

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE
SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELETRÔNICA**

MIKAEL FERRAZ ALDEBRAND

**IMPLEMENTAÇÃO E VALIDAÇÃO DE UM ALGORITMO DE REDE NEURAL
UTILIZANDO UM SOC FPGA**

FLORIANÓPOLIS, 2022.

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE
SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELETRÔNICA**

MIKAEL FERRAZ ALDEBRAND

**IMPLEMENTAÇÃO E VALIDAÇÃO DE UM ALGORITMO DE REDE NEURAL
UTILIZANDO UM SOC FPGA**

Trabalho de Conclusão de Curso submetido
ao Instituto Federal de Educação, Ciência
e Tecnologia de Santa Catarina como parte
dos requisitos para obtenção do título de
Engenheiro em Eletrônica.

Orientador:
Prof. Renan Augusto Starke, Dr. Eng.

FLORIANÓPOLIS, 2022.

Ficha de identificação da obra elaborada pelo autor.

Aldebrand, Mikael

Implementação e validação de um algoritmo de rede neural utilizando um SoC FPGA / Mikael Aldebrand; orientação de Renan Augusto Starke. - Florianópolis, SC, 2022.
70 p.

Trabalho de Conclusão de Curso (TCC) - Instituto Federal de Santa Catarina, Câmpus Florianópolis. Bacharelado em Engenharia Eletrônica. Departamento Acadêmico de Eletrônica.
Inclui Referências.

1. Edge AI. 2. FPGA. 3. SoC. 4. Cyclone V. 5. Inferência. I. Starke, Renan Augusto. II. Instituto Federal de Santa Catarina. III. Implementação e validação de um algoritmo de rede neural utilizando um SoC FPGA.

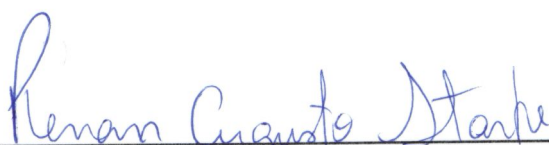
**IMPLEMENTAÇÃO E VALIDAÇÃO DE UM ALGORITMO DE REDE NEURAL
UTILIZANDO UM SOC FPGA**

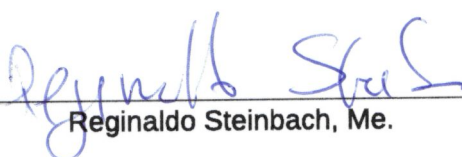
MIKAEL FERRAZ ALDEBRAND

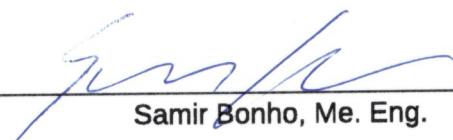
Este trabalho foi julgado adequado para obtenção do título de Engenheiro Eletrônico em dezembro de 2022 e aprovado na sua forma final pela banca examinadora do Curso de Engenharia Eletrônica do Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina.

Florianópolis, 15 de dezembro, 2022.

Banca Examinadora:


Renan Augusto Starke, Dr. Eng.


Reginaldo Steinbach, Me.


Samir Bonho, Me. Eng.

À minha família

AGRADECIMENTOS

A minha família que desde sempre incentivou os estudos e me apoiou durante todo esse tempo.

Ao Instituto Federal de Santa Catarina (IFSC), professores e membros do corpo docente pelo suporte, conhecimento e formação acadêmica fornecidos desde o primeiro dia.

"There is nothing
impossible to him
who will try."
(Alexander, the Great)

RESUMO

A execução de modelos de rede neural mais próximo à geração de dados tem sido cada vez mais procurada devido a preocupações com latência, privacidade e custos relacionados à quantidade de dados transmitidos por sensores, para servidores em nuvem. A implementação do conceito de *edge AI* se mostra um desafio devido à alta carga de processamento envolvido e também às restrições energéticas. Dessa forma, esse trabalho aborda a utilização de FPGAs (*Field-Programmable Gate Array*) como auxiliar a um SoC (*System-on-a-Chip*) em operações de redes neurais, removendo a carga do microprocessador e a delegando para um componente especializado, aumentando a eficiência energética do dispositivo. Ainda, este trabalho apresenta os conceitos necessários para utilizar um SoC-FPGA (Cyclone V), assim como a implementação de um *layer* totalmente conectado de uma rede neural para realizar inferência no FPGA e a comunicação necessária entre o SoC e o FPGA. Para validar o sistema, foi feita uma classificação binária entre dois tipos de flores presentes no *dataset* Iris.

Palavras-chave: Edge AI. FPGA. SoC. Cyclone V. Inferência.

ABSTRACT

The execution of neural network models closer to where the data is produced, known as edge AI, is being desired due to concerns on latency, privacy and costs related to the quantity of data transmitted over sensors to cloud servers. Processing data directly on the device, using AI algorithms, is still a huge challenge due to the high processing load and energy restrictions involved. One possible approach to deal with this complex task is the utilization of FPGAs (Field-Programmable Gate Array). This work addresses the use of FPGAs as an auxiliary to a SoC (System-on-a-Chip) in neural network operations. The aim is to remove the load from the microprocessor and delegate to a specialized component, so increasing the device energy efficiency. It also presents the concepts required to make use of an SoC-FPGA (Cyclone V), as well as an implementation of a fully connected layer to run inference in the FPGA and the required communication between the SoC and the FPGA. In order to validate the system, a binary classification was performed among two types of flowers on the Iris dataset.

Keywords: Edge AI. FPGA. SoC. Cyclone V. Inference.

LISTA DE FIGURAS

Figura 1 – Separação entre Edge, Fog e Cloud	20
Figura 2 – Diagrama de Bloco Intel SoC	21
Figura 3 – Processo de boot	22
Figura 4 – Protocolo de leitura e escrita	24
Figura 5 – Ponto flutuante	25
Figura 6 – Ponto fixo	25
Figura 7 – Perceptron	27
Figura 8 – Funções de ativação	28
Figura 9 – Rede Neural Multilayer	29
Figura 10 – Aplicação	31
Figura 11 – Conexão dos componentes efetuados no <i>Product Designer</i>	32
Figura 12 – Endereços de memória definidos no <i>Product Designer</i>	33
Figura 13 – Máquina de estados	34
Figura 14 – Simulação da máquina de estados no Questa	35
Figura 15 – Simulação da rede neural no Questa	37
Figura 16 – Teste da aplicação	39
Figura 17 – Iris <i>dataset</i>	40
Figura 18 – Setácea vs Versicolor	40
Figura 19 – Modelo usado na classificação	41
Figura 20 – Limiar de decisão	41
Figura 21 – Diagrama de blocos	42
Figura 22 – Fluxograma do processo de inferência	44

LISTA DE QUADROS

Quadro 1 – Espaço de endereçamento de memória	32
Quadro 2 – Sumário de recursos usados	42

LISTA DE TABELAS

Tabela 1 – Recursos disponíveis Cyclone V 5CSXFC6	23
---	----

LISTA DE ABREVIATURAS E SIGLAS

ASIC	<i>Application-Specific Integrated Circuit</i>
AVX	<i>Advanced Vector Extension</i>
CNN	<i>Convolutional Neural Network</i>
DSP	<i>Digital Signal Processor</i>
DT	<i>Devicetree</i>
DTO	<i>Devicetree Overlay</i>
FPGA	<i>Field-Programmable Gate Array</i>
GPU	<i>Graphics Processing Unit</i>
HPS	<i>Hard processor System</i>
I/O	<i>Input/Output</i>
IA	Inteligência Artificial
IFSC	Instituto Federal de Santa Catarina
IoT	<i>Internet of Things</i>
LAB	<i>Logical Array Blocks</i>
MMU	<i>Memory Management Unit</i>
MPU	<i>MicroProcessor Unit</i>
PLL	<i>Phase-Locked Loops</i>
RNN	<i>Recurrent Neural Network</i>
ROM	<i>Read Only Memory</i>
ReLU	<i>Rectified Linear Unit</i>
SD	<i>Secure Digital</i>
SIMD	<i>Single Instruction, Multiple Data</i>
SoC	<i>System-on-a-Chip</i>
TPU	<i>Tensor Processor Unit</i>
VHDL	<i>Very High Speed Integrated Circuits Hardware Description Language</i>

rbf	<i>Raw Binary File</i>
rootfs	<i>Root filesystem</i>
sof	<i>SRAM Object Files</i>

SUMÁRIO

1	INTRODUÇÃO	16
1.1	Justificativa	16
1.2	Definição do Problema	17
1.3	Objetivo Geral	17
1.4	Objetivos Específicos	17
1.5	Estrutura do Trabalho	18
2	FUNDAMENTAÇÃO TEÓRICA	19
2.1	<i>Edge Computing</i>	19
2.1.1	<i>Edge AI</i>	20
2.2	Cyclone V	20
2.2.1	Cortex-A9	21
2.2.1.1	Bootloader	21
2.2.1.2	Kernel	22
2.2.1.3	Devicetree	23
2.3	Malha de FPGA	23
2.4	Barramento Avalon	24
2.5	Sistemas numéricos	24
2.5.1	Ponto flutuante	25
2.5.2	Ponto fixo	25
2.5.2.1	Aritmética	26
2.5.2.2	Notação Q	26
2.6	Perceptron	26
3	DESENVOLVIMENTO	30
3.1	Aplicação	30
3.2	Mapeamento de memória	31
3.3	Implementação das especificações do barramento Avalon	34
3.4	Implementação da rede neural	35
4	RESULTADOS E DISCUSSÃO	38
4.1	Testes	38
4.2	Classificação	39
4.3	Síntese	41
4.4	Programando FPGA	43
4.5	Inferência no SoC-FPGA	43
5	CONSIDERAÇÕES FINAIS	46
5.1	Sugestões para trabalhos futuros	46
	REFERÊNCIAS	47
	APÊNDICES	50
	APÊNDICE A – CÓDIGO VHDL PONTO FIXO	51
	APÊNDICE B – CÓDIGO VHDL DEFINIÇÃO DE TIPOS	54
	APÊNDICE C – CÓDIGO VHDL REDE NEURAL	55
	APÊNDICE D – CÓDIGO VHDL IMPLEMENTAÇÃO DO PROTO- COLO AVALON	57

APÊNDICE E – CÓDIGO C COMUNICAÇÃO COM FPGA	60
APÊNDICE F – CÓDIGO C TESTES	62
APÊNDICE G – CÓDIGO C MAIN	70

1 INTRODUÇÃO

Tem-se visto com frequência notícias de aplicações envolvendo inteligência artificial. De acordo com Lee, Tsung e Wu (2018), a inteligência artificial possibilitou novas tecnologias como carros autônomos e reconhecimento facial. Executar modelos de inteligência artificial (IA) em nuvem é o mais usual até o momento com servidores provendo *hardware* dedicado para reduzir o tempo de processamento. Essa forma de operação restringe algumas aplicações que dependem de decisões rápidas e não podem estar sujeitas a latência da rede. Para que as dificuldades relacionadas a desempenho e a privacidade sejam mitigadas, se faz necessário que a computação aconteça o mais próximo possível da coleta de dados. Mas isso levanta alguns pontos como a capacidade de computação elevada e alta eficiência energética do dispositivo executando o modelo para cumprir com prazos temporais e restrições energéticas rígidos. Esse é um dos motivos que levam ao uso de processadores especializados, ou a utilização de aceleradores de redes neurais profundas.

O treinamento de redes neurais é geralmente realizado em nuvem. Frequentemente utiliza-se GPUs (Graphics Processing Unit) que apresentam maior flexibilidade comparada com FPGAs (Field-Programmable Gate Array) e ASICs (Application-Specific Integrated Circuit). Apesar de suas características, as GPUs geralmente não são adequadas para fazer a inferência de modelos em *edge devices* (dispositivos na ponta) devido ao consumo elevado de energia. Sacrificando a flexibilidade, arquiteturas sob medida são projetadas em ASICs para ter alta eficiência computacional e baixo consumo de energia. Para o treinamento e inferência de modelos em nuvem, GPUs e ASICs são excelentes, mas a necessidade de reprogramação de arquiteturas, baixo consumo energético, e performance, faz com que FPGAs sejam a escolha atraente para *edge computing* (computação na ponta) (VÉSTIAS, 2020).

Para aferir os benefícios de usar um FPGA no contexto de inferência de modelos de rede neural, fez-se o uso de um SoC (*system-on-a-chip*) integrado com um FPGA para possibilitar a execução de operações de redes neurais em hardware dedicado. Esse trabalho tem como característica o uso de duas arquiteturas diferentes, mas complementares, para classificação de dados.

1.1 Justificativa

No passado, a necessidade de desempenho em inúmeras tarefas fez com que processadores de propósito geral fossem complementados por circuitos de propósito específico como DSPs (*Digital Signal Processor*), ASICs e instruções capazes de operar em diferentes dados de uma só vez, denominadas SIMD (*Single Instruction, Multiple Data*) como MMX *Technology*, AVX (*Advanced Vector Extension*) (KUSSWURM,

2014). Recentemente, a popularização de redes neurais profundas tornou o desenvolvimento de circuitos integrados especializados uma área de grande relevância. Conforme os modelos de redes neurais ficam mais complexos, os recursos de energia e tempo de inferência aumentam. Isso levou a Google a produzir seu próprio hardware, chamado TPU (*Tensor Processor Unit*) para treinamento de modelos *Deep Learning* em seus servidores, e, mais recentemente, a empresa disponibilizou uma versão de TPU para equipamentos IoT (*Internet of Things*). Dado que a pesquisa em inteligência artificial leva a inovações muito rápidas, ASICs de propósito específico se tornam ultrapassados em um curto período (SHAHID; MUSHTAQ, 2020).

Devido a rápida inovação no campo de *Machine Learning*, arquiteturas rígidas como ASICs podem se tornar obsoletas rapidamente, inviabilizando o investimento de circuitos integrados especializados para otimizar modelos de redes neurais específicos. A importância desse trabalho está em explorar o uso de FPGAs como complemento nos já familiares processadores, assim, com a versatilidade de CPUs e a reprogramabilidade de FPGAs novas soluções em IoT podem ser desenvolvidas e atualizadas com o passar do tempo. Tão relevante quanto a própria aplicação, está a documentação desse conhecimento adquirido para futuros trabalhos envolvendo conceitos similares.

1.2 Definição do Problema

O uso de FPGAs como acelerador de operações específicas de redes neurais é o assunto mais pesquisado no campo de aceleradores. As FPGAs possuem características interessantes, como alto desempenho a um baixo consumo energético, alto grau de paralelização e flexibilidade. Infelizmente, devido à complexidade de descrever hardware em código, faz com que aplicações que envolvam aceleradores de hardware sejam populares apenas no campo da pesquisa (WANG *et al.*, 2019).

Dado que um dos entraves na adoção de soluções envolvendo FPGAs no mercado é a dificuldade técnica apresentada, esse questiona: é possível desenvolver um *hardware* dedicado em FPGA como auxiliar de um processador em operações de redes neurais?

1.3 Objetivo Geral

Implementar um *fully connected layer* (*layer* totalmente conectado) e usar como auxiliar de um microprocessador em aplicações de redes neurais.

1.4 Objetivos Específicos

O objetivo pode ser decomposto em objetivos específicos menores, sendo:

- a) desenvolver, simular e sintetizar um *layer* totalmente conectado no FPGA;
- b) verificar se o componente em *hardware* retorna resultados corretos;
- c) usar o componente para classificação binária de classes.

1.5 Estrutura do Trabalho

No próximo capítulo será apresentado o referencial teórico utilizado para o desenvolvimento do trabalho. Posteriormente, serão apresentadas as ferramentas usadas na execução do trabalho. Então, no capítulo quatro, serão relatados o processo e os resultados alcançados. Por fim, serão ditas as considerações finais.

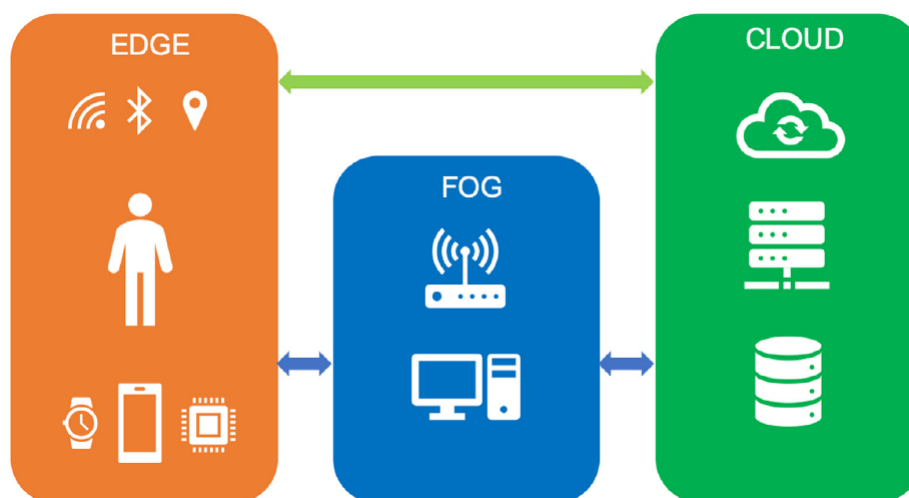
2 FUNDAMENTAÇÃO TEÓRICA

Diferentes conceitos são necessários para permitir a utilização de SoC-FPGA no contexto de *edge AI*. Neste capítulo será introduzido algumas definições de *edge computing* e *edge AI*, assim como as especificações e funcionamento do microprocessador e FPGA usado. Alguns outros pontos como o protocolo de comunicação entre o SoC-FPGA e o uso de ponto fixo ao invés de ponto flutuante serão apresentados. E por fim, a estrutura de um *layer* totalmente conectado, futuramente usada para inferência, é explicada aqui.

2.1 *Edge Computing*

O paradigma computacional predominante é o da computação em nuvem, que tipicamente executa todo o processamento em servidores com alta capacidade de computação e altas taxas de transferência de dados. Ao final, os dados processados são reenviados para os dispositivos que os requisitaram. Com o surgimento e a proliferação da IoT, assim como a popularização de *smartphones*, o número de dispositivos conectados aumentou significativamente, causando um aumento considerável na infraestrutura, com muitos dados gerados sendo enviados pela rede e recursos computacionais usados para computar os dados. A *edge computing* pode mitigar certa carga dos servidores em nuvem fazendo pré-processamento antes de fazer *upload*. Um exemplo é, ao invés de enviar os dados brutos como vídeos ou fotos, realiza-se compactação e ajuste de resolução no dispositivo, assim, otimizam-se recursos de rede e de processamento do servidor (SHI *et al.*, 2016).

Como explicado por Bierzynski, Escobar e Eberl (2017), para fazer completo uso da IoT é necessário a cooperação entre *cloud*, *fog* e *edge*. A separação desses três conceitos pode ser visualizada na Figura 1. *Edge*, ou *edge devices* é o campo dos sensores, microcontroladores, dispositivos portáteis ou móveis como *wearables* e *smartphones*. *Fog* ou traduzindo "névoa", é o meio entre os dispositivos e a nuvem, são equipamentos como roteadores e servidores na rede local. E *cloud* são servidores na internet com grande capacidade de armazenamento e computação.

Figura 1 – Separação entre Edge, Fog e Cloud

Fonte: Adaptado de Greco et al. (2020, p. 350).

2.1.1 Edge AI

Edge AI ou *edge intelligence* é usualmente referido como a execução de algoritmos de inteligência artificial em *edge devices*. O interesse de colocar modelos de inteligência artificial em *edge devices* tem relação com latência, custos e privacidade. Muitos desafios devem ser superados para ser possível fazer a inferência de forma local, ou híbrida, entre servidores e dispositivos locais. Técnicas como a redução do tamanho de modelos, otimização para problemas específicos, *cache* de resultados passados, entre outros, são maneiras de possibilitar o conceito de *edge AI* (ZHOU et al., 2019).

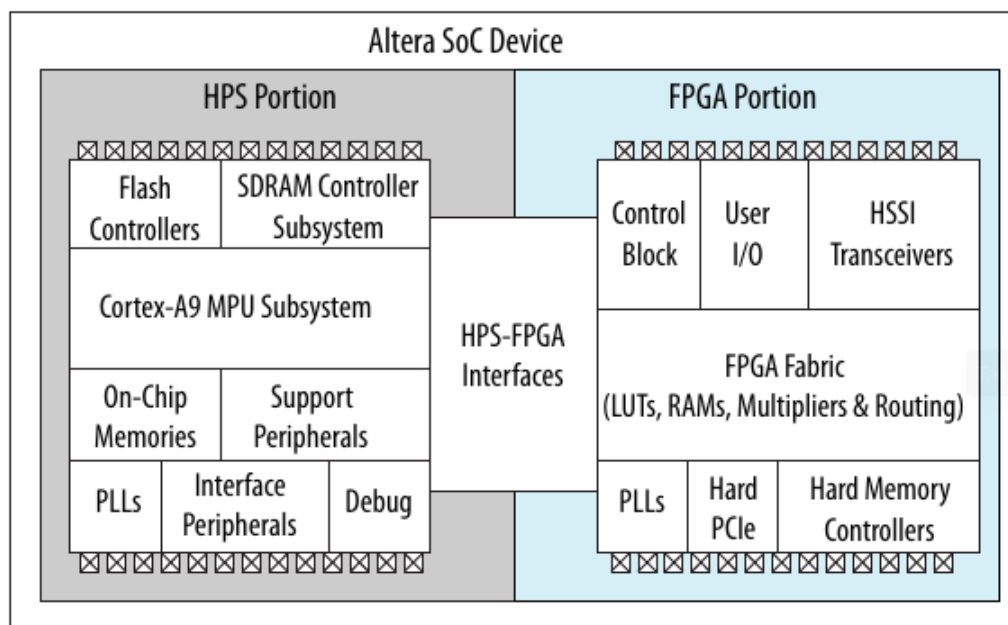
De acordo com Véstias (2020), GPUs podem ser usadas para executar qualquer modelo, entretanto, não são energeticamente viáveis para dispositivos IoT, mesmo GPUs embarcadas como a Jetson TX2 consomem de cerca de 15 W. Assim, para conseguir processar modelos de redes neurais profundas na ponta existem ASICs e FPGAs, sendo que o ciclo de desenvolvimento de um acelerador em FPGA é mais rápido que o desenvolvimento de um ASIC.

2.2 Cyclone V

O SoC Cyclone V é de propriedade intelectual da Intel e dispõe de um processador de dois núcleos Cortex-A9 e uma malha de FPGA no mesmo encapsulamento. Cada dispositivo tem sob seu controle, periféricos e pinos de entrada e saída dedicadas como se pode constatar na Figura 2. A parte em azul é a porção de recursos do FPGA, tem-se a própria malha programável com multiplicadores e memórias, assim como terminais de entrada e saída e controladores de memória. No outro bloco em cinza, é a parte do hps (*Hard processor System*), que é um processador Cortex-A9 de dois

núcleos com periféricos, controladores de memória, componentes para *debug*. E A comunicação entre o processador e o FPGA é feita por um barramento que conecta os dois (INTEL, 2018b).

Figura 2 – Diagrama de Bloco Intel SoC



Fonte: Intel (2018b, p. 2-1).

2.2.1 Cortex-A9

O Cortex-A9 é um microprocessador *dual core* de 32 bits com cache L2, *watchdog*, MMU (*Memory Management Unit*) e outros componentes. Dentro do SoC-FPGA esse MPU (*MicroProcessor Unit*) é denominado HPS (*Hard Processor System*), é um componente fixo, não é possível reconfigurar o hardware da mesma maneira que a malha reprogramável (INTEL, 2018b). Este microprocessador é relativamente complexo e necessita de um *bootloader* e um sistema operacional. Tais componentes de software são descritos a seguir.

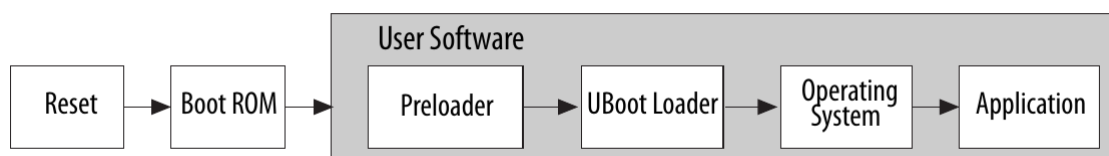
2.2.1.1 Bootloader

Um *bootloader* é o software responsável pela configuração básica dos periféricos do processador para permitir que um sistema operacional possa ser executado. No momento que o dispositivo é inicializado, dispositivos de armazenamento como NAND e o controlador de memória DRAM estão desconfigurados e não podem ser usados. Uma das prioridades do *bootloader* é fazer a memória principal funcionar para carregar o *kernel*. Os detalhes de quais componentes são inicializados e onde

o inicializador deve estar é específico de cada fabricante, mas o objetivo é o mesmo: configurar o necessário e passar o controle para o *kernel* (SIMMONDS, 2017).

No Cyclone V a inicialização do sistema pode acontecer de diferentes formas, inclusive com o FPGA inicializando o HPS. Nesse trabalho escolheu-se o processo mais convencional de inicialização do sistema que é o da Figura 3.

Figura 3 – Processo de boot



Fonte: Intel (2018b, p. A-1).

O boot ROM (*Read Only Memory*) é um inicializador do próprio microprocessador. Ele procura o pre-inicializador (*preloader*), podendo ser FPGA, memória NAND, cartão SD (*Secure Digital*). Por familiaridade com o processo de gravação, optou-se armazenar todos os componentes em um cartão SD.

2.2.1.2 Kernel

A parte central em um sistema operacional é denominado *kernel*, ele é responsável pelo gerenciamento dos recursos e a comunicação com o hardware. Aplicações executando no dispositivo devem se comunicar com o *kernel* para terem acesso ao hardware (memória e periféricos), ou se comunicarem com outros processos. O *kernel* também decide quanto do recurso do processador o programa pode usar e quando ele pode executar, assim como tratar interrupções recebidas pelo hardware (LOVE, 2010).

Optou-se por usar o *kernel* Linux, a versão do *kernel* usada foi a 5.16 do *fork* da Altera, agora Intel. A versão usada possui interfaces de abstração para FPGAs (ENNO *et al.*, 2021). O suporte pode ser habilitado nas configurações do *kernel*, o *defconfig* do repositório da altera para o kit de desenvolvimento usado já habilita por padrão. Dado que a aplicação desenvolvida não é um módulo do *kernel* mas usa chamadas do sistema para ter acesso ao componente em memória, é possível atualizar o *kernel* sem afetar o funcionamento do programa.

Após o *kernel* inicializar os dispositivos é montado um sistema chamado de *root filesystem* (sistema de arquivo raiz) ou mais conhecido como "rootfs", conforme A Fundação Linux (2015, tradução nossa) "O conteúdo do *root filesystem* deve ser adequado para iniciar, restaurar e/ou reparar o sistema." O rootfs usado foi o debian buster que foi gerado com debootstrap e qemu-arm.

2.2.1.3 Devicetree

Uma *devicetree* (DT) é uma estrutura de dados com a descrição do *hardware* presente, o *kernel* usa esse arquivo para carregar o *driver* correspondente de cada componente, não sendo necessário que o código do dispositivo esteja programado no próprio *kernel* (LIKELY, 2021).

É possível modificar o *hardware* visto pelo *kernel* dinamicamente com o uso de *devicetree overlay*, assim dispositivos podem ser inseridos e retirados enquanto o *kernel* está operando (THE KERNEL DEVELOPMENT COMMUNITY, 2022). Foi necessário habilitar as pontes entre o SoC e o FPGA na *devicetree* e usou-se o *devicetree overlay* para programar o FPGA.

2.3 Malha de FPGA

Segundo Intel (2020), o SoC tem diferentes componentes que compõem a parte do FPGA. Um deles é o LAB (Logical Array Blocks) isso é, arranjo de blocos lógicos. São elementos configuráveis que em conjunto podem ser usados para diversas implementações lógicas. Além dos blocos reprogramáveis, há elementos com finalidades específicas como os blocos de memória M10K que são memórias de 10 Kb dedicadas, DSPs especializados em multiplicação, PLLs (*Phase-Locked Loops*), interfaces para memórias externas e recursos I/O. O SoC-FPGA usado foi o Cyclone V 5CSXFC6, a Tabela 1 mostra a quantidade de recursos disponíveis desse modelo.

Tabela 1 – Recursos disponíveis Cyclone V 5CSXFC6

5CSXFC6	
Logic Elements (LE) (K)	110
ALM	41,910
Register	166,036
Block Memory (Kb)	5,570
MLAB Memory (Kb)	621
Variable-precision DSP Block	112
18 x 19 Multiplier	224
FPGA PLL	6
HPS PLL	3
3-Gbps Transceiver	9
FPGA GPIO	288
HPS I/O	181
LVDS Transmitter	72
LVDS Receiver	72
PCIe Hard IP Block	2
FPGA Memory Controller	1
HPS Memory Controller	1
ARM Cortex-A9 MPCore Processor	Dual-core

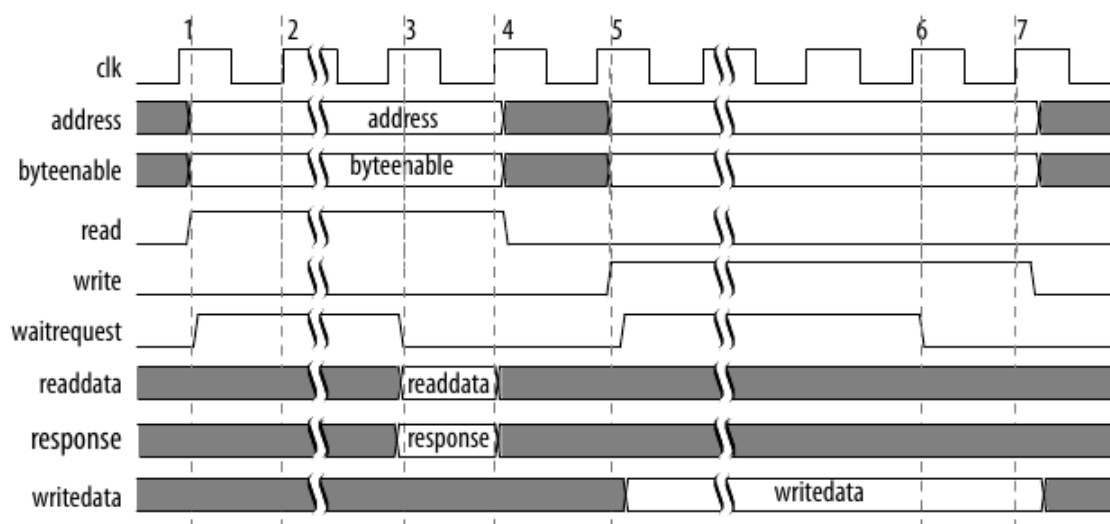
Fonte: Intel (2018a, p. 14)

2.4 Barramento Avalon

O Avalon é um barramento de comunicação desenvolvido pela Intel para permitir o fluxo de dados entre diferentes componentes, isso inclui o *streaming* de pacotes, enviar/receber sinais e interrupções, conexão com circuitos integrados fora do SoC-FPGA entre outros (INTEL, 2022).

Para enviar os dados do processador para o componente e enviar o resultado do componente para o processador, foi necessário seguir as especificações da memória mapeada (Avalon-MM). A Figura 4 mostra as especificações que o componente deve suportar para enviar e receber os dados. Os sinais de *read*, *write* e *writedata* são ditados pelo *master* (no caso o processador SoC) e os sinais de *readdata* e *waitrequest* são controlados pelo *slave* (nesse trabalho ele é o componente em FPGA). Quando o *master* sinaliza uma operação de envio (escrita) de dados para o *slave* com o sinal *write*, o *slave* pode solicitar para que o dado seja mantido na entrada enquanto for necessário com *waitrequest*, o sinal de *write* é mantido alto até *waitrequest* tornar-se baixo. Quando *waitrequest* for de-sinalizado ou não for sinalizado, o *slave* deve ler os dados enviados pelo *master* no mesmo ciclo de *clock*. Similarmente, o mesmo acontece com a operação de leitura do *slave* pelo *master* com a sinalização do sinal *read*. Nem todos os sinais precisam estar explicitamente declarados no componente, a ferramenta *Platform Designer* (INTEL, 2021b) abstrai os detalhes para o desenvolvedor.

Figura 4 – Protocolo de leitura e escrita



Fonte: Intel (2022, p. 21).

2.5 Sistemas numéricos

Assim como o sistema decimal, o sistema binário é posicional. Um sistema posicional é definido pela regra da Equação 1

$$(\dots a_3 a_2 a_1 a_0 . a_{-1} a_{-2} \dots)_b = \dots + a_3 b^3 + a_2 b^2 + a_1 b^1 + a_0 + a_{-1} b^{-1} + a_{-2} b^{-2} + \dots; \quad (1)$$

Os dígitos mais a esquerda a_k são ditos mais significativos para 'k's maiores, e menos significativos para 'k's menores. O subscrito 'b' é denominado base, o ponto entre o a_0 e o a_{-1} é denominado ponto decimal na base 10 e algumas vezes é referido como ponto binário em base 2 (KNUTH, 1981).

Segundo Knuth (1981) outro sistema chamado ponto flutuante é mais conveniente para algumas circunstâncias, onde é necessário expressar grandezas muito grandes ou muito pequenas. A diferença para o ponto fixo é que o "ponto" é dinâmico. Esse formato é usado com frequência em cálculos científicos.

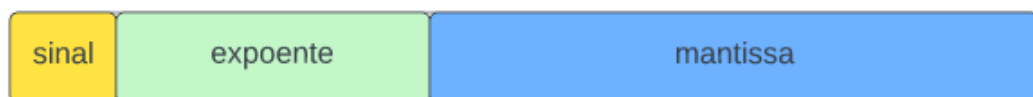
2.5.1 Ponto flutuante

A norma IEEE-754 define um padrão para ponto flutuante do tipo básico, ou precisão simples, e ponto flutuante de precisão dupla. A diferença entre os dois é a quantidade de números possíveis. A Equação 2 mostra o formato do ponto flutuante.

$$(-1)^s 2^E (b_0 . b_1 b_2 \dots b_{p-1}) \quad (2)$$

Onde s é o sinal, 0 é positivo e 1 é negativo, E é o expoente, p é a precisão. A representação disso em memória pode ser visto na Figura 5.

Figura 5 – Ponto flutuante

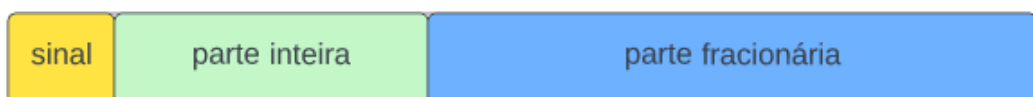


Fonte: Elaboração própria.

2.5.2 Ponto fixo

O ponto fixo é mais familiar, é dividido em parte inteira e parte fracionária, com um bit para sinal. Como pode ser visto na Figura 6

Figura 6 – Ponto fixo



Fonte: Elaboração própria.

Em comparação com o ponto flutuante, em FPGA, o ponto fixo usa menos recursos lógicos, é mais rápido, e necessita de menos energia (FINNERTY; RATIGNER, 2017). Nesse trabalho foi usado ponto fixo para a classificação.

2.5.2.1 Aritmética

As operações de soma, subtração, divisão e multiplicação são feitas como números inteiros, é responsabilidade do programador escalar o resultado para o valor correto. O fator de escala normalmente é implícito. Para distinguir as diferentes representações, é uma convenção usar a notação Q.

2.5.2.2 Notação Q

Como o hardware faz operação usando números inteiros, a diferença está apenas na interpretação do resultado. Como pode ter várias formas de interpretar um número binário, isso é, a quantidade de *bits* que representam a parte inteira e a parte fracionária, engenheiros usam a notação Q para evitar confusão. Assim, Q1.15 explicita que apenas 1 *bit* é usado para a parte inteira, e 15 *bits* é para mantissa. A conversão de decimal para ponto fixo é feita multiplicando um número fracionário pela quantidade de *bits* da mantissa e truncar ou arredondar o resultado inteiro, a Equação 3 é um exemplo com 15 *bits* na mantissa.

$$q_{1_15} = decimal \cdot 2^{15} \quad (3)$$

Tomando o número decimal 1, 14321780 como exemplo, seu equivalente em q_{1_15} será 37.460. E para reverter a conversão se divide o número inteiro pela quantidade de bits da mantissa como na Equação 4 (PADGETT; ANDERSON, 2009).

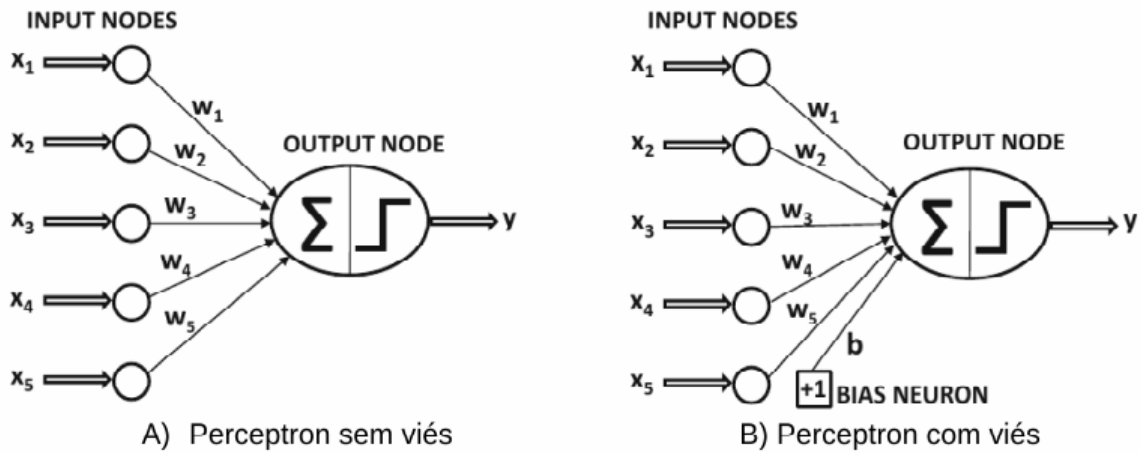
$$decimal = \frac{q_{1_15}}{2^{15}} \quad (4)$$

Revertendo o resultado anterior de q_{1_15} para decimal, o número 37.460 tem o equivalente 1, 14318847 em decimal.

2.6 Perceptron

A estrutura mais básica de uma rede neural artificial é chamada *perceptron*. Inspirada no *design* biológico do neurônio, produz uma saída dependente do produto das entradas com os pesos. O modelo do *perceptron* com e sem viés (*bias*) pode ser visualizado na Figura 7. Nota-se que são duas operações que produzem a saída: a primeira chamada de valor pré-ativação (multiplicação e soma), e a segunda denominada valor pós-ativação é o valor produzido após a função de ativação (AGGARWAL, 2018).

Figura 7 – Perceptron



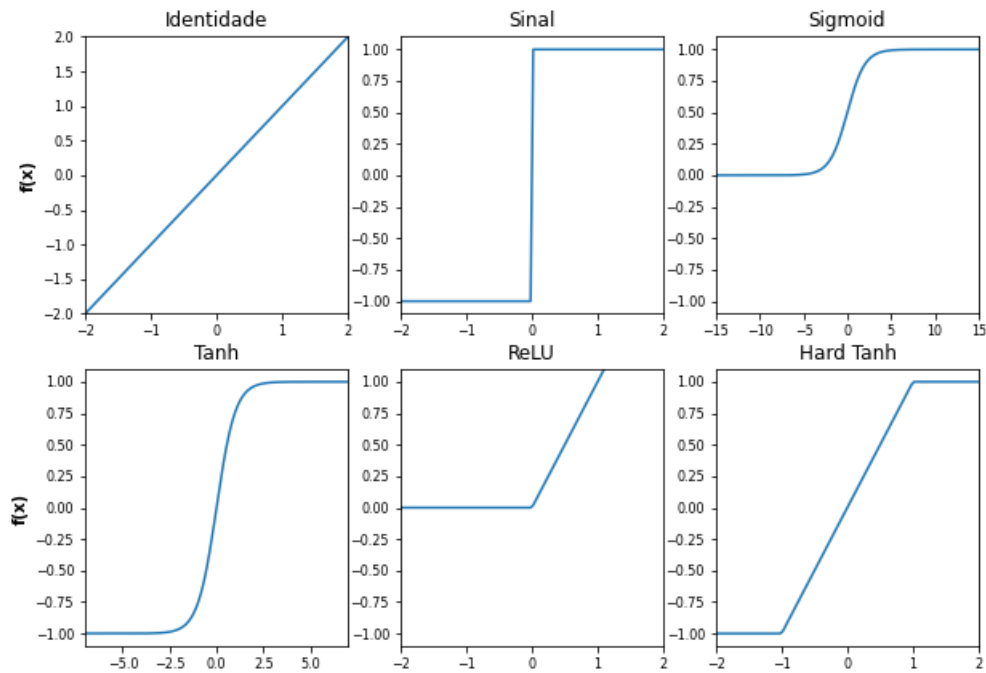
Fonte: Adaptado de Aggarwal (2018, p. 5).

Para classificações binárias, isso é para dizer que algo pertence a uma classe ou outra, o modelo do *perceptron* recorre a uma função, que retorna 1 caso o valor do resultado pré-ativação for maior ou igual a determinado limiar, e -1 caso o valor seja menor que o limiar. A Equação 5 é um modelo do *perceptron* sem viés (*bias*). Onde ϕ é a função de ativação, w são os pesos do *perceptron* e x são as entradas, a operação pode ser vista de duas maneiras, como um produto de dois vetores, ou como uma multiplicação e soma feita nas entradas e pesos, onde m é o número de entradas do *perceptron* (RASCHKA; MIRJALILI, 2017).

$$\hat{y} = \phi(\bar{w}^T \cdot \bar{x}) = \phi\left(\sum_{j=1}^m w_j x_j\right) \quad (5)$$

De acordo com Aggarwal (2018), o valor do perceptron é -1 ou 1 , útil apenas para classificações binárias, não tanto para outras aplicações onde o que precisa ser predito é uma variável contínua e não binária, para isso é usado outras funções de ativação não lineares como sigmoide, tangente hiperbólica, unidade linear retificada ou ReLU (Rectified Linear Unit), e outras. Alguns exemplos de funções de ativação podem ser visualizados na Figura 8.

Figura 8 – Funções de ativação



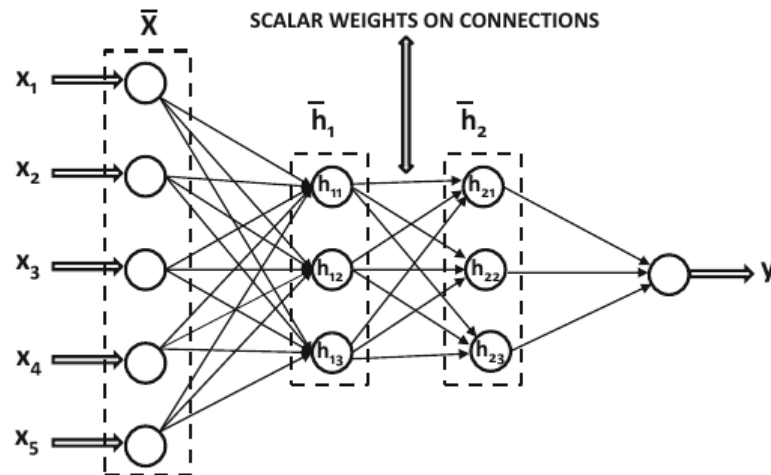
Fonte: Adaptado de Aggarwal (2018, p. 13).

Ainda segundo Aggarwal (2018), quando vários perceptrons são arranjados para formar um *layer* a função de ativação é computada para cada saída do *layer* individualmente. Quando vários *layers* são postos em conjunto, eles formam uma rede neural *multilayer* como exemplificado na Figura 9. As conexões entre os *layers* da rede neural *multilayer* podem ser representadas como matrizes e as saídas de cada *layer* como vetor, a Equação 6 mostra a operação que produz o resultado do primeiro *layer* da rede neural *multilayer* ilustrada na Figura 9. Aqui $\bar{W}^T$ é uma matriz com os pesos do primeiro *layer* oculto, os valores são obtidos após a multiplicação matriz vetor, e a função de ativação é aplicada individualmente a cada resultado. Então $\bar{h}_1$ se torna a entrada do próximo *layer*.

$$\bar{h}_1 = \phi(\bar{W}_1^T \bar{x}) \quad (6)$$

A Figura 9 é uma arquitetura com três *layers*, ela é do tipo *feed-forward* com dois *layers* ocultos e um *layer* de saída, normalmente o *layer* de entrada não conta com um *layer* visto que não é feito nenhum cálculo nele (AGGARWAL, 2018).

Figura 9 – Rede Neural Multilayer



Fonte: Aggarwal (2018, p. 18).

Tanto em pesquisas quanto em código de produção, existem *frameworks* consolidados de *deep learning* como Tensor Flow, Pytorch, Caffe entre outros, que já implementam os tipos mais comuns de modelos de rede neural como CNNs, RNNs, Transformers, e além. Essas ferramentas são usadas e testadas por várias pessoas e aceleram o desenvolvimento de soluções envolvendo redes neurais como tradução de texto, voz para texto, reconhecimento de imagens (INTERNATIONAL BUSINESS MACHINES CORPORATION – IBM, 2021). Nesse trabalho, utilizou-se o Pytorch para validar a implementação do *layer* em hardware, e o scikit-learn (SCIKIT-LEARN, 2022), para treinar um perceptron para classificação.

3 DESENVOLVIMENTO

Nesse capítulo apresentam-se as ferramentas e métodos usados para configurar o SoC. Assim como a programação, a simulação, e a síntese do componente desenvolvido. E os testes de integração entre o SoC e FPGA.

Para a configuração e utilização do SoC, foram usados o *bootloader* U-boot e o Linux *kernel* mantidos pela Intel.

O núcleo da operação executada no *layer* totalmente conectado foi descrito em VHDL (*Very High Speed Integrated Circuits Hardware Description Language*) e baseado na implementação de Pedroni (2004), a interface de comunicação com o processador foi feita seguindo as especificações de comunicação Avalon.

Para verificar a implementação do componente, isso é, seu comportamento a diferentes estímulos, foi realizada uma simulação no *software* Questa (INTEL, 2021c) *Starter Edition* disponibilizado pela Intel. O *tesbench* foi utilizado para verificar se o componente segue as especificações do protocolo Avalon e se os resultados gerados pelo componente estavam corretos antes de fazer a síntese e testar a implementação no *hardware*.

A ferramenta *Platform Designer* foi utilizado para conectar a componente com o microprocessador Cortex-A9 e gerar os arquivos necessários para usar o barramento Avalon.

Para gerar a síntese do componente foi usado o *software* QuartusII Prime Lite Edition (INTEL, 2021a) de propriedade da Intel, escolhido devido à plataforma de desenvolvimento utilizada que possui um SoC-FPGA da Intel, portanto as ferramentas usadas para gerar o binário foram do próprio fabricante.

A plataforma de desenvolvimento usada foi o Arrow SoC da Terasic (TERASIC, 2014) de propriedade do IFSC.

Para validar as operações em *hardware*, se usou o Pytorch para comparar o resultado do *hardware* com o do *Layer* totalmente conectado da ferramenta.

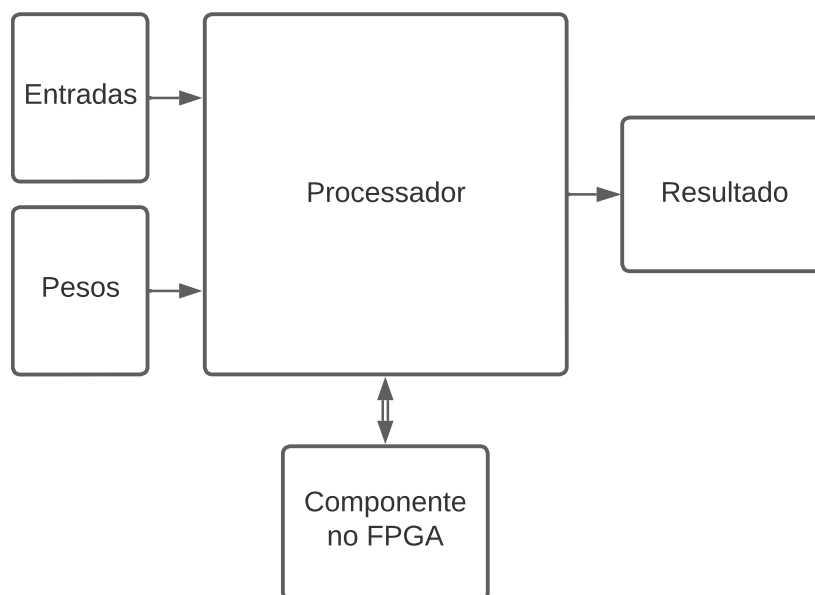
Foi usado o *framework* *scikit-learn* para treinar um perceptron em duas classes no *dataset* Iris para obter os parâmetros do modelo, esses pesos foram usados para fazer a inferência no FPGA, e o resultado da classificação foi comparado com o produzido pelo *framework*.

3.1 Aplicação

Para usar o componente foi escrito um programa em C para se comunicar com o componente no FPGA. Os pesos do *perceptron* e as entradas são passados para o executável por arquivos. A Figura 10 mostra o funcionamento do sistema: o

programa faz a leitura e envia para o *layer* os pesos, assim como as entradas para fazer a inferência. Quando o resultado estiver pronto, o programa salva o resultado em um arquivo de saída.

Figura 10 – Aplicação

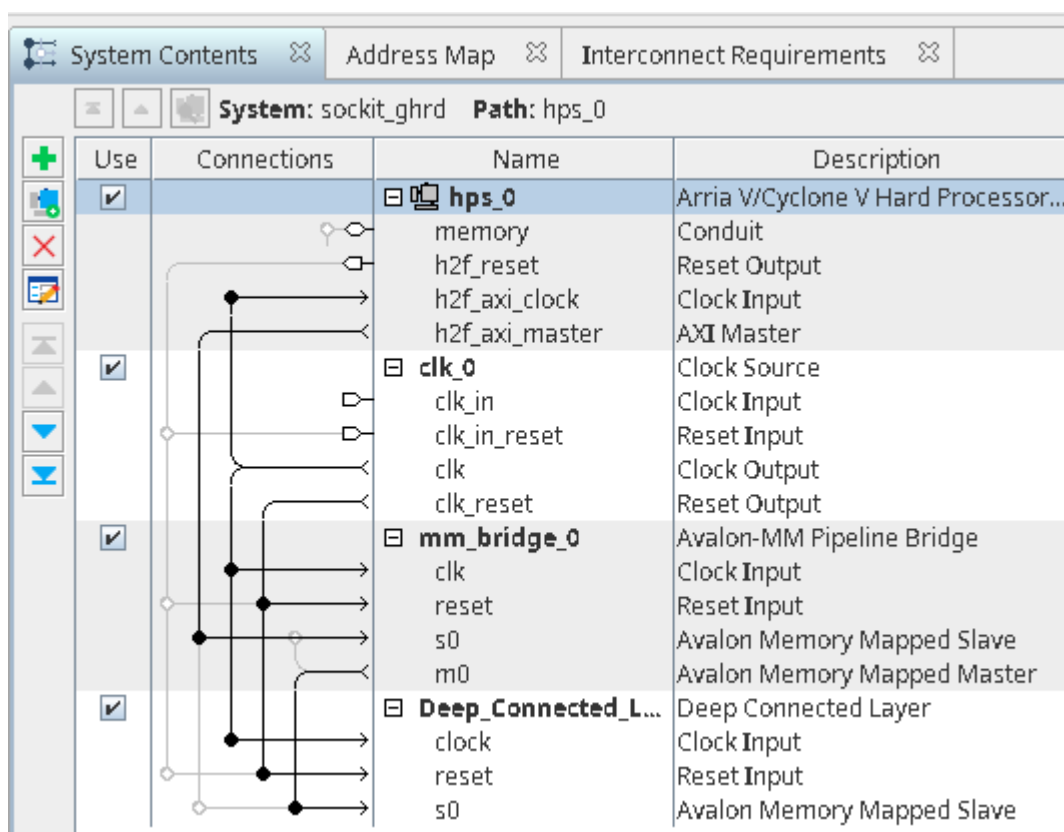


Fonte: Elaboração própria.

O código de leitura e escrita do componente fez uso de um laço e ponteiros para enviar e receber dados do endereço de memória do componente sintetizado na FPGA. O mapeamento de memória é descrito na próxima seção.

3.2 Mapeamento de memória

O endereço de memória do componente é definido no *Platform Designer*. A Figura 11 mostra como os componentes são interligados: há o *clock* do sistema compartilhado entre todos, o componente *Avalon-MM Pipeline Bridge* (mm_bridge_0) é responsável por conectar o Cortex-A9, na figura ele é denominado “hps_0”, com o componente desenvolvido no trabalho, denominado “Deep_Connected_Layer”.

Figura 11 – Conexão dos componentes efetuados no *Product Designer*

Fonte: Gerado a partir de Intel (2021a).

O endereço é dependente da ponte usada para conectar o processador com o componente, sendo que existem três pontes que fazem essa comunicação entre o FPGA e o SoC. Duas estão sob o controle do SoC, e a outra é controlada pelo FPGA. A que está sob o controle do FPGA é usada para acessar periféricos que tipicamente pertencem ao processador. Os endereços das pontes que pertencem ao processador estão na Tabela 1

Quadro 1 – Espaço de endereçamento de memória

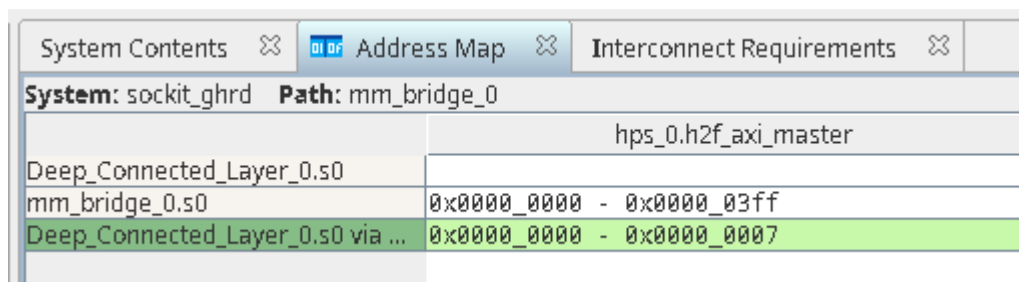
HPS-to-FPGA	0xC0000000
Lightweight HPS-to-FPGA	0xFF200000

Fonte: Adaptado de Intel (2018b, p. 2-16).

Como esse é o único componente instanciado no *Platform Designer*, e como foi usado a ponte HPS-to-FPGA para tornar o componente acessível ao processador, o endereço do componente é o da própria ponte HPS-to-FPGA, isso é, 0xC0000000. O número de *bytes* necessários para escrever e ler do componente é definido no tamanho da palavra usada. A ponte permite comunicação com 32, 64, e 128 bits, e isso define qual é a quantidade de *bytes* necessários para mapear o componente. A Figura 12 mostra quantos *bytes* pertencem ao dispositivo. No caso, “mm_bridge_0”

tem uma faixa de endereço de 1024 *bytes* sob sua responsabilidade, o componente desenvolvido tem 8 *bytes* desses 1024 *bytes*. Como não foi usada toda a faixa de endereço do “mm_bridge_0” é possível instanciar outros componentes até toda a faixa de endereçamento ser usada. Para se comunicar com o componente é usado um tamanho da palavra de 8 *bytes* (64 *bits*).

Figura 12 – Endereços de memória definidos no *Product Designer*



System: sockit_ghrd Path: mm_bridge_0	
	hps_0.h2f_axi_master
Deep_Connected_Layer_0.s0	
mm_bridge_0.s0	0x0000_0000 - 0x0000_03ff
Deep_Connected_Layer_0.s0 via ...	0x0000_0000 - 0x0000_0007

Fonte: Gerado a partir de Intel (2021a).

O componente em FPGA não está disponível de imediato para o programa no espaço de execução do programa em C. Para poder enviar e receber dados ao dispositivo, é necessário pedir ao kernel acesso ao endereço virtual da localização do dispositivo em memória. Segundo a documentação do *kernel* Linux, isso pode ser realizado com a função ‘mmap’, passando como argumento o endereço do dispositivo definido no *Platform Designer* com o deslocamento (*offset*) da ponte usada para ligar o SoC com o FPGA. Assim se pode enviar e ler dados como operação usuais de ponteiros. Segue um trecho do código usado para mapear o componente com as funções usadas e com o endereço obtido no Quadro 1. O código completo encontra-se no apêndice E.

```

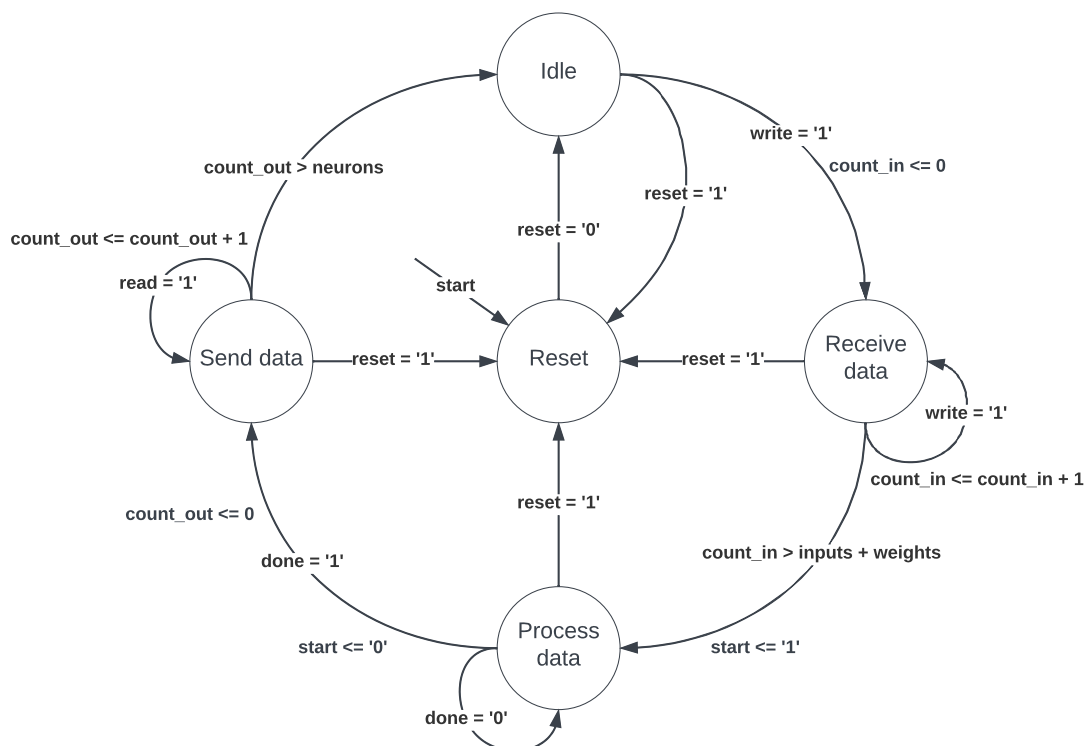
0 #define HPS_TO_FPGA_BRIDGE 0xC0000000
1 #define HPS_TO_FPGA_BRIDGE_SPAN 0x07
2
3 static uint8_t *bridge = NULL;
4
5 int map_component(void) {
6     fd = open("/dev/mem", O_RDWR | O_SYNC);
7
8     bridge = mmap(NULL, HPS_TO_FPGA_BRIDGE_SPAN, PROT_READ | PROT_WRITE,
9                  MAP_SHARED, fd, HPS_TO_FPGA_BRIDGE);
10
11     return 0;
12 }

```

3.3 Implementação das especificações do barramento Avalon

Para cumprir com as especificações do protocolo explicitados na Figura 4, uma máquina de estado foi projetada. Ela é responsável por armazenar os dados e passar para o componente, assim como enviar os valores do componente para quem estiver requisitando. O processo executado pela máquina de estado pode ser visualizado na Figura 13. O estado inicial é o de *Reset* que deixa todos os sinais internos do componente em valores conhecidos. Após o sinal de “reset” deixar o estado alto, o estado *Idle* é alcançado. Aqui o componente espera por um sinal de escrita “write” para receber dados. Enquanto a quantidade de dados recebidos for menor que a quantidade de entradas somada a quantidade de parâmetros do modelo, a máquina de estado permanece nesse estado incrementando o contador. Assim que o contador atinge o valor equivalente a soma das entradas com os parâmetros, o sinal de “start” sinaliza para o componente que os dados foram recebidos e que a operação pode ser executada. O estado *Process data* realiza o cálculo dos valores e quando a computação estiver pronta (indicado pelo sinal de “done”), o componente entra do estado de “*Send data*” e permanece ali enquanto a quantidade de dados enviados for inferior ao número de neurônios do *layer*. O processamento termina atingindo-se novamente o estado “*Idle*”. A qualquer momento “reset” pode ser voltar para o estado alto, levando automaticamente a máquina de estado para o estado de *Reset*.

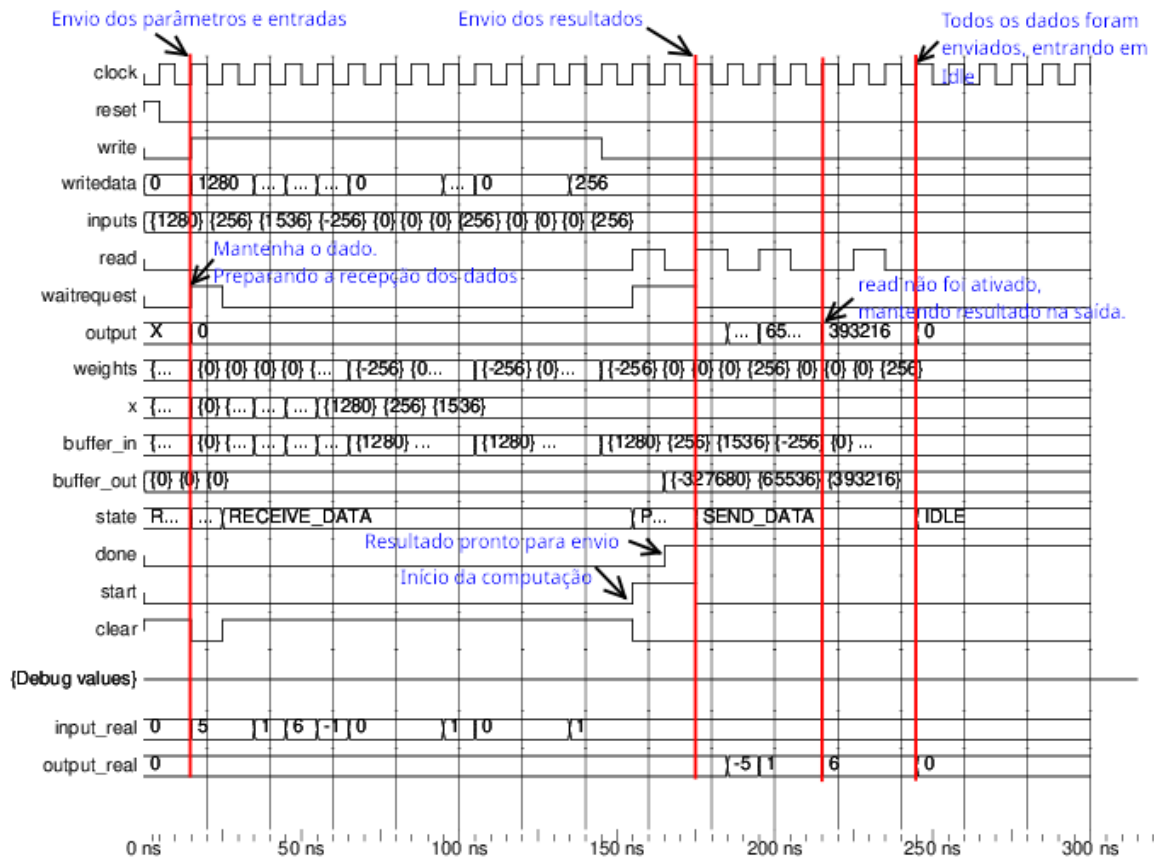
Figura 13 – Máquina de estados



Fonte: Elaboração própria.

A máquina de estado foi implementada em VHDL e a simulação do comportamento pode ser observada na Figura 14. Vale ressaltar que as entradas são em ponto fixo e os valores que os números representam estão nos sinais de “*input_real*” e “*output_real*”.

Figura 14 – Simulação da máquina de estados no Questa



Fonte: Elaboração própria.

3.4 Implementação da rede neural

A implementação da rede neural foi baseada no livro do Pedroni (2004). O código em VHDL descreve a multiplicação e soma conforme a Equação 5 (repetida aqui), é o laço entre as linhas 5 à 17 do bloco de código.

$$\hat{y} = \phi(\bar{w}^T \cdot \bar{x}) = \phi\left(\sum_{j=1}^m w_j x_j\right)$$

O número de neurônios, assim como o número de entradas, pode ser facilmente alterado quando o componente for instanciado devido ao uso de *generics* no código, entretanto, temos um limite de multiplicadores no FPGA, assim, caso o número de multiplicações exceder o disponível serão alocados recursos lógicos para efetuar a multiplicação.

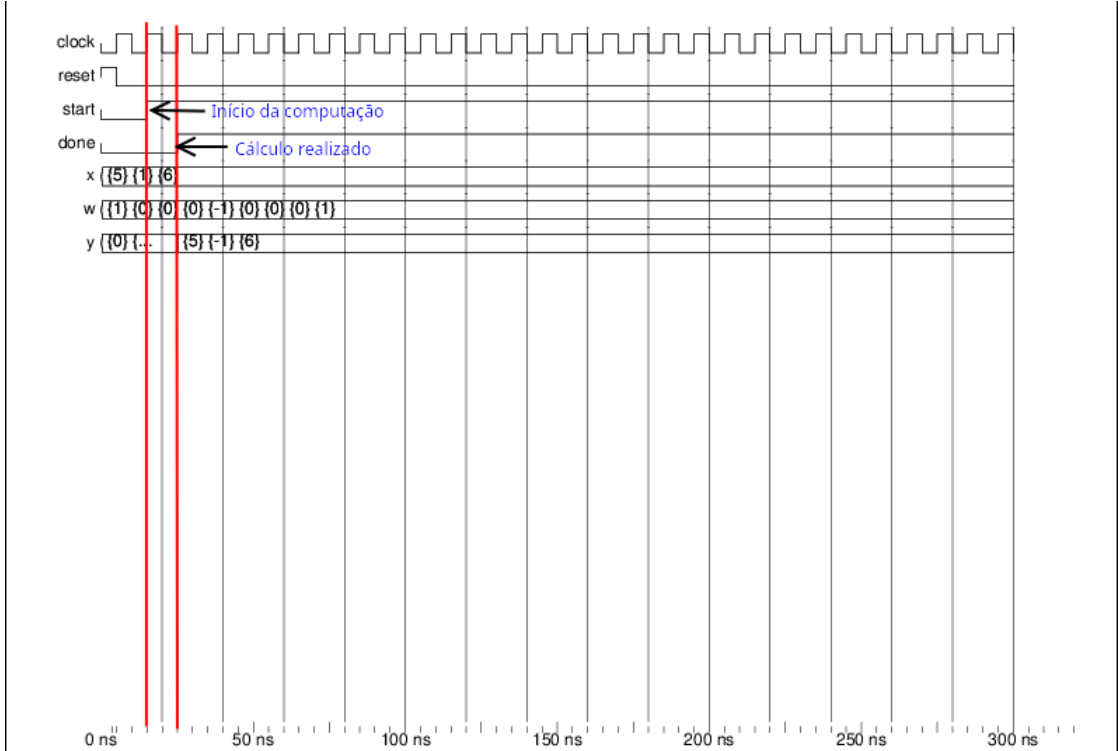
```

0   if rst = '1' then
1       y <= (others => (others => '0'));
2       done <= '0';
3   elsif rising_edge(clk) and start = '1' then
4       ----- multiply-accumulate: -----
5       l1: for i in 1 to neurons loop
6           sum := (others => '0');
7           l2: for j in 1 to n_inputs loop
8               product := x(j)*w(n_inputs*(i-1)+j);
9               sign := sum(sum'left);
10              sum := sum + product;
11          ----- overflow check: -----
12              if (sign = product(product'left)) and (sum(sum'left) /= sign) then
13                  sum := (sum'left => sign, others => not sign);
14              end if;
15          end loop l2;
16          ----- output: -----
17          y(i) <= sum;
18      end loop l1;
19      done <= '1';
20  end if;

```

Como não existem laços em *hardware*, são alocados recursos físicos para computar tudo de uma vez, assim esse “laço” é executado em um ciclo de *clock*. As adaptações feitas no código do Pedroni estão no sinal de “start”, para indicar que os pesos e as entradas estão prontas, e o sinal “done” para indicar quando os resultados da saída são válidos.

Figura 15 – Simulação da rede neural no Questa



Fonte: Elaboração própria.

4 RESULTADOS E DISCUSSÃO

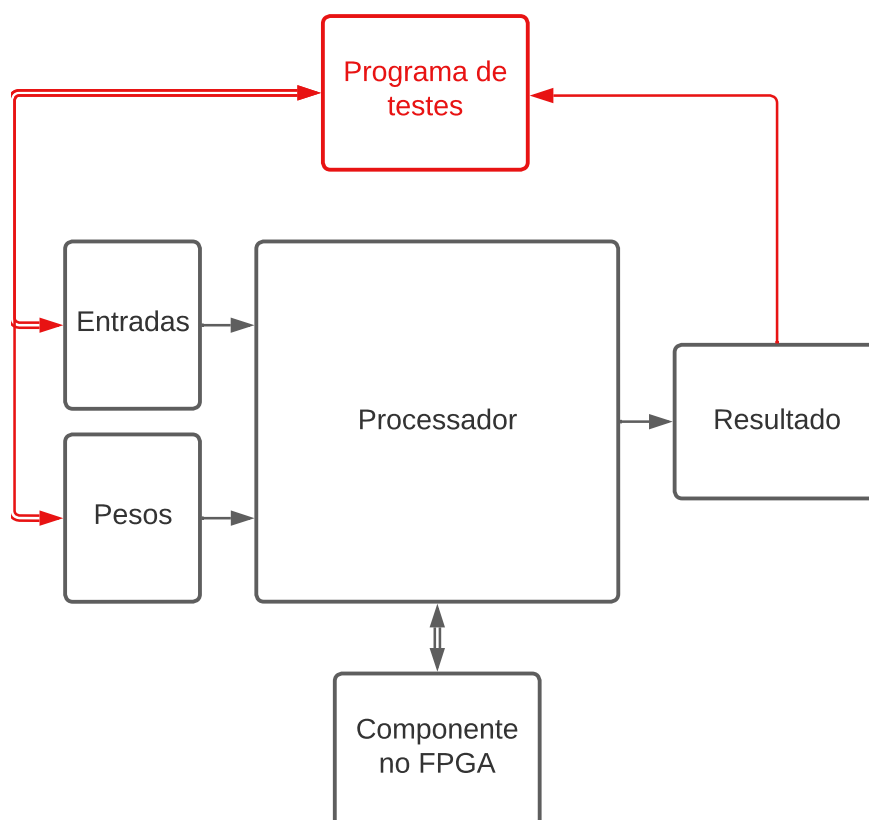
Para testar a funcionalidade do componente implementado em hardware, foi utilizado como referência o *framework* de *machine learning* PyTorch, bem conhecido em pesquisa e desenvolvimento. E para a aplicação, utilizou-se o *scikit-learn* para o treinamento de um *perceptron* na classificação de flores.

4.1 Testes

Para validar os resultados produzidos pelo componente a diferentes parâmetros e entradas, foi escrito um pequeno *script* para gerar dados a serem alimentados ao programa em C. A maneira usada para validar o componente pode ser observado na Figura 16. O programa de testes foi feito em Python e basicamente usa duas bibliotecas, isso é, o Numpy¹ para gerar números aleatórios inteiros, e o Pytorch para conferir o resultado. Os números gerados são as entradas e parâmetros a serem usados na computação do *layer* totalmente conectado, cada linha do arquivo de “Entradas” tem a quantidade de entradas do *layer* totalmente conectado e cada linha do arquivo de “Pesos” tem a quantidade de pesos do *layer* totalmente conectado. Esses arquivos são passados para o programa de teste que por sua vez lê linha-a-linha do arquivo de “Entradas” e do arquivo de “Pesos” e envia para o componente em FPGA as entradas e pesos, a computação é realizada e o resultado é enviado para o processador, que salva o resultado da multiplicação e soma em uma linha no arquivo “Resultado”. Ao final, o resultado é repassado para o *framework* PyTorch que faz a verificação e linha-a-linha do arquivo de “Resultado” com o que a função do *framework* produziria a partir das entradas e pesos, e informa caso existam valores divergentes produzidos pelo componente do FPGA com o que o *framework* retornou.

¹ Numpy é uma biblioteca em Python que estende funcionalidades da linguagem como a inclusão de matrizes multidimensionais e inúmeras funções matemáticas.

Figura 16 – Teste da aplicação



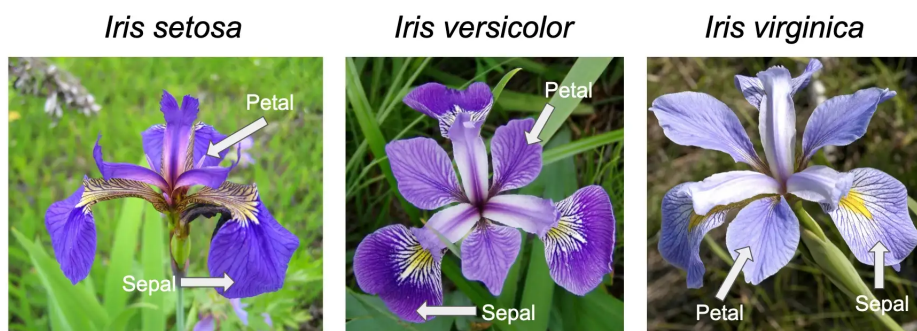
Fonte: Elaboração própria.

A validação do funcionamento correto do componente foi feita para números inteiros. Também foram executados testes com ponto fixo, mas devido à imprecisão gerada pela quantização em Q8 a verificação automatizada não foi feita.

4.2 Classificação

Como prova de conceito de uma aplicação que usa o componente em FPGA, aplicou-se um exemplo de classificação binária encontrada no livro de Raschka e Mirjalili (*Python Machine Learning*), onde são separados dois tipos de flores: setáceas e versicolor. O Scikit-learn disponibiliza o *dataset* Iris, onde existem três classes de flores: Setosa, Versicolor e Virgínica. Para cada classe, existem quatro características, ou *features* de cada flor: comprimento da sépala, largura da sépala, comprimento da pétala e largura da pétala. Seguindo o exemplo do livro, apenas duas *features* foram usadas, comprimento da sépala e comprimento da pétala, para classificar as duas classes. As flores com as características podem ser visualizadas na Figura 17.

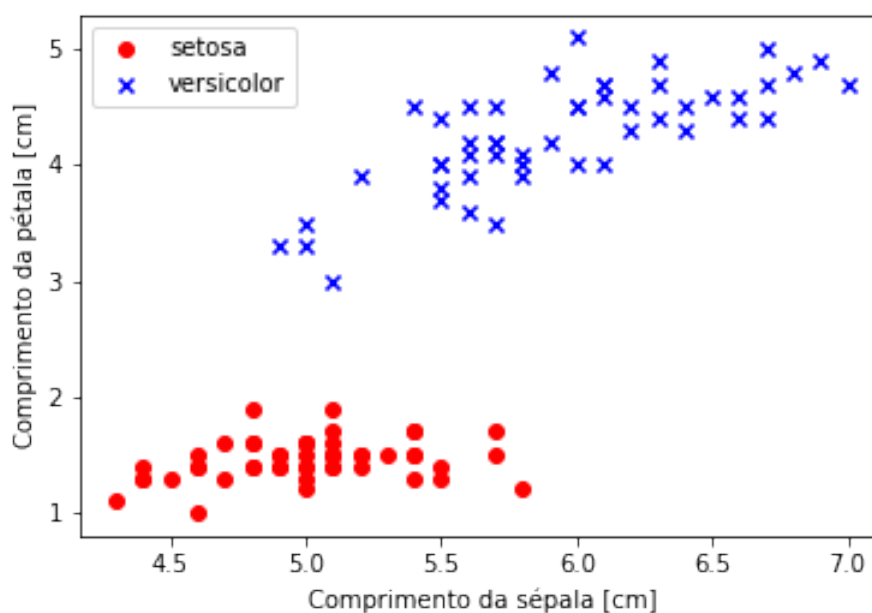
Figura 17 – Iris dataset



Fonte: Cui (2020).

Na Figura 18 é possível observar como as classes podem ser separadas com base nas características escolhidas.

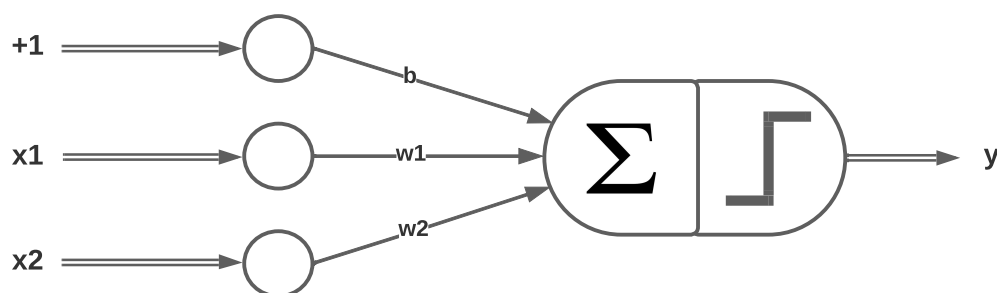
Figura 18 – Setácea vs Versicolor



Fonte: Adaptado de Raschka (2017, p. 30).

Para treinar o classificador, o *dataset* foi dividido em dois, sendo 30% reservado para teste e 70% para treino. O modelo usado para classificação é um *perceptron* pode ser visto na Figura 19 com uma saída, duas entradas (comprimento da sépala e comprimento da pétala) e *bias*.

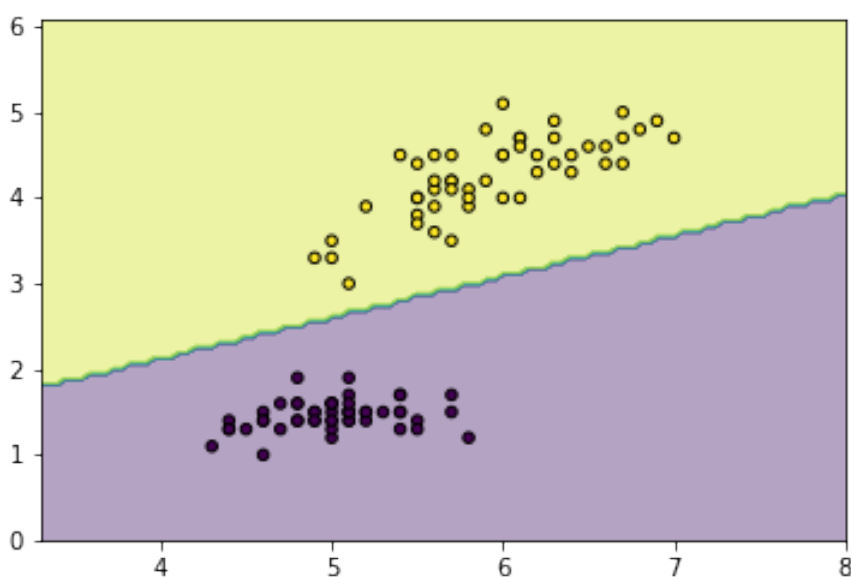
Figura 19 – Modelo usado na classificação



Fonte: Elaboração própria.

O modelo conseguiu separar corretamente todos os casos com uma função de ativação do tipo sinal, como pode ser visto na Figura 20.

Figura 20 – Limiar de decisão



Fonte: Adaptado de Raschka (2017, p. 33).

A Figura 20 indica onde o *perceptron* toma a decisão de classificar entre um tipo de flor e outro, caso o ponto com o comprimento da sépala e comprimento da pétala esteja na região amarela o modelo decide que a entrada é uma flor do tipo versicolor, e setosa caso o ponto esteja na região roxa.

4.3 Síntese

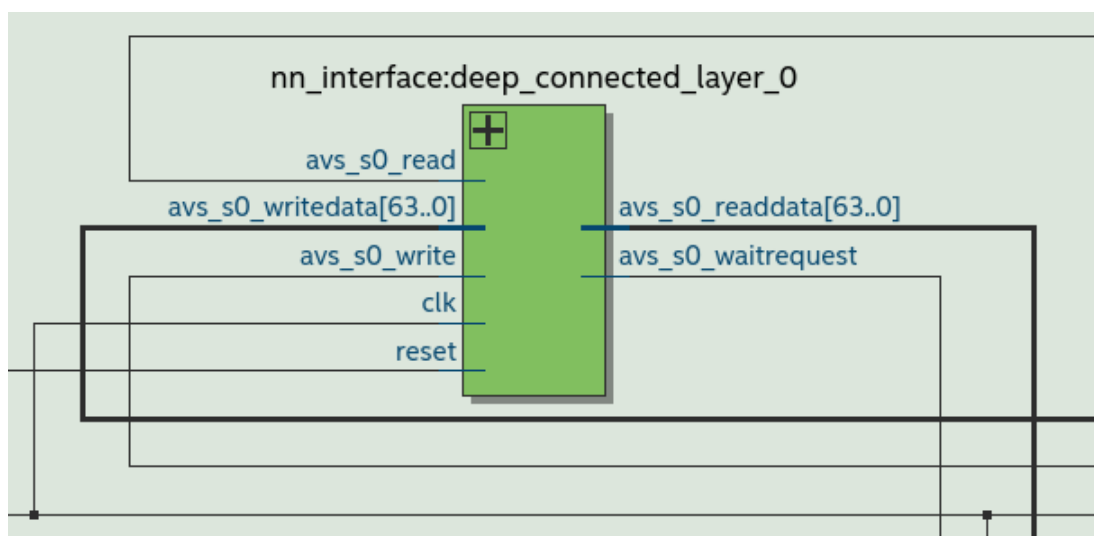
O Quadro 2 sumariza a quantidade de recursos usados no projeto.

Quadro 2 – Sumário de recursos usados

Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	socket_ghrd
Top-Level Entity Name	socket_ghrd_top
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	850 / 41,910 (2 %)
Total registers	2019
Total pins	81 / 499 (16 %)
Total virtual pins	0
Total block memory bits	41,472 / 5,662,720 (< 1 %)
Total DSP Blocks	3 / 112 (3 %)

Fonte: Gerado a partir de Intel (2021a).

Os recursos usados pelo componente são proporcionais ao número de entradas e saídas do dispositivo. O número de DSPs usados é igual à quantidade de entradas multiplicado pela quantidade de saídas. Foram usados dois parâmetros para a classificação das flores, como a computação envolve a soma do *bias* após as multiplicações, utilizou-se uma entrada a mais com o peso unitário, portanto o *bias* é multiplicado por '1' e somado aos outros valores, desse modo, toda a computação de multiplicação e soma é realizada no FPGA e o processador é responsável pela função de ativação e classificação, assim o número de multiplicadores usados foram três. O componente pode ser constatado no diagrama de bloco produzido pelo *software* Quartus.

Figura 21 – Diagrama de blocos

Fonte: Gerado a partir de Intel (2021a).

4.4 Programando FPGA

Para configurar o FPGA se usou *device tree overlay* (DTO), o arquivo gerado da síntese foi convertido de *SRAM Object Files* (sof) para *Raw Binary File* (rbf) no próprio Quartus.

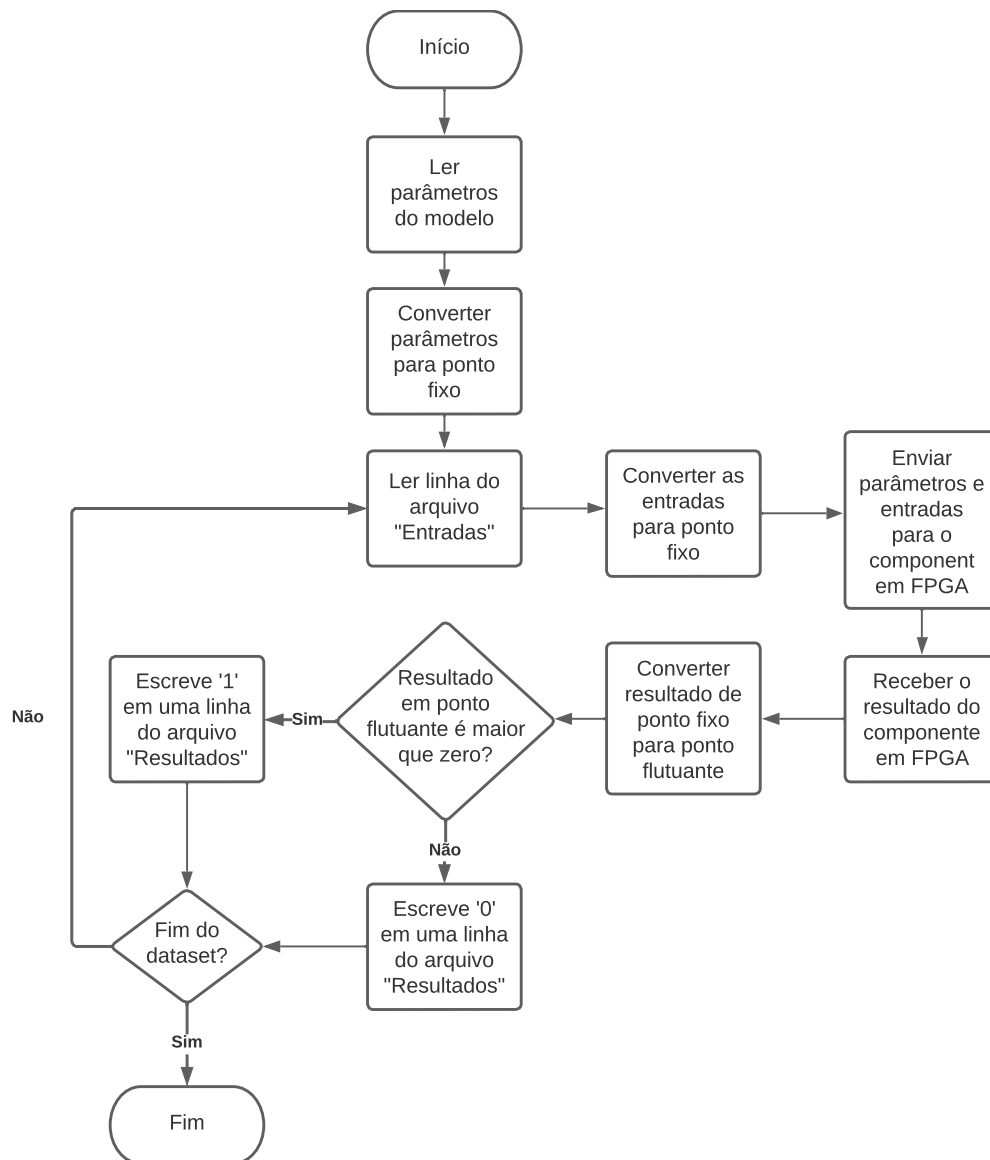
```
21 /dts-v1/;
22 /plugin/;
23
24 /{
25     fragment00 {
26         target-path = "/soc/base_fpga_region";
27         __overlay__ {
28             #address-cells = <1>;
29             #size-cells = <1>;
30             firmware-name = "perceptron.rbf";
31         };
32     };
33 };
```

Para compilar a *devicetree source overlay* (dtso) para uma *devicetree overlay binary* (dtbo) se usou o *devicetree compiler*.

4.5 Inferência no SoC-FPGA

Com os parâmetros do modelo obtidos no treinamento, a validação se deu de maneira semelhante aos testes. Os parâmetros, assim como os dados a serem classificados, foram salvos em um arquivo de texto e passados para o processador. O processador leu os parâmetros do modelo do arquivo, e para cada linha a ser classificada enviou sempre os mesmos parâmetros. O fluxograma na Figura 22 mostra os passos para realizar inferência.

Figura 22 – Fluxograma do processo de inferência



Fonte: Elaboração própria.

O arquivo com os parâmetros do modelo está no formato $b \ w_1 \ w_2$, o primeiro valor é o *bias* e os seguintes são os pesos.

0 -1.0 -3.0 6.5

O *bias* pode ser somado após a operação de multiplicação e soma do tamanho da sépala e pétala com seus respectivos pesos no programa sendo executado no processador, ou para manter toda a operação foram usados três entradas no componente em FPGA. Assim, é enviado o valor unitário (a ser multiplicado com o *bias*) antes do tamanho da sépala e da pétala, dessa forma o *perceptron* no FPGA possui três entradas, sendo que um dos pesos é sempre '1' como pode ser visto na Figura 19.

As entradas do modelo a serem classificadas estão como (tamanho da sépala, tamanho da pétala):

```
0 5.1 1.4
1 4.9 1.4
2 :    :
3 :    :
```

O FPGA está fazendo apenas multiplicação de números inteiros, portanto os parâmetros e as entradas são convertidas de decimal para ponto fixo antes de serem enviadas para o FPGA. Após, o resultado é convertido de ponto fixo para decimal antes de fazer a classificação.

A classificação é feita com um “*if*” (se), caso o valor seja maior do que zero pertence a uma classe, e pertence a outra caso contrário.

Cada linha do arquivo é o resultado da classificação para uma entrada.

```
0 0
1 0
2 :
3 :
4 1
5 1
```

O arquivo com o resultado da classificação foi comparado com o valor verdadeiro da classe através da função “*accuracy_score*” da biblioteca *scikit-learn* que retornou o valor máximo “1” significando que todas as predições foram corretas conforme esperado pelo limiar de decisão da Figura 20.

5 CONSIDERAÇÕES FINAIS

A realização do conceito de *edge AI* exige superação de diferentes obstáculos, alta carga de processamento combinada com restrições energéticas tornam *hardware* especializado uma necessidade. Além dos desafios mencionados, modificações na arquitetura do modelo usado exigem reprogramação do *hardware* dedicado. Assim, o uso de FPGAs em *edge devices* é uma possível solução as dificuldades a serem superadas, delegando tarefas custosas computacionalmente para o componente em FPGA enquanto o restante pode ser executado por um processador de propósito geral.

O interesse em aceleradores de *hardware* para algoritmos de rede neural tem-se intensificado, devido aos modelos terem ficado maiores e mais complexos. Assim, existem soluções que envolvem o uso de ASICs, sendo a TPU um exemplo para treinamento de modelos de rede neural. No campo da pesquisa existem trabalhos utilizando FPGAs para fazer inferência de modelos inteiros com resultados interessantes relativo ao consumo energético.

Nesse trabalho, pensou-se no uso do FPGA como um auxiliar, sendo responsável por computações que exigiriam muito de um microprocessador. Encontrando assim, utilidade em aplicações no qual é necessário eficiência energética e velocidade. Em conjunto com o processador, essa solução provê desempenho e flexibilidade em diversas situações.

No decorrer do trabalho, foi apresentado o conhecimento técnico para a configuração de um *System-on-a-Chip*, a implementação de um *layer* totalmente conectado em VHDL, a programação do FPGA usando *Devicetree Overlay*, e a comunicação do SoC com o componente em FPGA. Também foi demonstrado o funcionamento do sistema em um problema de classificação binária.

5.1 Sugestões para trabalhos futuros

Uma forma de reduzir a quantidade de dados movimentados seria o armazenamento dos parâmetros salvos nos blocos de memória M10K presentes na malha lógica, reduzindo assim o *overhead* da comunicação entre o SoC e o FPGA. Melhor uso dos recursos disponíveis do FPGA poderia ser alcançado usando uma arquitetura de rede neural convolucional, devido às operações de convolução não necessitarem de muitos multiplicadores para um filtro pequeno, e a possibilidade de recorrer à localidade dos dados. Outro ponto interessante seria medir e comparar métricas como latência e consumo energético de um modelo executando apenas no processador, com e sem auxílio de instruções SIMD, e com o auxílio do componente especializado no FPGA.

REFERÊNCIAS

- AGGARWAL, C. C. *Neural Networks and Deep Learning: A textbook*. 1. ed. Cham, Switzerland: Springer, 2018. 1–52 p. 26, 27, 28, 29
- BIERZYNSKI, K.; ESCOBAR, A.; EBERL, M. Cloud, fog and edge: Cooperation for the future? In: IEEE. *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*. [S.l.], 2017. p. 62–67. 19
- CUI, Y. *The Iris Dataset — A Little Bit of History and Biology*. 2020. Disponível em: <https://towardsdatascience.com/the-iris-dataset-a-little-bit-of-history-and-biology-fb4812f5a7b5>. Acesso em: 15 dez 2022. 40
- ENNO, L. *et al. FPGA Device Feature List (DFL) Framework Overview*. 2021. Disponível em: <https://github.com/torvalds/linux/blob/v5.16/Documentation/fpga/dfl.rst>. Acesso em: 22 out 2022. 22
- FINNERTY, A.; RATIGNER, H. Reduce power and cost by converting from floating point to fixed point. *WP491 (v1. 0)*, 2017. 26
- GRECO, L. *et al.* Trends in iot based solutions for health care: Moving ai to the edge. *Pattern Recognition Letters*, v. 135, p. 346–353, 2020. ISSN 0167-8655. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167865520301884>. 20
- INTEL. *Cyclone V Device Overview*. California, United States, 2018. 23
- INTEL. *Cyclone V Hard Processor System Technical Reference Manual*. California, United States, 2018. 21, 22, 32
- INTEL. *Cyclone V Device Handbook: Volume 1: Device interfaces and integration*. California, United States, 2020. 23
- INTEL. *Intel® Quartus® Prime and Quartus® II Software Support*. 2021. Disponível em: <https://www.intel.com/content/www/us/en/support/programmable/support-resources/design-software/sof-quartus.html>. Acesso em: 19 nov 2022. 30, 32, 33, 42
- INTEL. *Intel® Quartus® Prime Pro Edition User Guide: Platform designer*. California, United States, 2021. 24
- INTEL. *Questa*-Intel® FPGA Edition Software*. 2021. Disponível em: <https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/questa-edition.html>. Acesso em: 19 nov 2022. 30
- INTEL. *Avalon® Interface Specifications*. California, United States, 2022. 24
- INTERNATIONAL BUSINESS MACHINES CORPORATION – IBM. **Compare deep learning frameworks**. Picking the right deep learning framework based on your individual workload is an essential first step in deep learning. 2021. Disponível em: <https://developer.ibm.com/articles/compare-deep-learning-frameworks/>. Acesso em: 05 jul 2022. 29

KNUTH, D. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms (2nd Edition)*. [S.l.]: Addison-Wesley Publishing Company, 1981. 624 p. ISBN 978-0201038224. 25

KUSSWURM, D. *Modern X86 Assembly Language Programming: 32-bit, 64-bit, sse, and avx*. 1. ed. California, United States: Apress, 2014. 1–25 p. 17

LEE, Y.-L.; TSUNG, P.-K.; WU, M. Technology trend of edge ai. In: *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. Hsinchu, Taiwan: IEEE, 2018. 16

LIKELY, G. *Linux and the Devicetree*. 2021. Disponível em: <https://www.kernel.org/doc/html/latest/devicetree/usage-model.html>. Acesso em: 02 nov 2022. 23

LOVE, R. *Linux Kernel Development*. 3. ed. Indiana, United States: Pearson Education, Inc, 2010. 1–10 p. 22

PADGETT, W. T.; ANDERSON, D. V. *Fixed-Point Signal Processing (Synthesis Lectures on Signal Processing, 9)*. [S.l.]: Morgan and Claypool Publishers, 2009. 134 p. ISBN 978-1598292589. 26

PEDRONI, V. A. *Circuit Design with VHDL*. 1. ed. Massachusetts, United States: MIT Press, 2004. 275–303 p. 30, 35

RASCHKA, S.; MIRJALILI, V. *Python Machine Learning*. 2. ed. Birmingham, United Kingdom: Packt Publishing, 2017. 17–50 p. 27, 40, 41

SCIKIT-LEARN. *scikit-learn Machine Learning in Python*. 2022. Disponível em: <https://scikit-learn.org/stable/index.html>. Acesso em: 01 dez 2022. 29

SHAHID, A.; MUSHTAQ, M. A survey comparing specialized hardware and evolution in tpus for neural networks. In: *2020 IEEE 23rd International Multitopic Conference (INMIC)*. Bahawalpur, Pakistan: IEEE, 2020. p. 1–6. 17

SHI, W. *et al.* Edge computing: Vision and challenges. *IEEE internet of things journal*, IEEE, v. 3, n. 5, p. 637–646, 2016. 19

SIMMONDS, C. *Mastering Embedded Linux Programming*. 2. ed. Birmingham, United Kingdom: Packt, 2017. 47–78 p. 22

TERASIC. *SoCKit: User manual*. Hsinchu, Taiwan, 2014. 30

THE KERNEL DEVELOPMENT COMMUNITY. *Devicetree Overlay Notes*. 2022. Disponível em: <https://www.kernel.org/doc/html/latest/devicetree/overlay-notes.html>. Acesso em: 02 nov 2022. 23

THE LINUX FOUNDATION. *Chapter 3. The Root Filesystem*. 2015. Disponível em: https://refspecs.linuxfoundation.org/FHS_3.0/fhs/ch03.html. Acesso em: 02 nov 2022. 22

VÉSTIAS, M. Processing systems for deep learning inference on edge devices. In: MASTORAKIS, G. *et al.* (Ed.). *Convergence of Artificial Intelligence and the Internet of Things*. Cham, Switzerland: Springer, 2020. p. 213–240. 16, 20

WANG, T. *et al.* An overview of fpga based deep learning accelerators: Challenges and opportunities. In: *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. Zhangjiajie, China: IEEE, 2019. p. 1674–1681. 17

ZHOU, Z. *et al.* Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proceedings of the IEEE*, v. 107, n. 8, p. 1738–1762, 2019. 20

APÊNDICES

APÊNDICE A – CÓDIGO VHDL PONTO FIXO

```

0 library ieee;
1 use ieee.numeric_std.all;
2 use ieee.std_logic_1164.all;
3
4 package fixed_point_pkg is
5     type fixed_point is array ( integer range <> ) of std_logic;
6     subtype q1_15 is fixed_point(0 downto -15);
7     subtype q8_8 is fixed_point(7 downto -8);
8     subtype q16_16 is fixed_point(15 downto -16);
9
10    function "+" (a, b: fixed_point) return fixed_point;
11    function "*" (a, b: fixed_point) return fixed_point;
12
13    -- Conversion functions
14    function fixed_point2std_logic_vector(fp: fixed_point)
15    return std_logic_vector;
16
17    function fixed_point2signed(fp: fixed_point)
18    return signed;
19
20    function std_logic_vector2fixed_point(logic_vector: std_logic_vector; high, low:
21    integer)
22    return fixed_point;
23
24    function signed2fixed_point(signed_vector: signed; high, low: integer)
25    return fixed_point;
26
27    -- testbench purpose
28    function fixed_point2real(fp: fixed_point)
29    return real;
30 end package;
31
32
33 package body fixed_point_pkg is
34
35     function "+" (a, b: fixed_point)
36     return fixed_point is
37
38         variable a_s, b_s, temp_s : signed(a'length - 1 downto 0);
39         variable c_fp : fixed_point(a'range);
40
41     begin
42
43         a_s := fixed_point2signed(a);
44         b_s := fixed_point2signed(b);
45         temp_s := a_s + b_s;
46
47         c_fp := signed2fixed_point(temp_s, a'left, a'right);
48
49         return c_fp;
50
51     end "+";
52
53     function "*" (a, b: fixed_point)

```

```

54  return fixed_point is
55
56      variable a_s, b_s : signed(a'length - 1 downto 0);
57      variable temp_s   : signed(2*a'length - 1 downto 0);
58      variable c_fp     : fixed_point(a'left + b'left + 1 downto a'right + b'right);
59
60  begin
61
62      a_s := fixed_point2signed(a);
63      b_s := fixed_point2signed(b);
64      temp_s := a_s * b_s;
65
66      c_fp := signed2fixed_point(temp_s, a'left + b'left + 1, a'right + b'right);
67
68      return c_fp;
69  end "*";
70  -- Conversion functions
71
72  function fixed_point2std_logic_vector(fp: fixed_point)
73  return std_logic_vector is
74
75      variable logic_vector : std_logic_vector(fp'length - 1 downto 0);
76
77  begin
78
79      for i in fp'range loop
80          logic_vector(i-fp'right) := fp(i);
81      end loop;
82
83      return logic_vector;
84
85  end fixed_point2std_logic_vector;
86
87  function fixed_point2signed(fp: fixed_point)
88  return signed is
89
90      variable signed_vector : signed(fp'length - 1 downto 0);
91
92  begin
93
94      for i in fp'range loop
95          signed_vector(i-fp'right) := fp(i);
96      end loop;
97
98      return signed_vector;
99
100  end fixed_point2signed;
101
102  function std_logic_vector2fixed_point(logic_vector: std_logic_vector; high, low:
    integer)
103  return fixed_point is
104
105      variable fp: fixed_point(high downto low);
106
107  begin
108
109      for i in low to high loop
110          fp(i) := logic_vector(i-low);

```

```
111     end loop;
112
113     return fp;
114 end std_logic_vector2fixed_point;
115
116 function signed2fixed_point(signed_vector: signed; high, low: integer)
117 return fixed_point is
118
119     variable fp: fixed_point(high downto low);
120
121 begin
122
123     for i in low to high loop
124         fp(i) := signed_vector(i-low);
125     end loop;
126
127     return fp;
128 end signed2fixed_point;
129
130 function fixed_point2real(fp: fixed_point)
131 return real is
132
133     variable real_value, max_value : real := 0.0;
134     variable index_value : real := 1.0;
135
136 begin
137
138     for i in 0 to fp'high-1 loop
139         if fp(i) = '1' then
140             real_value := real_value + index_value;
141         end if;
142         index_value := index_value * 2.0;
143     end loop;
144
145     max_value := index_value;
146
147     index_value := 0.5;
148     for i in -1 downto fp'low loop
149         if fp(i) = '1' then
150             real_value := real_value + index_value;
151         end if;
152         index_value := index_value * 0.5;
153     end loop;
154
155     if fp(fp'high) = '1' then
156         real_value := real_value - max_value;
157         report "fixed_point2real: negative value" & real'image(real_value);
158     end if;
159
160     return real_value;
161 end fixed_point2real;
162
163 end package body;
```

APÊNDICE B – CÓDIGO VHDL DEFINIÇÃO DE TIPOS

```
0  library ieee;
1  use ieee.numeric_std.all;
2  use ieee.std_logic_1164.all;
3  use work.fixed_point_pkg.all;
4
5  package fixedpoint_array_types is
6      type q8_8array is array (natural range <>) of q8_8;
7      type q16_16array is array (natural range <>) of q16_16;
8  end fixedpoint_array_types;
```

APÊNDICE C – CÓDIGO VHDL REDE NEURAL

```

0  ----- Project nn (main code): -----
1  library ieee;
2  use ieee.numeric_std.all;
3  use ieee.std_logic_1164.all; -- package needed for signed
4  use work.fixed_point_pkg.all;
5  use work.fixedpoint_array_types.all;
6  -----
7
8  entity nn is
9      generic (
10         neurons : integer := 3;
11         n_inputs: integer := 3
12     );
13     port (
14         clk      : in  std_logic;
15         rst      : in  std_logic;
16         start    : in  std_logic;
17         x        : in  q8_8array(1 to n_inputs);
18         w        : in  q8_8array(1 to n_inputs*neurons);
19         y        : out q16_16array(1 to neurons);
20         done     : out std_logic
21     );
22 end nn;
23
24 architecture neural of nn is
25 begin
26
27     compute : process (rst, clk)
28         variable product, sum : q16_16;
29         variable sign        : std_logic;
30     begin
31         if rst = '1' then
32             y <= (others => (others => '0'));
33             done <= '0';
34         elsif rising_edge(clk) and start = '1' then
35             ----- multiply-accumulate: -----
36             l1: for i in 1 to neurons loop
37                 sum := (others => '0');
38                 l2: for j in 1 to n_inputs loop
39                     product := x(j)*w(n_inputs*(i-1)+j);
40                     sign := sum(sum'left);
41                     sum := sum + product;
42                 ----- overflow check: -----
43                 if (sign = product(product'left)) and (sum(sum'left) /= sign) then
44                     sum := (sum'left => sign, others => not sign);
45                 end if;
46             end loop l2;
47             ----- output: -----
48             y(i) <= sum;
49         end loop l1;
50         done <= '1';
51     end if;
52 end process;
53
54 end neural;

```

APÊNDICE D – CÓDIGO VHDL IMPLEMENTAÇÃO DO PROTOCOLO AVALON

```

0 library ieee;
1 use ieee.numeric_std.all;
2 use ieee.std_logic_1164.all; -- package needed for signed
3 use work.fixed_point_pkg.all;
4 use work.fixedpoint_array_types.all;
5
6
7 entity nn_interface is
8   generic (
9     neurons : integer := 3;
10    n_inputs: integer := 3;
11    bits     : integer := 32
12  );
13  port (
14    clk           : in  std_logic;
15    reset         : in  std_logic;
16    avs_s0_read   : in  std_logic;
17    avs_s0_readdata : out std_logic_vector(63 downto 0);
18    avs_s0_write  : in  std_logic;
19    avs_s0_writedata : in  std_logic_vector(63 downto 0);
20    avs_s0_waitrequest : out std_logic
21  );
22 end entity nn_interface;
23
24 architecture RTL of nn_interface is
25   type state_type is (RESET_STATE, IDLE, RECEIVE_DATA, PROCESS_DATA, SEND_DATA);
26   signal state : state_type;
27   signal x : q8_8array(1 to n_inputs);
28   signal w : q8_8array(1 to neurons*n_inputs);
29   signal buffer_in : q8_8array(1 to neurons*n_inputs + n_inputs);
30   signal buffer_out : q16_16array(1 to neurons);
31   signal done, start, clear_nn : std_logic;
32 begin
33
34   state_transition : process(reset, clk)
35     variable index_in, index_out : integer;
36   begin
37     if reset = '1' then
38       state <= RESET_STATE;
39     elsif rising_edge(clk) then
40       avs_s0_readdata <= (others => '0');
41       case state is
42         when RESET_STATE =>
43           state <= IDLE;
44           buffer_in <= (others => (others => '0'));
45
46         when IDLE =>
47           index_in := 1;
48           if avs_s0_write = '1' then
49             state <= RECEIVE_DATA;
50           end if;
51
52         when RECEIVE_DATA =>
53           if index_in > neurons*n_inputs + n_inputs then
54             state <= PROCESS_DATA;

```

```

55         else
56             -- Sign reduction --
57             buffer_in(index_in)(buffer_in(index_in)'left) <= avs_s0_writedata(
avs_s0_writedata'left);
58             buffer_in(index_in)(buffer_in(index_in)'left-1 downto buffer_in(
index_in)'right) <= std_logic_vector2fixed_point(avs_s0_writedata(bits-2 downto
0), buffer_in(index_in)'left-1, buffer_in(index_in)'right);
59         end if;
60
61         if avs_s0_write = '1' then
62             index_in := index_in + 1;
63         end if;
64
65         when PROCESS_DATA =>
66             index_out := 1;
67             if done = '1' then
68                 state <= SEND_DATA;
69             end if;
70
71         when SEND_DATA =>
72             if index_out > neurons then
73                 state <= IDLE;
74             else
75                 -- Sign Extension --
76                 avs_s0_readdata(63 downto bits) <= ((avs_s0_readdata'length - 1) downto
bits => buffer_out(index_out)(buffer_out(index_out)'left));
77                 avs_s0_readdata(bits-1 downto 0) <= fixed_point2std_logic_vector(
buffer_out(index_out));
78             end if;
79
80             if avs_s0_read = '1' then
81                 index_out := index_out + 1;
82             end if;
83         end case;
84     end if;
85 end process;
86
87 meely : process(state)
88 begin
89
90     clear_nn <= '0';
91     start <= '0';
92     avs_s0_waitrequest <= '0';
93
94     case state is
95         when RESET_STATE =>
96             clear_nn <= '1';
97
98         when IDLE =>
99             if avs_s0_write = '1' then
100                 avs_s0_waitrequest <= '1';
101             end if;
102
103         when RECEIVE_DATA =>
104             clear_nn <= '1';
105             start <= '0';
106
107         when PROCESS_DATA =>

```

```
108     start <= '1';
109
110     if avs_s0_read = '1' then
111         avs_s0_waitrequest <= '1';
112     end if;
113
114     when SEND_DATA =>
115
116     end case;
117 end process;
118
119 x(1 to n_inputs) <= buffer_in(1 to n_inputs);
120 w(1 to neurons*n_inputs) <= buffer_in(n_inputs + 1 to n_inputs + neurons*n_inputs
121 );
122
123 linear_layer : entity work.nn
124 generic map (
125     neurons => neurons, n_inputs => n_inputs
126 )
127 port map (
128     clk => clk, rst => clear_nn, x => x, w => w,
129     start => start, y => buffer_out, done => done
130 );
131 end architecture RTL;
```

APÊNDICE E – CÓDIGO C COMUNICAÇÃO COM FPGA

```

0 #include <error.h>
1 #include <fcntl.h>
2 #include <stdint.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/mman.h>
6 #include <unistd.h>
7
8 #define HPS_TO_FPGA_BRIDGE 0xC0000000
9 #define HPS_TO_FPGA_BRIDGE_SPAN 0x07
10
11 #define NN_ACCELERATOR 0x00
12
13 #define N_WEIGHTS 9
14
15 static int fd;
16 static uint8_t *bridge = NULL, *component_address = NULL;
17
18 int map_component(void) {
19     fd = open("/dev/mem", O_RDWR | O_SYNC);
20     if (fd < 0) {
21         perror("Couldn't open /dev/mem\n");
22         return -1;
23     }
24
25     bridge = mmap(NULL, HPS_TO_FPGA_BRIDGE_SPAN, PROT_READ | PROT_WRITE,
26         MAP_SHARED, fd, HPS_TO_FPGA_BRIDGE);
27     if (bridge == MAP_FAILED) {
28         perror("couldn't map bridge");
29         return -1;
30     }
31
32     component_address = bridge + NN_ACCELERATOR;
33
34     return 0;
35 }
36
37 int unmap_component(void) {
38     int result = munmap(bridge, HPS_TO_FPGA_BRIDGE_SPAN);
39     if (result < 0) {
40         perror("Couldn't unmap memory");
41         return -1;
42     }
43
44     close(fd);
45
46     component_address = NULL;
47
48     return 0;
49 }
50
51 void send_data(void *data, size_t n) {
52     if (!component_address) {
53         perror("Component not accessible, map device first");
54         return;

```

```
55     }
56
57     for(size_t i = 0; i < n; ++i) {
58         *((uint64_t *)component_address) = ((uint64_t *)data)[i];
59     }
60 }
61
62 void read_data(void *data, size_t n) {
63     if (!component_address) {
64         perror("Component not accessible, map device first");
65         return;
66     }
67
68     for(size_t i = 0; i < n; ++i) {
69         ((uint64_t *)data)[i] = *((uint64_t *)component_address);
70     }
71 }
```

APÊNDICE F – CÓDIGO C TESTES

```

0 #include <error.h>
1 #include <fcntl.h>
2 #include <inttypes.h>
3 #include <stdint.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <sys/mman.h>
7 #include <unistd.h>
8
9 #include "fpga_component.h"
10
11 #define SHIFT 8
12
13 void test_int(uint32_t n_inputs, uint32_t n_outputs) {
14     FILE *input_file = fopen("input_int.txt", "r");
15     if (input_file == NULL) {
16         perror("Couldn't open input_int.txt");
17         return;
18     }
19
20     uint32_t input_lines = 0;
21     int ch;
22     while ((ch = fgetc(input_file)) != EOF) {
23         if (ch == '\n') {
24             input_lines++;
25         }
26     }
27     rewind(input_file);
28
29     FILE *weights_file = fopen("weight_int.txt", "r");
30     if (weights_file == NULL) {
31         fclose(input_file);
32         perror("Couldn't open weight_int.txt");
33         return;
34     }
35
36     uint32_t weight_lines = 0;
37     while ((ch = fgetc(weights_file)) != EOF) {
38         if (ch == '\n') {
39             weight_lines++;
40         }
41     }
42     rewind(weights_file);
43
44     if (input_lines != weight_lines) {
45         printf("File Inputs and Weights must have the same number of lines\n");
46         fclose(input_file);
47         fclose(weights_file);
48         return;
49     }
50
51     FILE *output_file = fopen("result_int.txt", "w");
52     if (output_file == NULL) {
53         fclose(input_file);
54         fclose(weights_file);

```

```
55     perror("Couldn't open result_int.txt");
56     return;
57 }
58
59 int64_t *x = malloc(n_inputs*sizeof(int64_t));
60 if (x == NULL) {
61     fclose(input_file);
62     fclose(weights_file);
63     fclose(output_file);
64     perror("Error allocating memory");
65     return;
66 }
67
68 int64_t *w = malloc(n_inputs*n_outputs*sizeof(int64_t));
69 if (w == NULL) {
70     fclose(input_file);
71     fclose(weights_file);
72     fclose(output_file);
73     free(x);
74     perror("Error allocating memory");
75     return;
76 }
77
78 int64_t *y = malloc(n_outputs*sizeof(int64_t));
79 if (y == NULL) {
80     fclose(input_file);
81     fclose(weights_file);
82     fclose(output_file);
83     free(x);
84     free(w);
85     perror("Error allocating memory");
86     return;
87 }
88
89 for (uint32_t l = 0; l < input_lines; ++l) {
90     for (uint32_t i = 0; i < n_inputs; ++i) {
91         fscanf(input_file, "%" PRIu64, &x[i]);
92     }
93     send_data(x, n_inputs);
94
95     for (uint32_t i = 0; i < n_inputs*n_outputs; ++i) {
96         fscanf(weights_file, "%" PRIu64, &w[i]);
97     }
98     send_data(w, n_inputs*n_outputs);
99
100     read_data(y, n_outputs);
101     for (uint32_t i = 0; i < n_outputs; ++i) {
102         fprintf(output_file, "%" PRIu64 " ", y[i]);
103     }
104     fprintf(output_file, "\n");
105 }
106 fclose(input_file);
107 fclose(weights_file);
108 fclose(output_file);
109 free(x);
110 free(w);
111 free(y);
112 }
```

```
113
114 int floatpoint2fixedpoint(float fp_number, int shift) {
115     int fixedpoint = fp_number * (1 << shift);
116     return fixedpoint;
117 }
118
119 float fixedpoint2floatpoint(int fx_number, int shift) {
120     float floatpoint = fx_number / ((1 << shift) * 1.0);
121     return floatpoint;
122 }
123
124 void test_fixed_point(uint32_t n_inputs, uint32_t n_outputs) {
125     FILE *input_file = fopen("inputs_fp.txt", "r");
126     if (input_file == NULL) {
127         perror("Couldn't open inputs_fp.txt");
128         return;
129     }
130
131     uint32_t input_lines = 0;
132     int ch;
133     while ((ch = fgetc(input_file)) != EOF) {
134         if (ch == '\n') {
135             input_lines++;
136         }
137     }
138     rewind(input_file);
139
140     FILE *weights_file = fopen("weight_fp.txt", "r");
141     if (weights_file == NULL) {
142         fclose(input_file);
143         perror("Couldn't open weight_fp.txt");
144         return;
145     }
146
147     uint32_t weight_lines = 0;
148     while ((ch = fgetc(weights_file)) != EOF) {
149         if (ch == '\n') {
150             weight_lines++;
151         }
152     }
153     rewind(weights_file);
154
155     if (input_lines != weight_lines) {
156         printf("File Inputs and Weights must have the same number of lines\n");
157         fclose(input_file);
158         fclose(weights_file);
159         return;
160     }
161
162     FILE *output_file = fopen("result_fp.txt", "w");
163     if (output_file == NULL) {
164         fclose(input_file);
165         fclose(weights_file);
166         perror("Couldn't open result_fp.txt");
167         return;
168     }
169
170     int64_t *x_fixedpoint = malloc(n_inputs*sizeof(int64_t));
```

```
171     if (x_fixedpoint == NULL) {
172         fclose(input_file);
173         fclose(weights_file);
174         fclose(output_file);
175         perror("Error allocating memory");
176         return;
177     }
178
179     float *x_floatpoint = malloc(n_inputs*sizeof(float));
180     if (x_floatpoint == NULL) {
181         fclose(input_file);
182         fclose(weights_file);
183         fclose(output_file);
184         free(x_fixedpoint);
185         perror("Error allocating memory");
186         return;
187     }
188
189     int64_t *w_fixedpoint = malloc(n_inputs*n_outputs*sizeof(int64_t));
190     if (w_fixedpoint == NULL) {
191         fclose(input_file);
192         fclose(weights_file);
193         fclose(output_file);
194         free(x_fixedpoint);
195         free(x_floatpoint);
196         perror("Error allocating memory");
197         return;
198     }
199
200     float *w_floatpoint = malloc(n_inputs*n_outputs*sizeof(float));
201     if (w_floatpoint == NULL) {
202         fclose(input_file);
203         fclose(weights_file);
204         fclose(output_file);
205         free(x_fixedpoint);
206         free(x_floatpoint);
207         free(w_fixedpoint);
208         perror("Error allocating memory");
209         return;
210     }
211
212     int64_t *y_fixedpoint = malloc(n_outputs*sizeof(int64_t));
213     if (y_fixedpoint == NULL) {
214         fclose(input_file);
215         fclose(weights_file);
216         fclose(output_file);
217         free(x_fixedpoint);
218         free(x_floatpoint);
219         free(w_fixedpoint);
220         free(w_floatpoint);
221         perror("Error allocating memory");
222         return;
223     }
224
225     float *y_floatpoint = malloc(n_outputs*sizeof(float));
226     if (y_floatpoint == NULL) {
227         fclose(input_file);
228         fclose(weights_file);
```

```

229     fclose(output_file);
230     free(x_fixedpoint);
231     free(x_floatpoint);
232     free(w_fixedpoint);
233     free(w_floatpoint);
234     free(y_fixedpoint);
235     perror("Error allocating memory");
236     return;
237 }
238
239 for (uint32_t l = 0; l < input_lines; ++l) {
240     for (uint32_t i = 0; i < n_inputs; ++i) {
241         fscanf(input_file, "%f", &x_floatpoint[i]);
242         x_fixedpoint[i] = floatpoint2fixedpoint(x_floatpoint[i], 8);
243     }
244     send_data(x_fixedpoint, n_inputs);
245
246     for (uint32_t i = 0; i < n_inputs*n_outputs; ++i) {
247         fscanf(weights_file, "%f", &w_floatpoint[i]);
248         w_fixedpoint[i] = floatpoint2fixedpoint(w_floatpoint[i], 8);
249     }
250     send_data(w_fixedpoint, n_inputs*n_outputs);
251
252     read_data(y_fixedpoint, n_outputs);
253     for (uint32_t i = 0; i < n_outputs; ++i) {
254         y_floatpoint[i] = fixedpoint2floatpoint(y_fixedpoint[i], 16);
255         fprintf(output_file, "%f ", y_floatpoint[i]);
256     }
257     fprintf(output_file, "\n");
258 }
259 fclose(input_file);
260 fclose(weights_file);
261 fclose(output_file);
262 free(x_fixedpoint);
263 free(x_floatpoint);
264 free(w_fixedpoint);
265 free(w_floatpoint);
266 free(y_fixedpoint);
267 free(y_floatpoint);
268 }
269
270 void test_perceptron(uint32_t n_inputs, uint32_t n_outputs) {
271     FILE *dataset_file = fopen("dataset.txt", "r");
272     if (dataset_file == NULL) {
273         perror("Couldn't open dataset.txt");
274         return;
275     }
276
277     uint32_t input_lines = 0;
278     int ch;
279     while ((ch = fgetc(dataset_file)) != EOF) {
280         if (ch == '\n') {
281             input_lines++;
282         }
283     }
284     rewind(dataset_file);
285
286     FILE *weights_file = fopen("perceptron_parameters.txt", "r");

```

```
287 if (weights_file == NULL) {
288     fclose(dataset_file);
289     perror("Couldn't open perceptron_parameters.txt");
290     return;
291 }
292
293 FILE *output_file = fopen("predictions.txt", "w");
294 if (output_file == NULL) {
295     fclose(dataset_file);
296     fclose(weights_file);
297     perror("Couldn't open predictions.txt");
298     return;
299 }
300
301 // Save one more space to multiply with bias
302 int64_t *x_fixedpoint = malloc(n_inputs*sizeof(int64_t));
303 if (x_fixedpoint == NULL) {
304     fclose(dataset_file);
305     fclose(weights_file);
306     fclose(output_file);
307     perror("Error allocating memory");
308     return;
309 }
310
311 float *x_floatpoint = malloc(n_inputs*sizeof(float));
312 if (x_floatpoint == NULL) {
313     fclose(dataset_file);
314     fclose(weights_file);
315     fclose(output_file);
316     free(x_fixedpoint);
317     perror("Error allocating memory");
318     return;
319 }
320
321 int64_t *w_fixedpoint = malloc(n_inputs*n_outputs*sizeof(int64_t));
322 if (w_fixedpoint == NULL) {
323     fclose(dataset_file);
324     fclose(weights_file);
325     fclose(output_file);
326     free(x_fixedpoint);
327     free(x_floatpoint);
328     perror("Error allocating memory");
329     return;
330 }
331
332 float *w_floatpoint = malloc(n_inputs*n_outputs*sizeof(float));
333 if (w_floatpoint == NULL) {
334     fclose(dataset_file);
335     fclose(weights_file);
336     fclose(output_file);
337     free(x_fixedpoint);
338     free(x_floatpoint);
339     free(w_fixedpoint);
340     perror("Error allocating memory");
341     return;
342 }
343
344 int64_t *y_fixedpoint = malloc(n_outputs*sizeof(int64_t));
```

```

345     if (y_fixedpoint == NULL) {
346         fclose(dataset_file);
347         fclose(weights_file);
348         fclose(output_file);
349         free(x_fixedpoint);
350         free(x_floatpoint);
351         free(w_fixedpoint);
352         free(w_floatpoint);
353         perror("Error allocating memory");
354         return;
355     }
356
357     float *y_floatpoint = malloc(n_outputs*sizeof(float));
358     if (y_floatpoint == NULL) {
359         fclose(dataset_file);
360         fclose(weights_file);
361         fclose(output_file);
362         free(x_fixedpoint);
363         free(x_floatpoint);
364         free(w_fixedpoint);
365         free(w_floatpoint);
366         free(y_fixedpoint);
367         perror("Error allocating memory");
368         return;
369     }
370
371     // Is the same parameters for all input data
372     for (uint32_t i = 0; i < n_inputs*n_outputs; ++i) {
373         fscanf(weights_file, "%f", &w_floatpoint[i]);
374         w_fixedpoint[i] = floatpoint2fixedpoint(w_floatpoint[i], 8);
375     }
376     fclose(weights_file);
377
378     // The bias will be multiplied with 1, so we won't need to add the
379     // bias after
380     x_fixedpoint[0] = 1;
381     for (uint32_t l = 0; l < input_lines; ++l) {
382         for (uint32_t i = 1; i < n_inputs; ++i) {
383             fscanf(dataset_file, "%f", &x_floatpoint[i]);
384             x_fixedpoint[i] = floatpoint2fixedpoint(x_floatpoint[i], 8);
385         }
386         send_data(x_fixedpoint, n_inputs);
387
388         send_data(w_fixedpoint, n_inputs*n_outputs);
389
390         read_data(y_fixedpoint, n_outputs);
391         for (uint32_t i = 0; i < n_outputs; ++i) {
392             y_floatpoint[i] = fixedpoint2floatpoint(y_fixedpoint[i], 16);
393             // Perform classification here
394             if (*y_floatpoint > 0) {
395                 fprintf(output_file, "%d", 1);
396             } else {
397                 fprintf(output_file, "%d", 0);
398             }
399         }
400         fprintf(output_file, "\n");
401     }
402     fclose(dataset_file);

```

```
403     fclose(output_file);  
404     free(x_fixedpoint);  
405     free(x_floatpoint);  
406     free(w_fixedpoint);  
407     free(w_floatpoint);  
408     free(y_fixedpoint);  
409 }
```

APÊNDICE G – CÓDIGO C MAIN

```
0 #include <stdio.h>
1 #include <stdlib.h>
2 #include <string.h>
3
4 #include "fpga_component.h"
5 #include "tests.h"
6
7 int main(int argc, char *argv[]) {
8
9     if (argc == 1 || argc > 2) {
10         perror("Wrong number of parameters");
11         return EXIT_FAILURE;
12     }
13
14     if (map_component()) {
15         return EXIT_FAILURE;
16     }
17
18     if (!strcmp(argv[1], "integer_test")) {
19         test_int(N_INPUTS, N_NEURONS);
20     }
21
22     if (!strcmp(argv[1], "fixedpoint_test")) {
23         test_fixed_point(N_INPUTS, N_NEURONS);
24     }
25
26     if (!strcmp(argv[1], "perceptron_test")) {
27         test_perceptron(N_INPUTS, N_NEURONS);
28     }
29     if (unmap_component()) {
30         return EXIT_FAILURE;
31     }
32
33     return EXIT_SUCCESS;
34 }
```