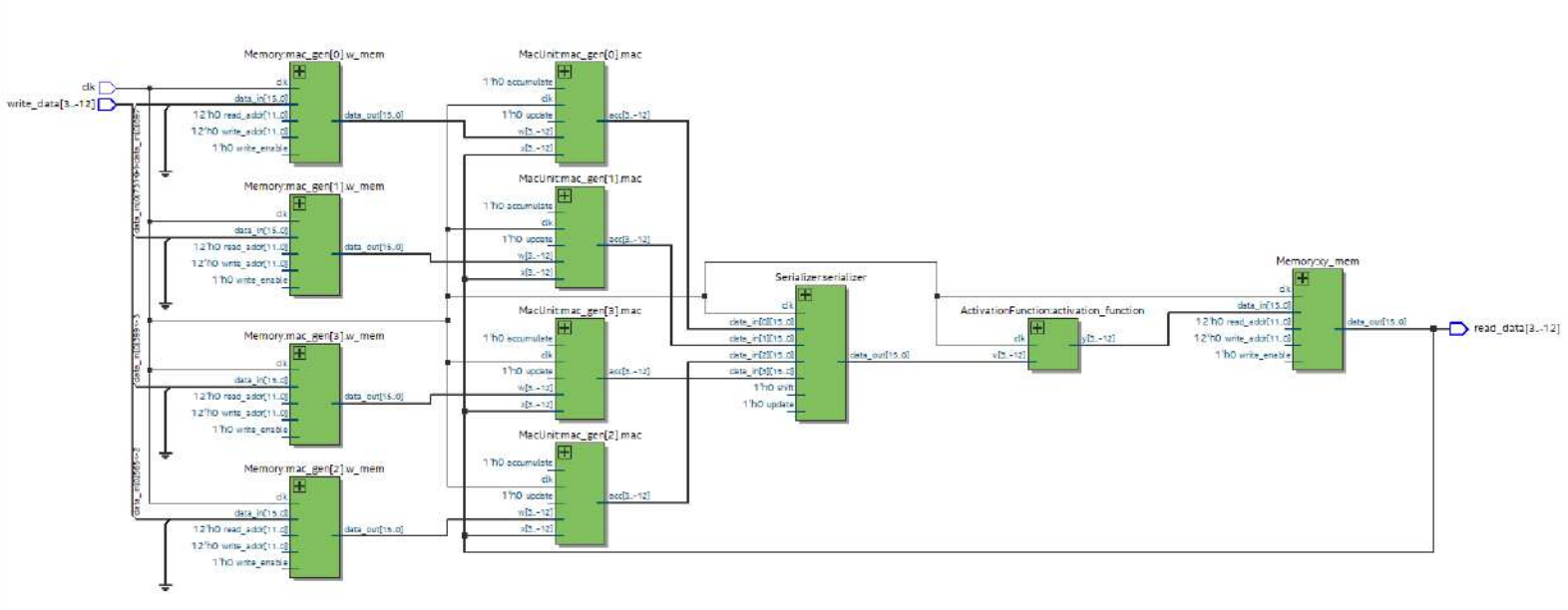
YOUSSEF, A.; MOHAMMED, K.; NASAR, A. Two novel generic, reconfigurable neural network fpga architectures. In: IEEE. **2014 4th International Conference on Artificial Intelligence with Applications in Engineering and Technology**. [S.l.], 2014. p. 3–7. Citado na página

26.

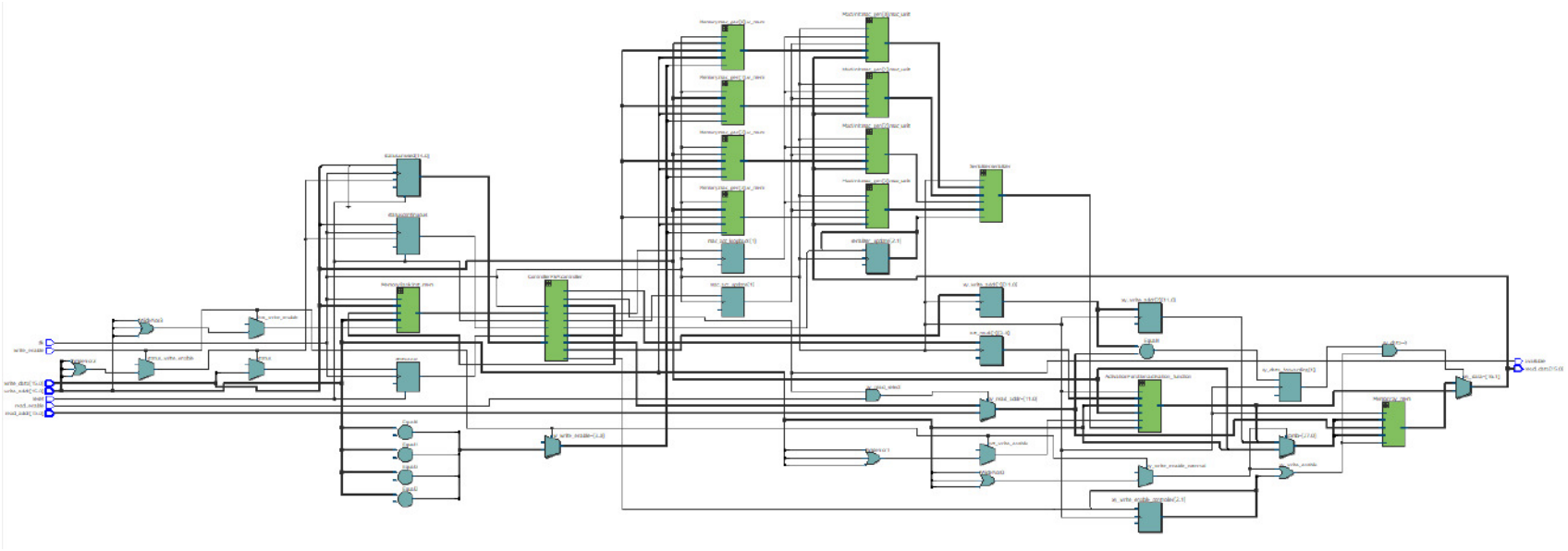
ÍNDICE REMISSIVO

Apêndices

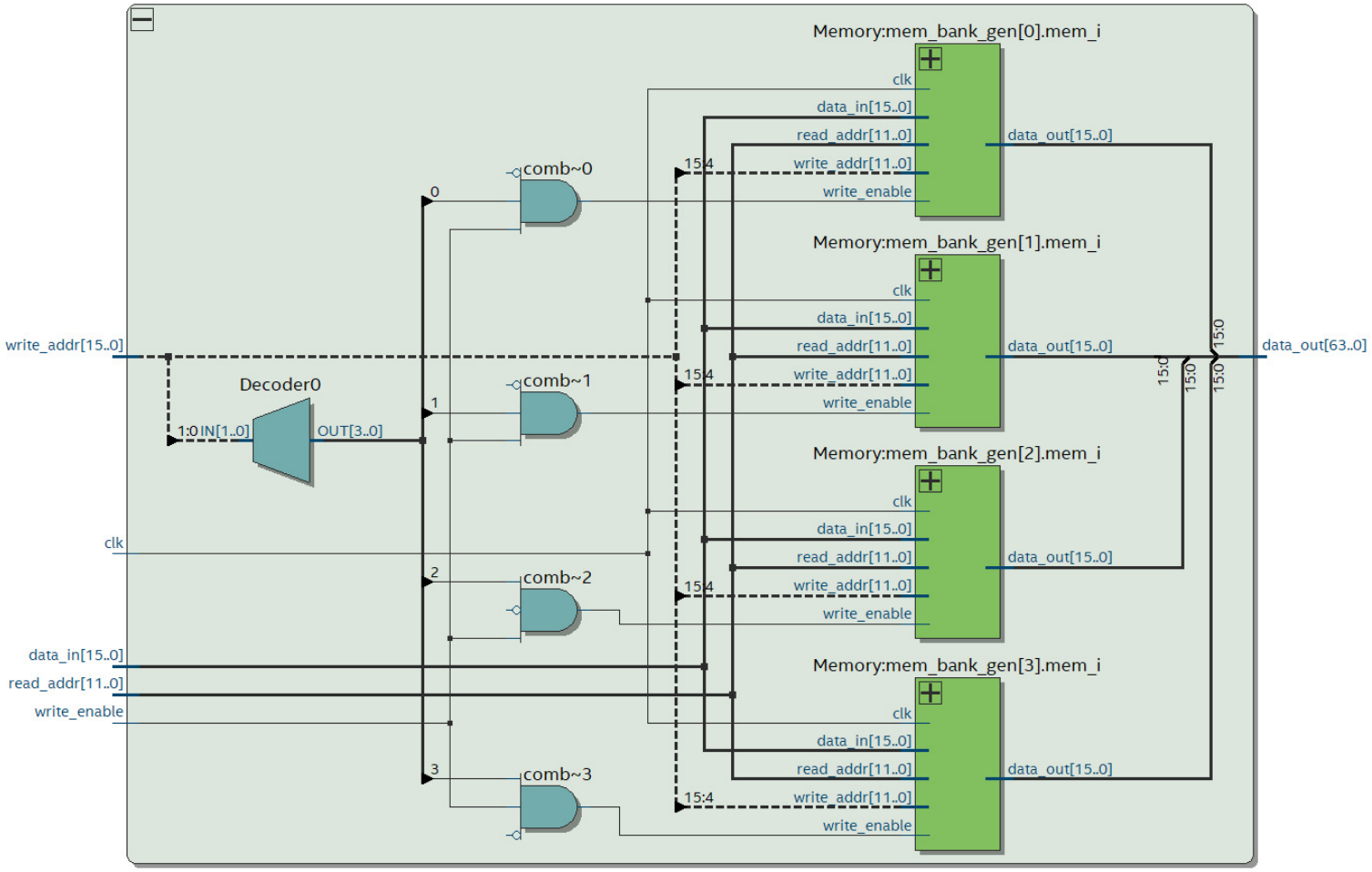
APÊNDICE A – Diagrama RTL do datapath e sem pipeline



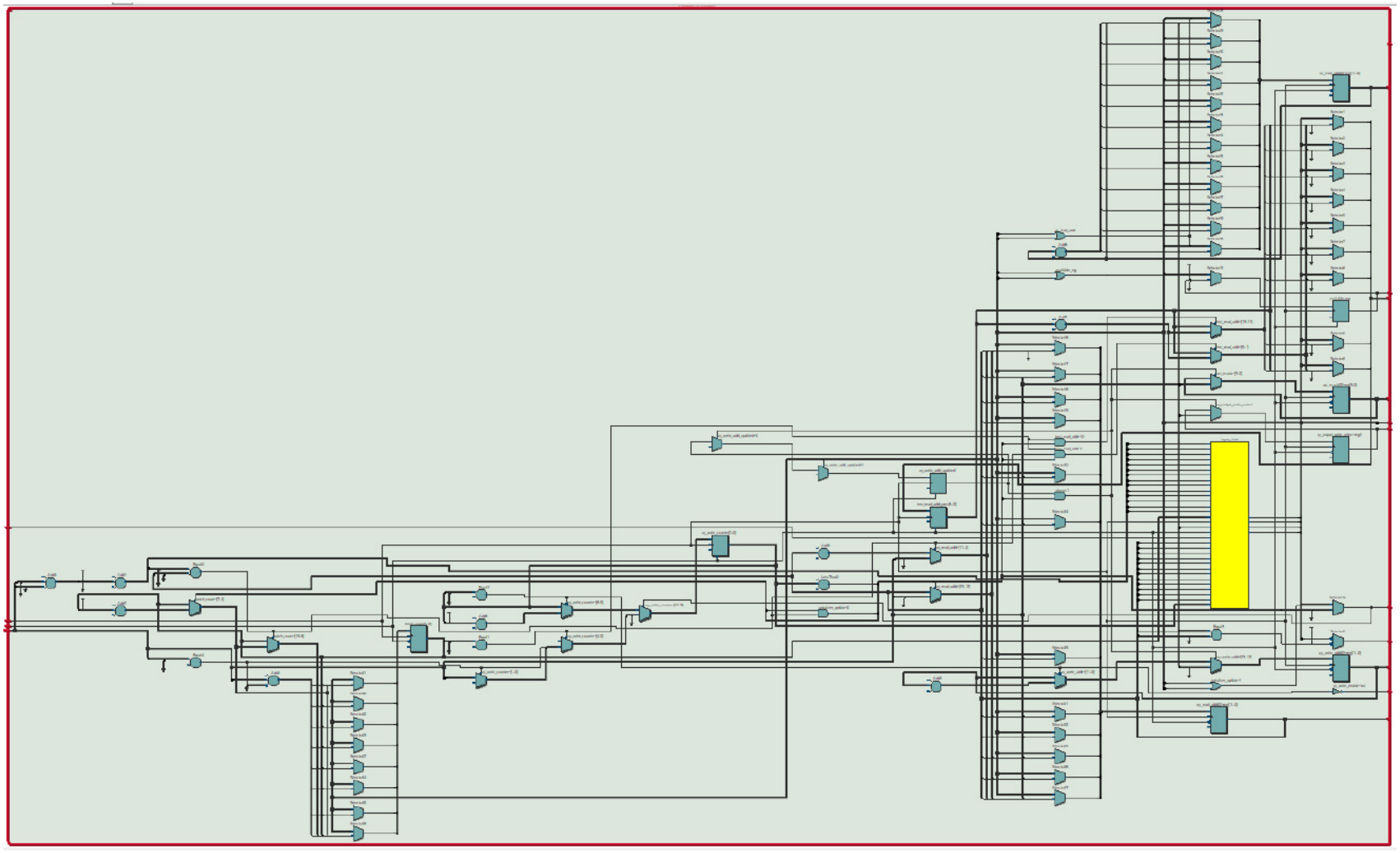
APÊNDICE B – Diagrama RTL da arquitetura



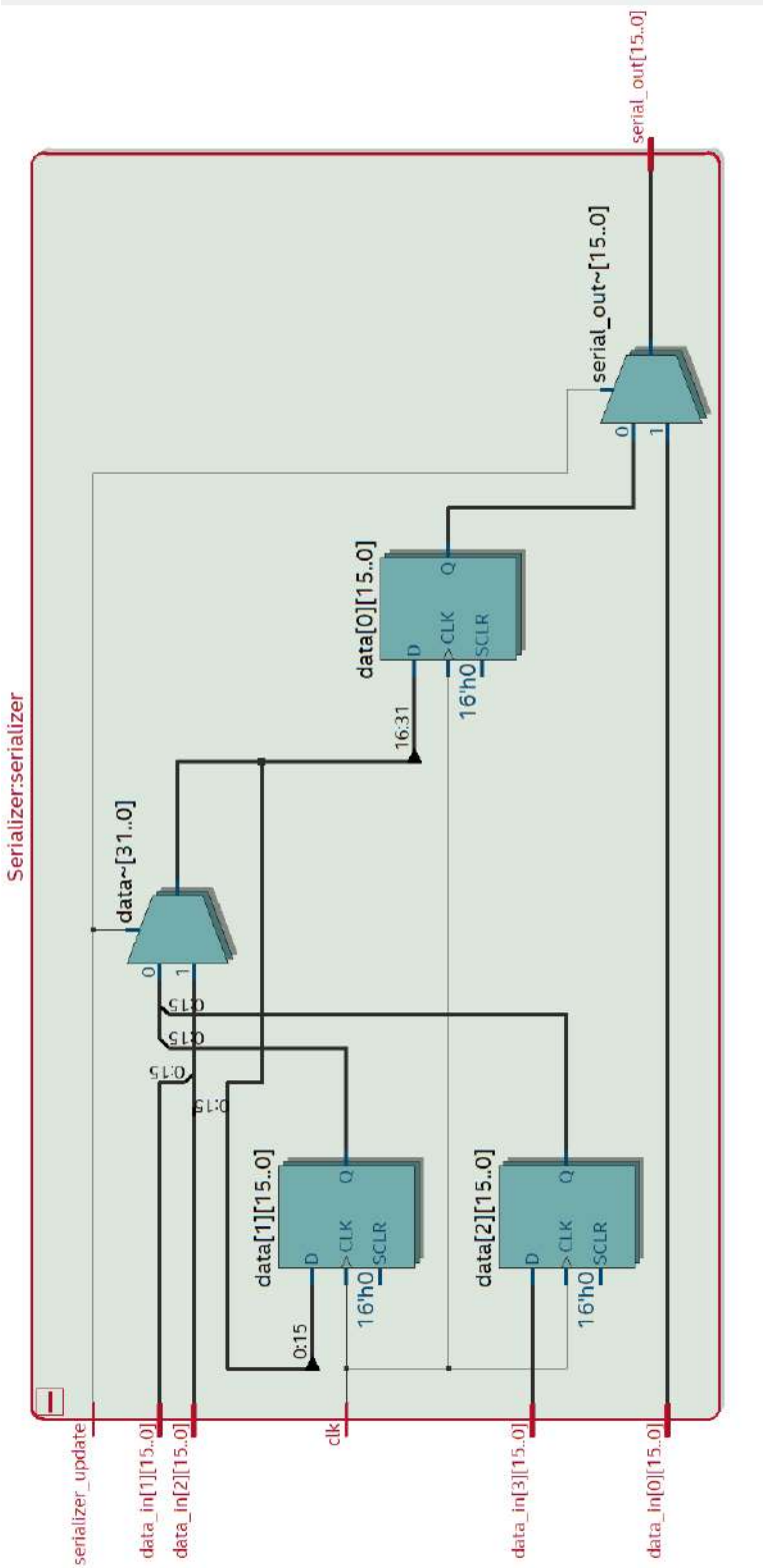
APÊNDICE C – Diagrama RTL do banco de memórias



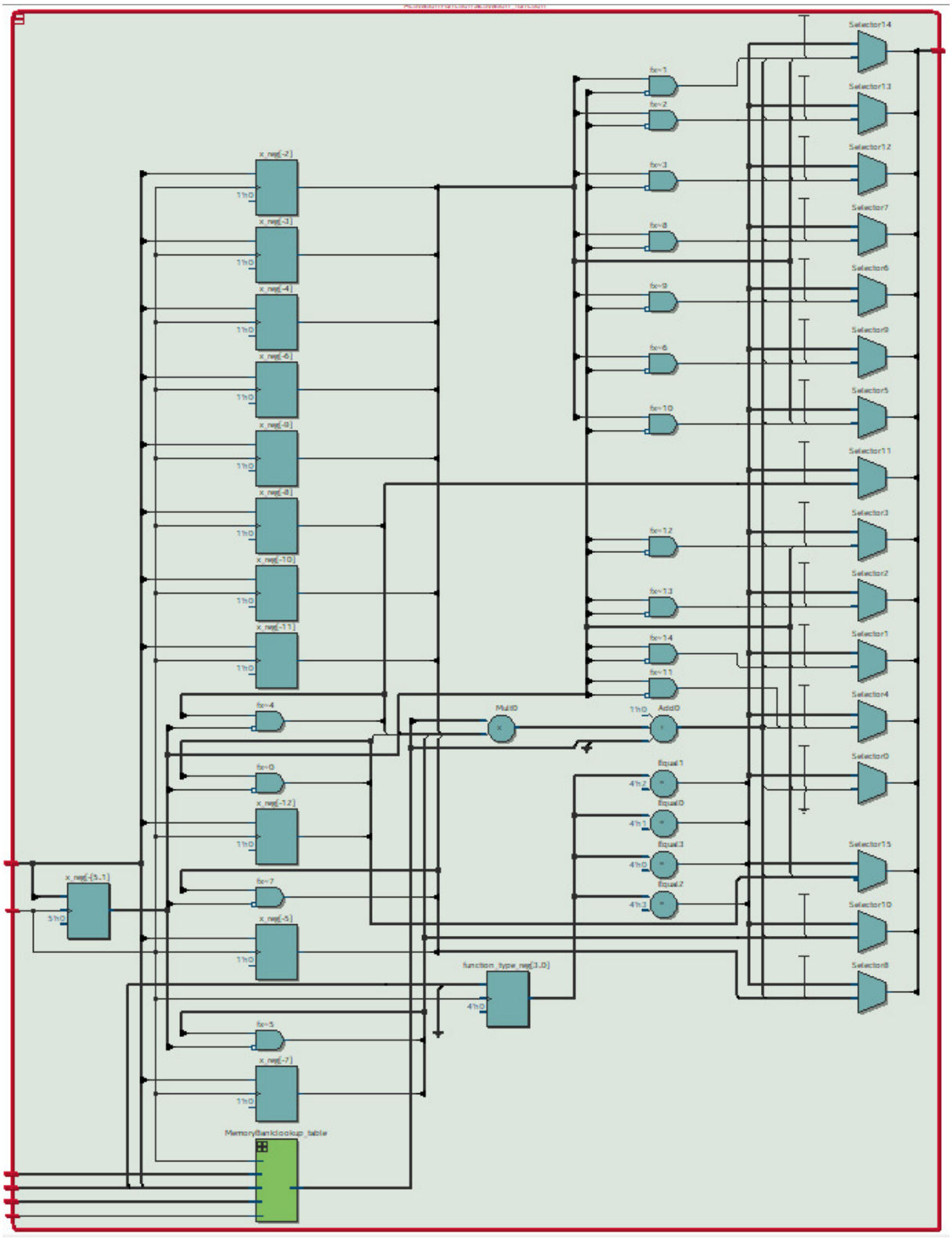
APÊNDICE D – Diagrama RTL do controlador



APÊNDICE E – Diagrama RTL do serializador



APÊNDICE F – Diagrama RTL da função de ativação



INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA
Campus João Pessoa
Av. Primeiro de Maio, 720, Jaguaribe, CEP 58015-435, João Pessoa (PB)
CNPJ: 10.783.898/0002-56 - Telefone: (83) 3612.1200

Documento Digitalizado Restrito

Trabalho de Conclusão de Curso

Assunto: Trabalho de Conclusão de Curso
Assinado por: Victor Herbert
Tipo do Documento: Tese
Situação: Finalizado
Nível de Acesso: Restrito
Hipótese Legal: Informação Pessoal (Art. 31 da Lei no 12.527/2011)
Tipo do Conferência: Cópia Simples

Documento assinado eletronicamente por:

- **Victor Herbert Ferreira de Sousa**, **ALUNO (20181610015) DE BACHARELADO EM ENGENHARIA ELÉTRICA - JOÃO PESSOA**, em 01/03/2023 18:18:40.

Este documento foi armazenado no SUAP em 01/03/2023. Para comprovar sua integridade, faça a leitura do QRCode ao lado ou acesse https://suap.ifpb.edu.br/verificar-documento-externo/ e forneça os dados abaixo:

Código Verificador: 762633
Código de Autenticação: 7f3f7f14e4