Geração Procedural de Redes Neurais Flexíveis em VHDL para Aceleração em FPGA

Davi Mortari Vargas[1], **Gabriel Silveira de Oliveira**[2], **Israel Ribeiro Alencar**[3]

[1]Faculdade de Computação (FACOM)[1]
Universidade Federal do Mato Grosso do Sul (UFMS)
Campo Grande - MS - Brazil

Abstract. *With the increasing demand for efficient Artificial Intelligence solutions, hardware acceleration using Field-Programmable Gate Arrays (FPGAs) has emerged as a powerful alternative to traditional software implementations. However, a significant limitation in many VHDL-based Neural Network designs is their static nature, where the network topology is fixed in the source code, restricting scalability and reuse. To address this rigidity, this work proposes a methodology for the procedural generation of flexible Multi-Layer Perceptron (MLP) architectures. Using VHDL generic parameters and generate statements, the proposed system allows the dynamic instantiation of layers and neurons at synthesis time, enabling rapid prototyping without manual code rewriting. The architecture utilizes fixed-point arithmetic (Q16.16) and offline training in Python to optimize hardware resources. Validation was performed through simulations using ModelSim and synthesis via Intel Quartus II, targeting a Cyclone IV GX device. The results demonstrated a bit-perfect match between the hardware simulation and the software reference model for an XOR problem. Although the generic architecture incurred a higher resource utilization (21% of Logic Elements) compared to a static equivalent (17%), the study concludes that this trade-off is justified by the significant gain in flexibility, allowing for the efficient reconfiguration of neural network topologies.*
Keywords: *FPGA; VHDL; Neural Networks; Hardware Acceleration; Procedural Generation.*

Resumo. *Com a crescente demanda por soluções eficientes de Inteligência Artificial, a aceleração em hardware utilizando Field-Programmable Gate Arrays (FPGAs) surgiu como uma alternativa poderosa às implementações tradicionais em software. No entanto, uma limitação significativa em muitos projetos de Redes Neurais baseados em VHDL é a sua natureza estática, onde a topologia da rede é fixada no código-fonte, restringindo a escalabilidade e o reuso. Para abordar essa rigidez, este trabalho propõe uma metodologia para a geração procedural de arquiteturas flexíveis de Multi-Layer Perceptron (MLP). Utilizando parâmetros generic e instruções generate do VHDL, o sistema proposto permite a instanciação dinâmica de camadas e neurônios em tempo de síntese, possibilitando a prototipagem rápida sem a necessidade de reescrita manual de código. A arquitetura utiliza aritmética de ponto fixo (Q16.16) e treinamento offline em Python para otimizar os recursos de hardware. A validação foi realizada através de simulações utilizando o ModelSim e síntese via Intel Quartus II, visando um dispositivo Cyclone IV GX. Os resultados demonstraram uma correspondência exata (bit-perfect) entre a simulação de hardware e o modelo de referência em software para um problema XOR. Embora a arquitetura genérica tenha apresentado uma maior utilização de recursos (21% dos Elementos Lógicos) em comparação com um equivalente estático (17%), o estudo conclui que*

essa troca (trade-off) é justificada pelo ganho significativo em flexibilidade, permitindo a reconfiguração eficiente de topologias de redes neurais.
Palavras-chave: *FPGA; VHDL; Redes Neurais; Aceleração em Hardware; Geração Procedural.*

1. Introdução

Com a ascensão do aprendizado de máquina e da inteligência artificial, surgiu uma demanda crescente por soluções computacionais eficientes. Embora muitas aplicações sejam tradicionalmente projetadas para software (CPU/GPU), a complexidade e o custo energético de executar algoritmos de IA dedicados impulsionam a busca por alternativas mais eficientes, com estudos recentes indicando que FPGAs podem oferecer um consumo de energia por inferência significativamente menor que GPUs de alto desempenho [Giasemis et al. 2025].

Nesse âmbito, FPGAs destacam-se na aceleração de redes neurais por superarem CPUs e GPUs em eficiência energética e latência [Antunes and Podobas 2025]. Para explorar esse potencial, adota-se a estratégia de treinamento offline em software com implementação em VHDL, uma abordagem de baixo custo e alta performance já validada em sistemas de tempo real [Ortiz Arciniega et al. 2019].

Contudo, uma análise da literatura revela uma limitação recorrente em muitas implementações de redes neurais em hardware: a natureza estática da arquitetura. Nos projetos desta natureza, a topologia da rede (número de camadas e neurônios) é definida de forma fixa no código-fonte, o que restringe a escalabilidade e o reuso [Fernández-Caro 2023]. A reconfiguração da arquitetura para um novo problema exige, portanto, uma modificação manual e complexa do código, dificultando a prototipagem rápida.

De forma a atacar estas limitações, o foco deste trabalho é apresentar uma metodologia que possibilita a prototipação rápida de aceleradores de hardware para redes neurais, aumentando a escalabilidade e o reuso de código. Inicialmente, os trabalhos relacionados que serviram de base para este estudo são revisados na Seção 2. A metodologia de pesquisa e desenvolvimento é, então, detalhada na Seção 3. Na sequência, a Seção 4 dedica-se à descrição técnica da arquitetura parametrizável implementada. Os resultados obtidos, bem como a discussão sobre o trade-off entre flexibilidade e uso de recursos, compõem a Seção 5. Por fim, as considerações finais e a viabilidade da abordagem são expostas na Seção 6.

2. Trabalhos Relacionados

A busca por metodologias eficientes para implementação de redes neurais em FPGA é um tema ativo na literatura recente, com propostas focadas tanto na tradução automática de modelos de software para hardware [Ghaffari and Savaria 2020] quanto na adaptabilidade em tempo real através de treinamento on-chip [Liu et al. 2022]. As pesquisas que fundamentam este trabalho iniciaram-se com a coleta e análise de projetos públicos de código-aberto, focando nos componentes que estruturam uma rede neural em VHDL: multiplicação de matrizes, regressão linear e MLP (***Multi-Layer Perceptron***) simples. Esta análise resultou na elaboração de três relatórios técnicos internos que detalham os benefícios e desafios de cada implementação em hardware.

2.1. Análise Qualitativa da Implementação de Multiplicador de Matrizes em FPGA

Este estudo apresenta uma análise qualitativa do projeto de hardware para a multiplicação de matrizes utilizando um Dispositivo Lógico Programável (FPGA). A multiplicação de matrizes é uma operação fundamental em diversas áreas da computação, como processamento de

imagens, inteligência artificial e simulações científicas. A operação matemática é dada por $C = A \times B$, onde A e B são as matrizes de entrada de tamanho fixo e C é a matriz resultante. A implementação desta operação em hardware oferece uma oportunidade significativa para aceleração computacional através do paralelismo. [Costa 2023].

2.1.1. Arquitetura e Metodologia do Projeto

A arquitetura do sistema foi projetada de forma modular em VHDL para permitir clareza e reutilização de código. O projeto foi dividido em pacotes e uma entidade principal:

- **MatrixPackage:** Define os tipos de dados e constantes globais, como as dimensões das matrizes (4x4) e a largura dos dados (16 bits).
- **FixedPointOperations:** Provê funções para aritmética de ponto fixo, essencial para representar números não inteiros de forma eficiente em hardware.
- **MatrixMultiplier:** Entidade principal que implementa a lógica da multiplicação, utilizando os pacotes acima.

2.1.2. Aritmética de Ponto Fixo

Uma decisão fundamental foi a utilização de aritmética de ponto fixo. Esta escolha visa otimizar o uso de recursos e a velocidade, uma vez que operações de ponto fixo são menos complexas em hardware do que as de ponto flutuante. A implementação utiliza uma base de ponto fixo (`FIXED_POINT_BASE`) de valor 8. A multiplicação é realizada através da função `multiply_fixed_point`, multiplicando os valores fixos e dividindo pelo valor da base, ajustando a escala do produto.

2.1.3. Paralelismo e Pipeline

A arquitetura do '*MatrixMultiplier*' explora intensivamente o paralelismo de dados. Utilizando a construção `generate` do VHDL, o design instancia um bloco de cálculo dedicado para cada elemento da matriz de resultado. Isso significa que todos os produtos escalares são calculados simultaneamente.

Apesar do alto grau de paralelismo, o bloco de cálculo é implementado como um circuito puramente combinacional. A análise dos relatórios de síntese confirma a ausência total de registradores (`Total logic registers: 0`), indicando que a técnica de *pipeline* não foi utilizada para dividir o caminho lógico em estágios menores.

2.1.4. Análise da Implementação e Hardware Gerado

O projeto foi compilado utilizando o software Intel Quartus II 64-Bit versão 13.1.0, tendo como alvo um dispositivo da família Cyclone IV GX. A compilação resultou no uso de recursos conforme detalhado na Tabela 1, extraída do relatório *Flow Summary*.

A visualização do hardware em nível de transferência de registradores (RTL), gerada pela ferramenta de síntese, confirma a estrutura massivamente paralela do projeto, como visto na Figura 1.

Tabela 1. Relatório de Utilização de Recursos do Fitter.

Recurso	**Utilizado**	**Total Disponível**	**Consumo (%)**
Células Lógicas	796	29,440	3%
Registradores Lógicos	0	29,440	0%
Pinos de I/O	128	307	42%
Bits de Memória	0	1,105,920	0%
Multiplicadores (9-bit)	0	160	0%
PLLs	0	6	0%

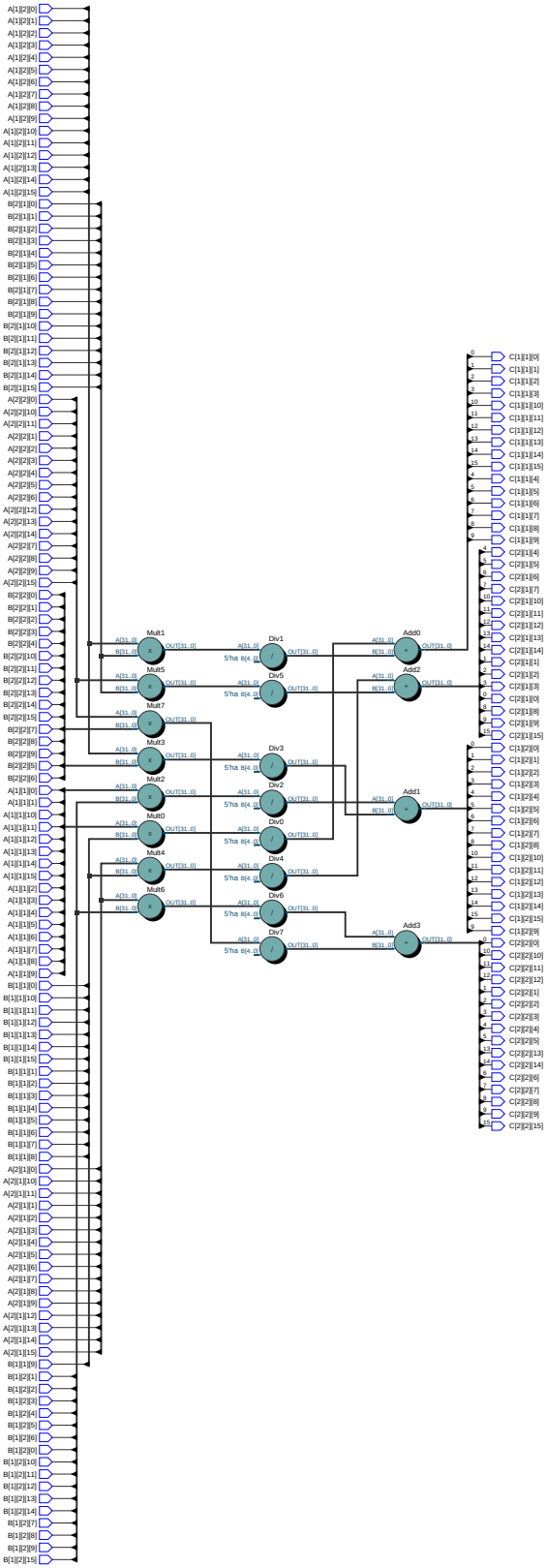

Figura 1. Visualização RTL do Multiplicador de Matrizes mostrando a estrutura paralela.

A análise do esquemático revela em detalhes a implementação física da arquitetura. Na parte superior, encontram-se os dois grandes barramentos de entrada, que correspondem às matrizes A e B. Na inferior, está o barramento de saída para a matriz resultante C. A lógica central representa as unidades de processamento. Cada grupo de blocos é responsável por calcular um único elemento da matriz de saída. Eles executam o produto escalar entre uma linha completa da matriz A e uma coluna completa da matriz B. A ligação densa demonstra como os elementos de múltiplas linhas e colunas são roteados para as unidades de cálculo corretas.

Esta estrutura visual é a contrapartida direta da construção ‘generate‘ no código VHDL, que instancia hardware concorrente para cada par ‘(i, j)‘ da matriz C. A imagem torna explícito como o paralelismo de dados foi mapeado no hardware, com todos os caminhos de cálculo existindo simultaneamente e de forma independente, validando o design como uma arquitetura puramente combinacional e paralela.

2.2. Análise de Acelerador Estático para Regressão Linear

O segundo estudo preliminar analisou um hardware desenvolvido para acelerar o cálculo de regressão linear utilizando FPGA. O objetivo foi avaliar uma arquitetura de *dataflow* fixo, onde a estrutura do hardware espelha diretamente as operações matemáticas da equação normal: $\beta = (X^T X)^{-1} X^T Y$ [Costa 2023].

Esta análise foi essencial para estabelecer um contraponto à arquitetura puramente combinacional (Seção 2.1) e para identificar a principal limitação dos designs estáticos: a inflexibilidade.

2.2.1. Arquitetura e Estrutura Modular

O sistema é composto por um conjunto de entidades VHDL que colaboram para executar o cálculo. A hierarquia é liderada pela entidade de topo `MatrixLinearRegression`, que gerencia o fluxo de dados e controle entre os seguintes módulos principais:

- **`MatrixLinearRegression`**: A entidade de mais alto nível, que instancia os módulos de cálculo e define a sequência de operações (recepção, cálculo, transmissão).
- **`DataReceiver / DataTransmitter`**: Módulos de interface que gerenciam a comunicação serial (UART) para receber as matrizes X e Y e enviar o resultado β.
- **`MatrixOperations`**: Um módulo de controle que orquestra as chamadas para os componentes de hardware na ordem correta da equação:
  1. `MatrixTranspose` (para calcular X^T)
  2. `MatrixMultiplier` (para $X^T X$)
  3. `MatrixInverter` (para $(X^T X)^{-1}$)
  4. `MatrixMultiplier` (para $X^T Y$)
  5. `MatrixMultiplier` (para o resultado final β)
- **`SerialUartCommunication`**: Módulo de baixo nível que implementa o protocolo de comunicação serial.

2.2.2. Escolhas de Projeto Críticas

Interface e Formato dos Dados: A entidade de topo possui uma interface síncrona com `clk` e `reset`, além das linhas `rx` e `tx` para a UART. Internamente, os dados são representados em ponto fixo no formato Q8.8 (16 bits totais), permitindo um balanço entre a faixa dinâmica e a precisão dos cálculos.

Aritmética de Ponto Fixo: A decisão de usar ponto fixo Q8.8 (em vez de ponto flutuante) foi fundamental para otimizar o uso de recursos e a velocidade de operação no FPGA, uma vez que operações de ponto fixo são significativamente menos complexas em hardware [de Assis Pedrobon Ferreira et al. sd].

Paralelismo vs. Sequenciamento: A arquitetura implementa um *dataflow* fixo. Os módulos de multiplicação de matriz (`MatrixMultiplier`) são puramente combinacionais e exploram o paralelismo de dados (instanciando hardware para cada elemento do resultado via `GENERATE`). Em contraste, o módulo de inversão (`MatrixInverter`) é um processo inerentemente sequencial, controlado por uma máquina de estados interna.

2.2.3. Análise de Implementação

O projeto foi compilado no Intel Quartus II 13.1.0, tendo como alvo um dispositivo Cyclone IV GX (EP4CGX30BF14C6).

Utilização de Recursos A Tabela 2 detalha a utilização de recursos do dispositivo. Em contraste direto com a análise da Seção 2.1, esta arquitetura síncrona foi altamente eficiente na utilização de recursos de cálculo. O ponto mais significativo é a utilização de 100 elementos multiplicadores de 9 bits, 76% do total. Este dado comprova que a ferramenta de síntese mapeou com sucesso as multiplicações do VHDL para os blocos DSP dedicados, validando a principal fonte de aceleração. No entanto, similar à outra análise, o projeto utilizou 0% dos bits de memória interna, inferindo o armazenamento em registradores lógicos, o que consumiu 57% dos elementos lógicos do dispositivo.

Tabela 2. Tabela de Utilização de Recursos (Fitter) da Regressão Linear

Recurso	Utilizado	Total Disponível	Consumo (%)
Células Lógicas	16.836	29.440	57%
Registradores Lógicos	5.488	29.440	19%
Pinos de I/O	5	106	5%
Bits de Memória	0	1.179.648	0%
Multiplicadores (9-bit)	100	132	76%
PLLs	0	6	0%

RTL Viewer A visualização RTL (Figura 2) confirma a estrutura modular descrita. É possível identificar claramente os blocos funcionais (‘MatrixOperations‘, ‘DataReceiver‘, etc.) e o fluxo de dados fixo entre eles, validando a arquitetura *dataflow* do projeto.

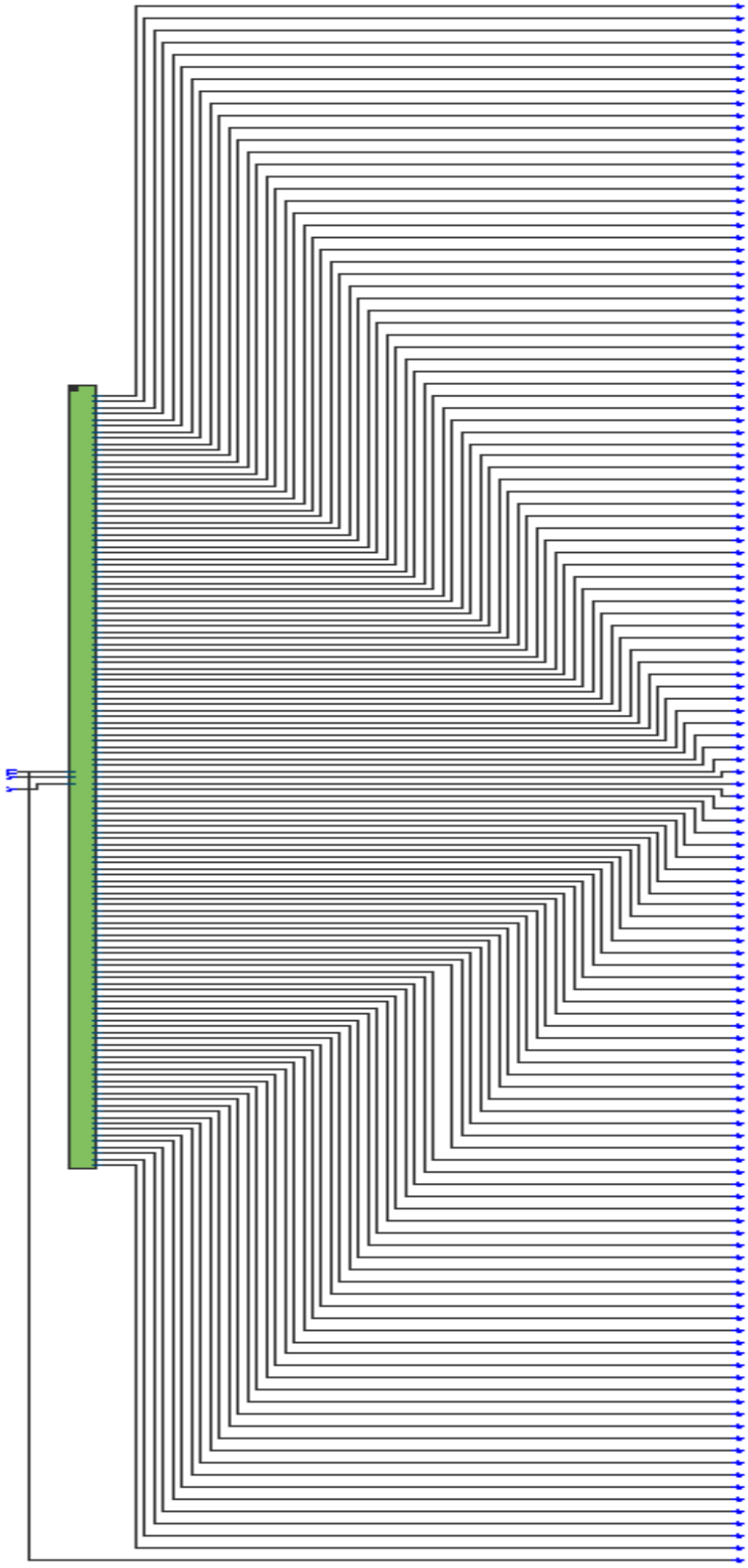

Figura 2. Diagrama RTL do componente de topo MatrixLinearRegression.

A análise deste acelerador foi fundamental para a definição do projeto final. Ela provou que uma arquitetura síncrona e bem estruturada pode, de fato, utilizar eficientemente os blocos DSP dedicados do FPGA (76% de utilização). Contudo, ela também expôs a principal desvantagem dessa abordagem: a rigidez. A arquitetura é estática, construída para resolver uma única equação. Esta inflexibilidade serviu como a principal motivação para o desenvolvimento da arquitetura parametrizável (proposta na Seção 4), que busca manter a eficiência de recursos enquanto permite a reconfiguração da topologia da rede.

2.3. Análise de Rede Neural MLP com Aprendizado On-Chip

A terceira análise da equipe investigou um projeto de hardware para um acelerador de Rede Neural Artificial (RNA) do tipo Multi-Layer Perceptron (MLP) [Slavescu and Bastone 2023], com foco na implementação de um mecanismo de aprendizado on-chip em um Dispositivo Lógico Programável (FPGA). O objetivo foi avaliar a arquitetura modular, as escolhas de design e os resultados de desempenho obtidos após a síntese na ferramenta Intel Quartus II.

A implementação de algoritmos de Machine Learning em hardware, especialmente redes neurais com treinamento, apresenta uma oportunidade significativa para aceleração computacional. Nesta análise, o foco foi validar uma arquitetura para uma camada neural com múltiplas unidades de processamento (neurônios) e seus respectivos mecanismos de aprendizado (algoritmo de gradiente descendente), todos operando em paralelo.

2.3.1. Arquitetura e Estrutura Modular

O design do sistema analisado é modular e hierárquico em VHDL, com uma entidade de topo (`MLP`) que orquestra o fluxo de dados entre os sub-módulos. Os principais componentes são:

- **`layer`**: Implementa a camada neural. Utiliza um laço `GENERATE` para instanciar múltiplos neurônios que operam em paralelo.
- **`neuron`**: A unidade de processamento fundamental que, nesta implementação simplificada, processa uma única entrada e um único peso.
- **`gradient_descent`**: O módulo de aprendizado para o peso de um único neurônio. Contém uma pipeline interna de 5 estágios. Instâncias deste módulo são geradas em paralelo, uma para cada neurônio.
- **`weight_register`**: Um registrador que atrasa o vetor de entrada, sincronizando os dados para o cálculo do gradiente.

2.3.2. Escolhas de Projeto Críticas

Aritmética e Largura de Dados: Uma decisão fundamental do projeto foi o uso de aritmética de inteiros com sinal (`signed`) com 32 bits de largura (`DATA_WIDTH`), utilizando a biblioteca `IEEE.NUMERIC_STD`. Para conter o crescimento da largura de bits, os resultados das multiplicações 32x32 bits (que geram 64 bits) são explicitamente truncados ou redimensionados de volta para 32 bits nas etapas subsequentes.

Paralelismo e Pipeline: A arquitetura explora intensivamente o paralelismo de dados. A construção `GENERATE` é utilizada em dois níveis: na entidade `layer` (para instanciar neurônios) e na entidade `MLP` (para instanciar as unidades `gradient_descent`). Isso significa que o processamento (forward pass) e o aprendizado ocorrem simultaneamente para todos os neurônios. Além disso, o módulo `gradient_descent` é profundamente pipelined (aprox. 5 estágios), permitindo que o sistema aceite novas entradas a cada ciclo de clock, mantendo um alto throughput.

2.3.3. Resultados da Análise de Implementação

O projeto foi compilado no Intel Quartus II 13.1.0, tendo como alvo um dispositivo Cyclone IV GX (EP4CGX30CF23C6), com uma configuração de camada de 2 neurônios.

Utilização de Recursos A Tabela 3 detalha a utilização de recursos do dispositivo. Um ponto significativo da análise é que nenhum dos blocos DSP dedicados foi utilizado (0%). As quatro operações de multiplicação 32x32 bits do design foram implementadas utilizando lógica geral, o que explica o consumo de 4.526 elementos lógicos (15%). O não uso de memória RAM indica que os pesos foram inferidos como um banco de registradores.

Tabela 3. Tabela de Utilização de Recursos (Fitter) da MLP analisada

Recurso	Utilizado	Total Disponível	Consumo (%)
Total de Elementos Lógicos (LEs)	4.526	29.440	15%
Total de Registradores	628	29.440	2%
Pinos de I/O	130	292	45%
Bits de Memória (M9K)	0	1.179.648	0%
Multiplicadores (9-bit)	0	132	0%
PLLs	0	6	0%

RTL Viewer A visualização RTL (Register-Transfer Level) gerada pelo Quartus, apresentada na Figura 3, confirma a estrutura modular e paralela. É possível identificar os blocos funcionais (como `gradient_descent`) e o fluxo de dados entre eles, validando a arquitetura analisada.

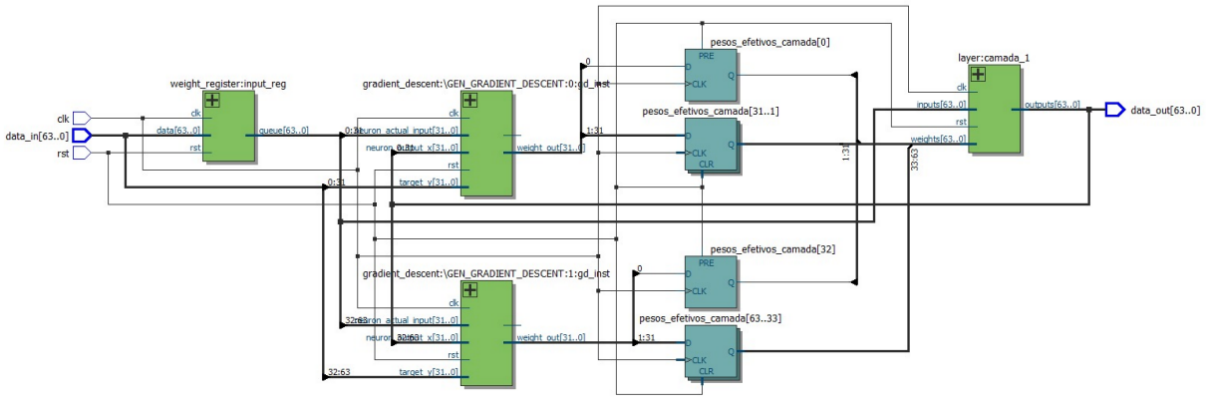

Figura 3. Diagrama RTL da MLP externa com aprendizado on-chip.

Recentemente, abordagens dinâmicas buscam mitigar a rigidez do hardware. O NeuroCoreX, por exemplo, permite reconfiguração via software em arquiteturas neuromórficas [Gautam et al. 2025], enquanto novas metodologias utilizam DSLs para geração rápida de filtros [Campos et al. 2024]. Apesar da flexibilidade, essas soluções tendem a aumentar o consumo de recursos, contrastando com a eficiência da geração procedural proposta neste trabalho.

3. Metodologia

A condução deste trabalho visou responder ao problema central do projeto: a possibilidade de implementação de hardware em VHDL que seja facilmente parametrizável, permitindo o processamento de uma arquitetura genérica de rede neural artificial. O objetivo principal, portanto, foi investigar, implementar e validar tal arquitetura, demonstrando uma solução que supera a rigidez dos designs estáticos estudados anteriormente.

Para atingir os objetivos propostos, foi estabelecida uma sequência de procedimentos metodológicos. A primeira etapa consistiu em uma revisão da literatura para identificar o estado das implementações investigadas e os desafios na construção de redes neurais em FPGAs. Sequencialmente, foi realizada uma análise aprofundada das implementações existentes (conforme detalhado na Seção 2), que revelou a rigidez de designs estáticos como a principal lacuna. O procedimento central do trabalho foi, então, o desenvolvimento de uma nova arquitetura VHDL que aplica os princípios do design genérico, permitindo a parametrização completa da topologia da rede. Por fim, o fluxo de trabalho definiu o treinamento dos pesos da rede de forma offline, tipicamente em Python, e a subsequente utilização desses pesos como constantes de configuração quando a entidade VHDL parametrizada é instanciada para simulação ou síntese. [Silva 2019]

Para a execução dos procedimentos, foi empregado um conjunto de ferramentas e linguagens de software e hardware. A linguagem VHDL (VHSIC Hardware Description Language) foi utilizada como padrão para a descrição da arquitetura de hardware. O fluxo de validação funcional e simulação do código foi realizado com o auxílio dos softwares GHDL e ModelSim, com a visualização e análise das formas de onda feitas medidas através GTKWave. Para a etapa de implementação e análise de hardware, o software Quartus II 13.1 (64-bit) Web Edition foi utilizado para a síntese, alocação de recursos e obtenção dos relatórios de desempenho e ocupação da FPGA, tendo como alvo o dispositivo Cyclone IV GX EP4CGX110DF31C7. O treinamento prévio da rede para comparação e validação de resultados foi realizado em Python.

Por fim, a validação foi realizada em duas etapas principais. Primeiro: a validação funcional, na qual o comportamento do código foi verificado por meio de simulações (ModelSim/GHDL) com as saídas sendo comparadas com um script de referência em Python, utilizando os mesmos pesos e entradas para garantir que o hardware produzisse os resultados corretos. Em seguida, realizou-se uma análise da implementação, focada na flexibilidade, eficiência e viabilidade. O projeto foi, então, sintetizado no Quartus II diversas vezes, alterando os parâmetros da rede (como número de neurônios e camadas). Para cada versão, foram observados os relatórios de síntese para medir a ocupação de recursos da FPGA referencial (Elementos Lógicos, memórias) e a frequência máxima de operação (F_{max}), comprovando o impacto da parametrização no hardware final.

4. Proposta de Arquitetura Genérica em VHDL para MLP

A arquitetura VHDL proposta neste trabalho foi adaptada para resolver a rigidez dos designs estáticos analisados na Seção 2. A principal inovação reside na capacidade de gerar a topologia da rede proceduralmente em tempo de síntese, permitindo uma reconfiguração eficiente e compacta.

4.1. Componentes Fundamentais

types.vhd: A base para as operações aritméticas da rede foi consolidada no pacote `types.vhd`, que define os tipos de dados customizados utilizados em todo o projeto. Sua

função é **centralizar todas as definições de tipos de dados customizados**, garantindo consistência e facilitando a manutenção. Para garantir a eficiência, foi adotada a representação numérica em ponto-fixo, utilizando o tipo `sfixed` da biblioteca **`ieee.fixed_pkg`**. Estudos recentes demonstram que essa abordagem reduz significativamente a latência e o uso de recursos lógicos em comparação ao ponto flutuante, sem perda crítica de precisão para inferência [Ghaffari and Savaria 2020]. Conforme definido no pacote, o formato é configurado através das constantes `int := 16` e `frac := 16`, resultando em um barramento de 32 bits totais (`sfixed(15 downto -16)`). O tipo `sfixed` é um formato com sinal, que utiliza a representação em complemento de dois para suportar valores negativos. Esta escolha (um formato Q16.16) oferece um balanço otimizado entre a precisão requerida para os cálculos da rede neural (como as operações de multiplicação-acumulação e as funções de ativação) e o uso eficiente dos recursos lógicos disponíveis no hardware.

act_func.vhd: Para garantir a modularidade e a extensibilidade do projeto, as funções de ativação foram encapsuladas em uma entidade VHDL dedicada, a `act_func.vhd`. Esta abordagem desacopla a lógica de ativação da unidade de processamento do neurônio, resultando em um código mais claro e de fácil manutenção. A técnica de design, **conforme o diagrama da Figura 4**, utiliza uma única `entity` com múltiplas `architectures`, onde cada `architecture` implementa uma função de ativação específica (como `Threshold` e `ReLU`). No escopo deste trabalho, o neurônio instancia esta entidade usando a arquitetura padrão (`relu`), mas a vantagem desta flexibilidade é que o sistema permanece expansível, pois novas funções podem ser adicionadas como novas `architectures` sem exigir modificação nos outros componentes da rede.

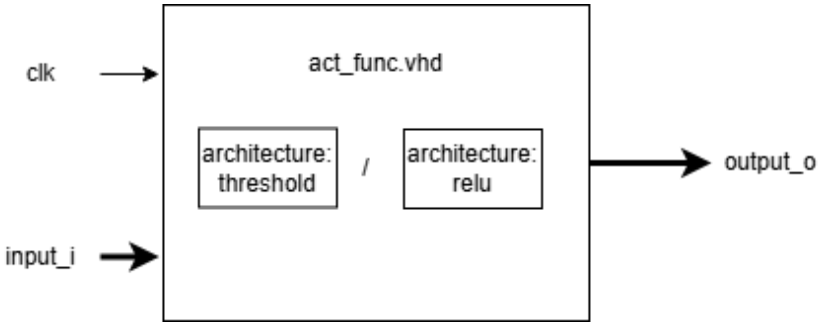

Figura 4. Diagrama da entidade flexível `act_func.vhd` e suas múltiplas arquiteturas.

neuron.vhd: A unidade de processamento fundamental da arquitetura é a entidade `neuron.vhd`, que emula o comportamento matemático de um neurônio artificial. Seu funcionamento é governado por uma máquina de estados finitos (FSM) sendo sua função, calcular o produto escalar entre suas entradas e pesos, somar o viés, e aplicar uma função de ativação. Projetado com foco em reutilização e escalabilidade, o componente tem como principal característica a utilização do parâmetro `generic` para definir seu número de entradas. **A interface da entidade, detalhada no diagrama da Figura 5**, permite que um único código-fonte de neurônio seja instanciado em qualquer camada da rede, adaptando-se dinamicamente ao número de conexões da camada anterior, o que representa uma vantagem significativa em comparação com designs estáticos.

- **Eficiência (Reuso de Hardware):** Em vez de instanciar `N` multiplicadores em paralelo (o que consumiria muitos blocos DSP do FPGA), o `neuron.vhd` usa uma **Máquina de Estados Finitos (FSM)**. Esta FSM controla um laço de **Multiplicação-Acumulação (MAC) sequencial**. Este design reutiliza *um único* multiplicador ao

longo de `N` ciclos de clock, um *trade-off* (troca de latência por área) que economiza drasticamente os recursos de hardware.

- **Flexibilidade (Generics):** O componente é duplamente genérico:
    1. O `generic (inputs : integer)` permite que este *mesmo* arquivo VHDL seja instanciado em qualquer camada, adaptando-se automaticamente ao número de entradas.
    2. O `generic (use_threshold : boolean)` é usado em um comando `if...generate` para selecionar qual arquitetura do `act_func.vhd` (ReLU ou Threshold) deve ser fisicamente instanciada. Isso permite que a camada de saída se comporte de forma diferente das camadas ocultas usando o mesmo código-fonte.

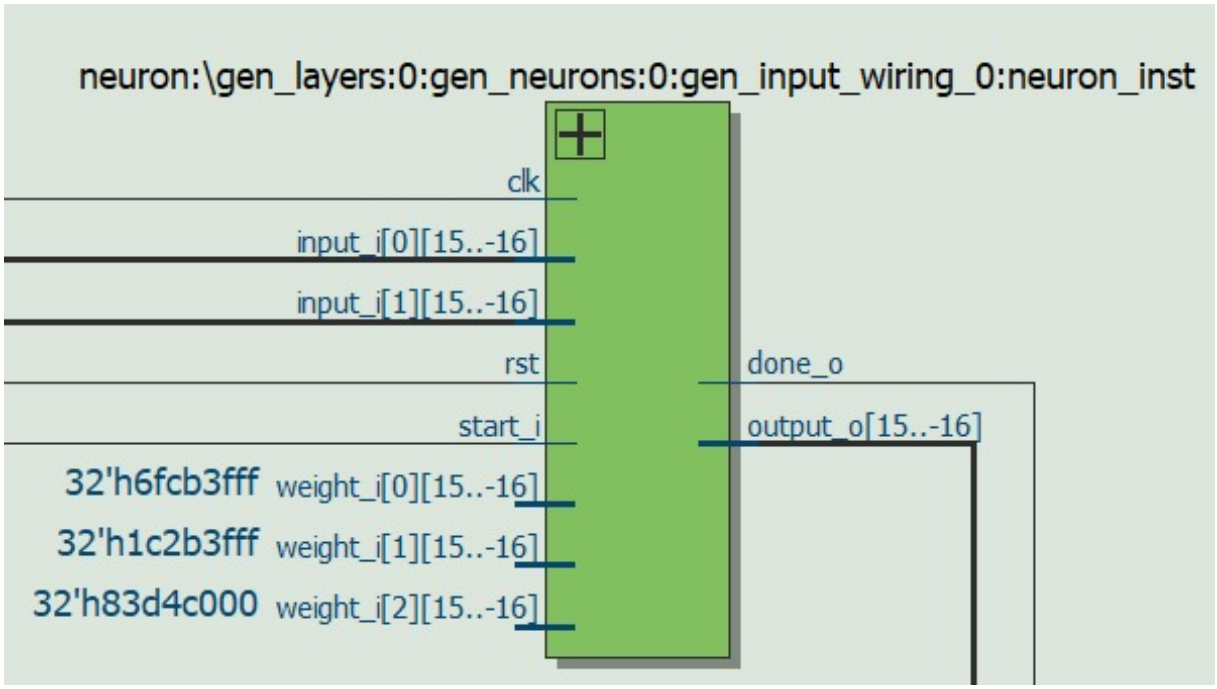

Figura 5. Diagrama de entidade (caixa-preta) do componente `neuron.vhd`.

network.vhd: Esta entidade é de notória relevância, sua função é atuar como um construtor de rede procedural. Ele resolve o problema central do trabalho (rigidez estática), sendo uma entidade capaz de gerar qualquer topologia de MLP em tempo de síntese, com base em parâmetros `generic`. O design é centrado em seus `generic` principais:

- `NEURONS_PER_LAYER`: Um *array* de inteiros que define a topologia completa (ex: (2, 2, 1)).
- `weights`: Um *array* `sfixed` unidimensional (plano) que contém todos os pesos e viés da rede, lidos de um arquivo Python.

O grande diferencial dessa entidade está no uso de laços `for...generate` aninhados (`gen_layers` e `gen_neurons`). Tais comandos VHDL instruem o sintetizador Quartus a:

1. **Instanciar Camadas:** Criar uma camada para cada item no *array* `NEURONS_PER_LAYER`;
2. **Instanciar Neurônios:** Dentro de cada camada, instanciar o número correto de componentes `neuron.vhd`;

3. **Gerenciar Ligação (*Wiring*):** Conectar automaticamente a entrada principal (`input_i`) à primeira camada e, para as demais, conectar a saída da camada anterior (`layer_outputs(i-1)`) à entrada da camada atual;
4. **Distribuir os Pesos:** Usar funções ('get_layer_start_offset') para calcular e "fatiar"o *array* `weights` principal, entregando o bloco de pesos correto para cada neurônio individual;
5. **Controlar o Fluxo:** Gerar automaticamente a lógica de controle ('gen_start_control') que implementa um pipeline por camada, onde uma camada só inicia ('neuron_start') quando a anterior termina ('layer_all_done(i-1)').

network_top.vhd: Esta é a entidade principal (top-level) do projeto e atua como um empacotador. Sua única função é servir como a ponte entre o design genérico (`network.vhd`) e uma implementação de hardware *específica* e concreta. Enquanto `network.vhd` é abstrato, ou seja, pode ser qualquer rede, o `network_top.vhd` define seus parâmetros generic. Como visto no código, esta entidade caracteriza as constantes para uma topologia específica: `INPUT_SIZE_C := 2` e `NEURONS_PER_LAYER_C := (2, 1)` (uma rede XOR 2-2-1). Mais importante, é neste arquivo que os pesos treinados previamente, utilizando Python, são instanciados no design, através da constante `WEIGHTS_C`. Ele então instancia a entidade `network` e passa essas constantes para seu generic map. Para sintetizar uma rede diferente da 2-2-1 já instanciada, como a 2-3-2-1, não é necessário modificar a lógica central do `network.vhd` ou `neuron.vhd`. Basta criar uma nova configuração com constantes `NEURONS_PER_LAYER` e `WEIGHTS_C` diferentes, provando a flexibilidade e a prototipação rápida do design.

Portanto, a arquitetura é hierárquica, composta pela entidade principal `network_top` que configura e instancia a entidade `network`, a qual gerencia a instância e a conectividade de múltiplos componentes `neuron.vhd`. A topologia da rede MLP é definida através de um parâmetro `generic` central, denominado `NEURONS_PER_LAYER`, que consiste em um *array* de inteiros. O *array* define o número de neurônios em cada camada, da primeira camada oculta até a camada de saída. O número de entradas da rede é inferido pelo tamanho da primeira camada.

A construção física da rede é realizada por meio de laços `for...generate` aninhados. A Figura 6 ilustra essa arquitetura procedural. A entidade `neuron` atua como um entidade, nela:

- Um laço externo itera sobre o *array* `NEURONS_PER_LAYER` para criar cada camada.
- Um laço interno instancia o número correspondente de entidades `neuron.vhd` para aquela camada.

Os sinais intermediários (barramentos `layer_outputs`) são igualmente gerados de forma procedural. Isso estabelece a conectividade *feed-forward* totalmente conectada, ligando as saídas dos neurônios de uma camada *i* às entradas da camada *i+1*. O parâmetro `generic inputs` de cada neurônio também é configurado automaticamente com o número de saídas da camada anterior, completando a automação do design.

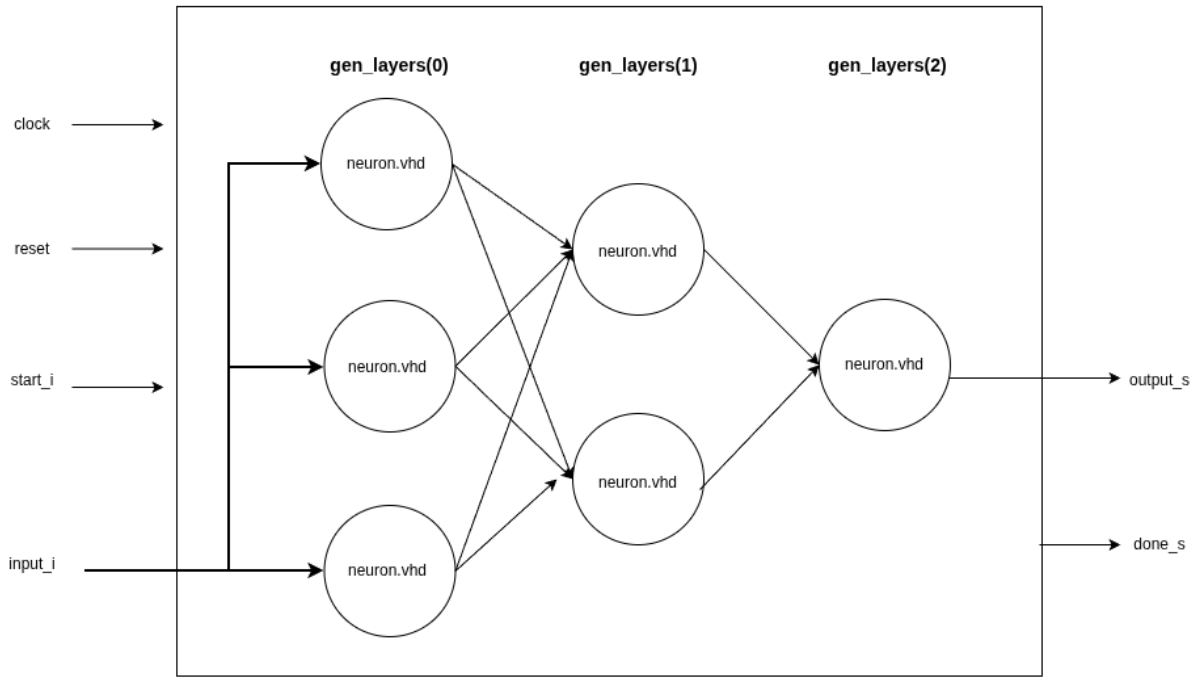

Figura 6. Diagrama de blocos entidade `network.vhd`, ilustrando a instanciação dos neurônios de uma simulação 3-2-1.

5. Simulações e Resultados

5.1. Arquitetura Genérica da Rede Neural

A partir do hardware desenvolvido anteriormente, foi dado início à etapa de conferência de resultados e recursos, de modo a validar o que foi produzido. Para isso, foi utilizado o software ModelSim, juntamente ao Quartus, para realizar a simulação da rede neural e analisar os pontos de acertos, erros e imprecisões. Primeiramente, foi realizada uma análise de recursos utilizados pela rede; essa etapa diz respeito à plataforma Quartus, na qual é feita a compilação de todo o código produzido e a ferramenta gera o relatório de síntese, além de um modelo esquemático da estrutura do projeto. A partir disso, foram obtidos os dados expressos na Tabela 4:

Tabela 4. Utilização de Recursos Arquitetura Flexível 2-2-1.

Recurso	Utilizado	Total disponível	Consumo
Total logic elements	4.379	109.424	2%
Total combinational functions	4.376	109.424	2%
Dedicated logic registers	30	109.424	$<$1%
Total registers	15	-	-
Total pins	100	508	19%

Com base nesses resultados, podemos concluir que não houve sobrecarga da placa utilizada para síntese/implementação, a EP4CGX110DF31C7, mantendo-se ampla margem de utilização de recursos lógicos.

5.2. Simulação e Visualização (ModelSim)

Em sequência, houve a validação funcional ao interagir o hardware gerado com a simulação do comportamento da FPGA, ou seja, com estímulos através de pulsos e valores de entrada para validação dos valores gerados na saída. Essa etapa acontece por meio do ModelSim e de um modelo VHDL denominado *testbench* (`network_tb.vhd`), responsável por fornecer os estímulos e monitorar as respostas. Paralelamente, foi realizado o treinamento dos pesos (em Python), que são passados para a entidade `network_top`. Esse algoritmo realiza o treino de um MLP 2–2–1 com ReLU nas camadas ocultas e saída linear (*logit*), usando a loss `BCEWithLogitsLoss`; em seguida, valida os quatro padrões do XOR antes e depois da quantização em ponto fixo Q16.16 (arredondamento para 2^{-16}), e só então exporta os pesos no formato esperado pelo VHDL (ordem por neurônio: **bias, w0, w1**, camada a camada). Essa validação pós-quantização evita divergências entre o modelo em ponto flutuante e a implementação em ponto fixo.

Com essa preparação, foi gerada a simulação para o modelo XOR 2–2–1, isto é, a rede operando com duas entradas, uma camada intermediária com dois neurônios e uma camada de saída com um neurônio, obtendo-se o resultado ilustrado na Figura 7.

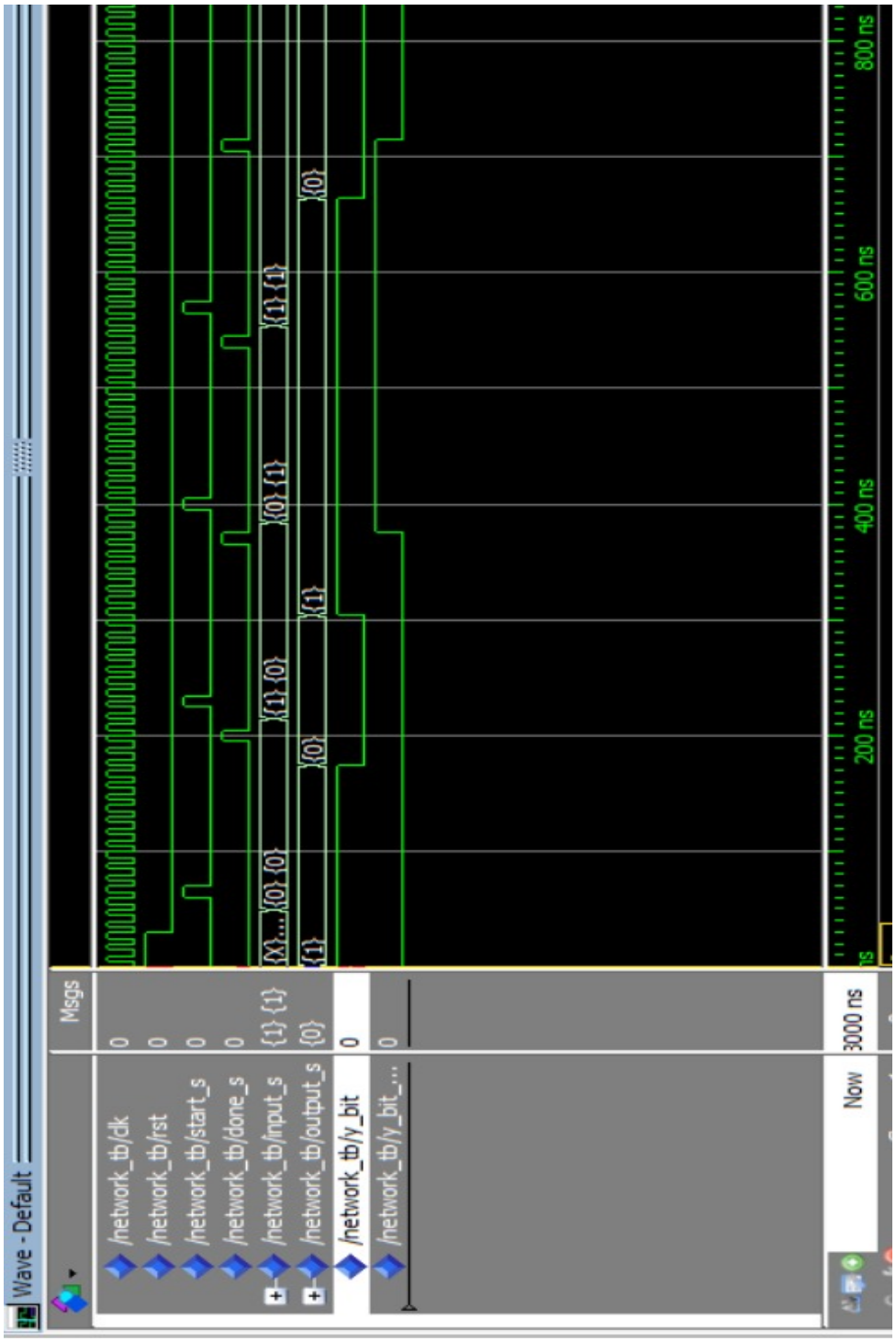

Figura 7. Wave XOR 2, 2, 1.

Utilizando a ferramenta *Wave* do ModelSim, é possível verificar os pulsos do sistema e a sequência de estímulos. No experimento, são aplicados, nesta ordem, os vetores de entrada {00}, {01}, {10} e {11}. Os principais sinais observados são:

- `clock`: responsável pela cadência temporal do circuito;
- `reset`: (permite inicialização do sistema ao ser ativado e garante retorno a um estado conhecido;
- `start_s`: pulsado para iniciar o processamento em cada caso de teste;

- `done_s`: indica a conclusão do cálculo da rede, também por pulso;
- `input_s`: vetor que carrega as amostras de entrada para a DUT;
- `output_s`: responsável por exibir os valores de saída gerados pela rede após a propagação direta e aplicação dos pesos.

A resposta é considerada válida no ciclo em que `done_s` sobe; o *testbench* amostra e mantém esse valor para facilitar a inspeção em forma de *waveform*. Quando a última camada é configurada com `threshold`, `output_s(0)` já assume 0.0 ou 1.0 em ponto fixo; alternativamente, quando a saída permanece linear, o *testbench* deriva o bit de decisão por um comparador com zero (regra $y{=}1$ se logit > 0), garantindo equivalência com o critério de treinamento.

Como critério de aceitação, verificou-se que, para os quatro padrões do XOR, a decisão binária segue a sequência esperada $\{00\} \rightarrow 0$, $\{01\} \rightarrow 1$, $\{10\} \rightarrow 1$, $\{11\} \rightarrow 0$. Para assegurar reprodutibilidade, o *testbench* realiza a amostragem no instante de `done_s` e pode opcionalmente incluir *asserts* por caso de teste. Por fim, ressalta-se que a consistência entre simulação e hardware depende (i) da ordem correta de empacotamento dos pesos (por neurônio: **bias, w0, w1, ...**), (ii) da quantização Q16.16 aplicada no software antes da exportação, e (iii) da seleção de ativações (ReLU nas ocultas e `threshold` apenas na última camada), que materializa no hardware a mesma regra de decisão usada no treinamento.

5.3. Análise do circuito RTL

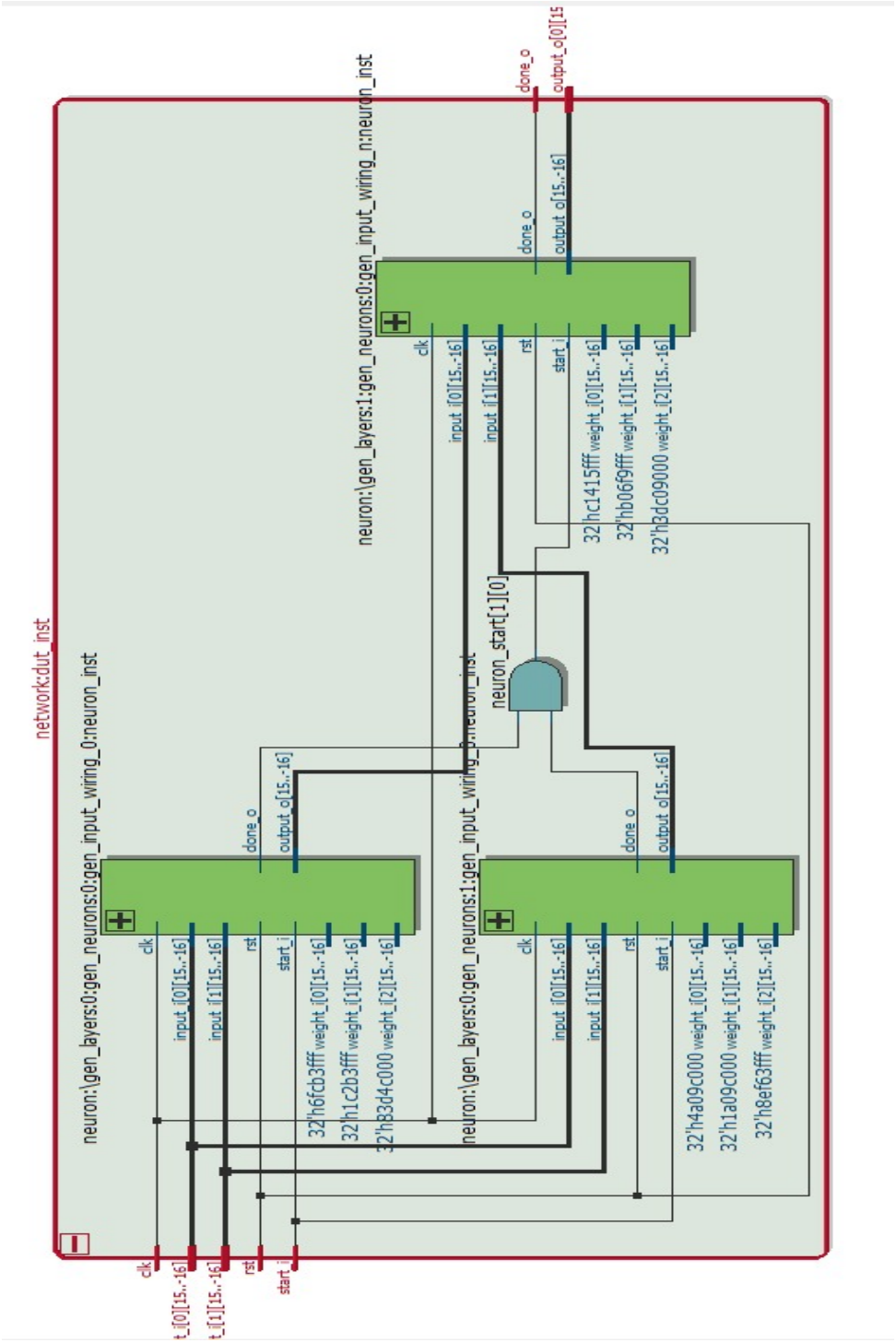

Figura 8. Hardware sintetizado (RTL Viewer) para a rede de topologia 2, 2, 1.

A visualização RTL gerada pelo Quartus na Figura 8 confirma a organização da rede 2–2–1: dois neurônios na camada oculta (`neuron:n1` e `neuron:n2`) alimentando um neurônio de saída (`neuron:n3`). Os sinais `clk`, `rst`, `start_i` e as entradas `input_i[0]` e `input_i[1]` (todas em `sfixed(15 downto -16)`) chegam em paralelo aos neurônios da primeira camada. As saídas `output_o` dos neurônios ocultos são encaminhadas como

`input_i[0..1]` do neurônio de saída, preservando a mesma largura Q16.16, enquanto o `done_o` do neurônio final é propagado ao topo como indicação de término do cálculo da rede.

Observa-se, ainda, a lógica de controle `start_n3` entre as camadas, que habilita o disparo do neurônio de saída apenas após a conclusão dos neurônios da primeira camada (combinação dos sinais `done_o` de `n1` e `n2`). Essa organização implementa o protocolo *start/-done* por camada, evitando o consumo simultâneo antes de os dados intermediários estarem válidos.

No que se refere aos pesos, cada neurônio expõe um barramento `weight_i[0..inputs]` em Q16.16, onde o índice mais alto representa o *bias*. No diagrama, as conexões aparecem como constantes do tipo `32'hXXXX`, refletindo o valor quantizado:

- `32'h00008000` $\Rightarrow +0{,}5$ (Q16.16),
- `32'hFFFF0000` $\Rightarrow -1{,}0$,
- `32'h00010000` $\Rightarrow +1{,}0$,
- `32'h0001FFFF` $\Rightarrow \approx 1{,}9999847$,
- `32'h00007FFF` $\Rightarrow \approx 0{,}4999847$.

A partir dessa notação, é possível conferir, diretamente no RTL, se os valores sintetizados correspondem aos pesos exportados pelo treinamento. É importante lembrar que a ordem por neurônio no projeto é **(bias, w0, w1)**, compatível com os *slices* utilizados no `network.vhd`.

Portanto, a análise confirma que a estrutura topológica e o fluxo de dados do RTL corroboram o modelo especificado:

1. **Camada oculta**: dois neurônios recebendo `input_i[0]` e `input_i[1]`, Multiplicação das entradas pelos respectivos pesos + ativação (ReLU);
2. **Controle**: `start_i` dispara a primeira camada; o `start_n3` habilita a camada de saída após ambos `done_o` estarem ativos;
3. **Camada de saída**: um neurônio que recebe os dois intermediários, aplica MAC e, na configuração final, `threshold` para decisão binária (0/1).

Como melhorias de engenharia (boas práticas) observadas a partir do RTL:

- **Estabilização do disparo**: O registro de `start_n3` em flanco de `clk` elimina *glitches* e assegura o sincronismo entre camadas, gerando o sinal no ciclo seguinte à conclusão da etapa anterior.
- **Verificação de pesos**: A conferência dos literais `32'hXXXX` no RTL contra os valores treinados (convertidos para hexadecimal) é essencial para garantir que a síntese reflete a versão correta do `network_top.vhd`.
- **Funções de ativação**: O *netlist* deve confirmar a instância correta de `act_func(threshold)` na saída e `act_func(relu)` nas camadas ocultas.
- **Latência total**: O RTL demonstra que a camada de saída aguarda o `done` da oculta, resultando em uma latência cumulativa que deve ser validada na medição de ciclos do *testbench*.

Em resumo, o RTL comprova a implementação estrutural 2–2–1 com protocolo *start/-done* por camada, pesos constantes em Q16.16 ligados às portas `weight_i` (com o *bias* no índice superior) e seleção de ativação conforme o nível da rede. A validação cruzada dos literais em hexadecimal com os pesos exportados do treinamento (`WEIGHTS_C`) é recomendada para assegurar que a versão compilada reflete exatamente o conjunto de pesos utilizado na simulação funcional.

Houve, também, a sintetização de duas redes maiores, configuradas com múltiplas camadas. A primeira MLP possui a topologia 2–3–2–1, representada pela Figura 9, composta por duas camadas intermediárias (a primeira com três neurônios e a segunda com dois neurônios) e uma camada de saída com um neurônio, operando sobre duas entradas. A segunda rede ampliada segue a topologia 10–10–6–4–3–2–1, representada pela Figura 10, partindo de dez entradas e distribuindo o processamento ao longo de cinco camadas ocultas de tamanhos decrescentes, culminando em uma camada de saída com um único neurônio. Em ambas as arquiteturas, observa-se a mesma organização estrutural do modelo 2–2–1: barramentos `sfixed` em Q16.16 nas entradas e entre camadas, vetores de pesos/bias conectados como constantes aos `weight_i` de cada neurônio, e o protocolo de controle por camada (start/done), que encadeia o disparo da camada subsequente somente após a conclusão da precedente. A última camada permanece com `threshold` para decisão binária, enquanto as camadas ocultas utilizam ReLU. Em relação ao 2–2–1 original, nota-se o aumento progressivo de fan-in/fan-out nas duas novas topologias, mais sinais de `start` intermediários e, consequentemente, maior latência total, mantendo, contudo, a mesma largura de dados e o mesmo critério de decisão. Os recursos utilizados por essas novas redes estão representados, 2-3-2-1, na Tabela 5 e, 10-6-4-3-2-1, pela Tabela 6.

Tabela 5. Utilização de Recursos Arquitetura Flexível 2-3-2-1.

Recurso	Utilizado	Total disponível	Consumo
Total logic elements	5 995	109 424	2%
Total combinational functions	5 994	109 424	2%
Dedicated logic registers	30	109 424	<1%
Total registers	30	–	–
Total pins	100	508	19%

Tabela 6. Utilização de Recursos Arquitetura Flexível 10-10-6-4-3-2-1.

Recurso	Utilizado	Total disponível	Consumo
Total logic elements	25 363	109 424	23%
Total combinational functions	25 344	109 424	23%
Dedicated logic registers	130	109 424	<1%
Total registers	130	–	–
Total pins	356	508	70%

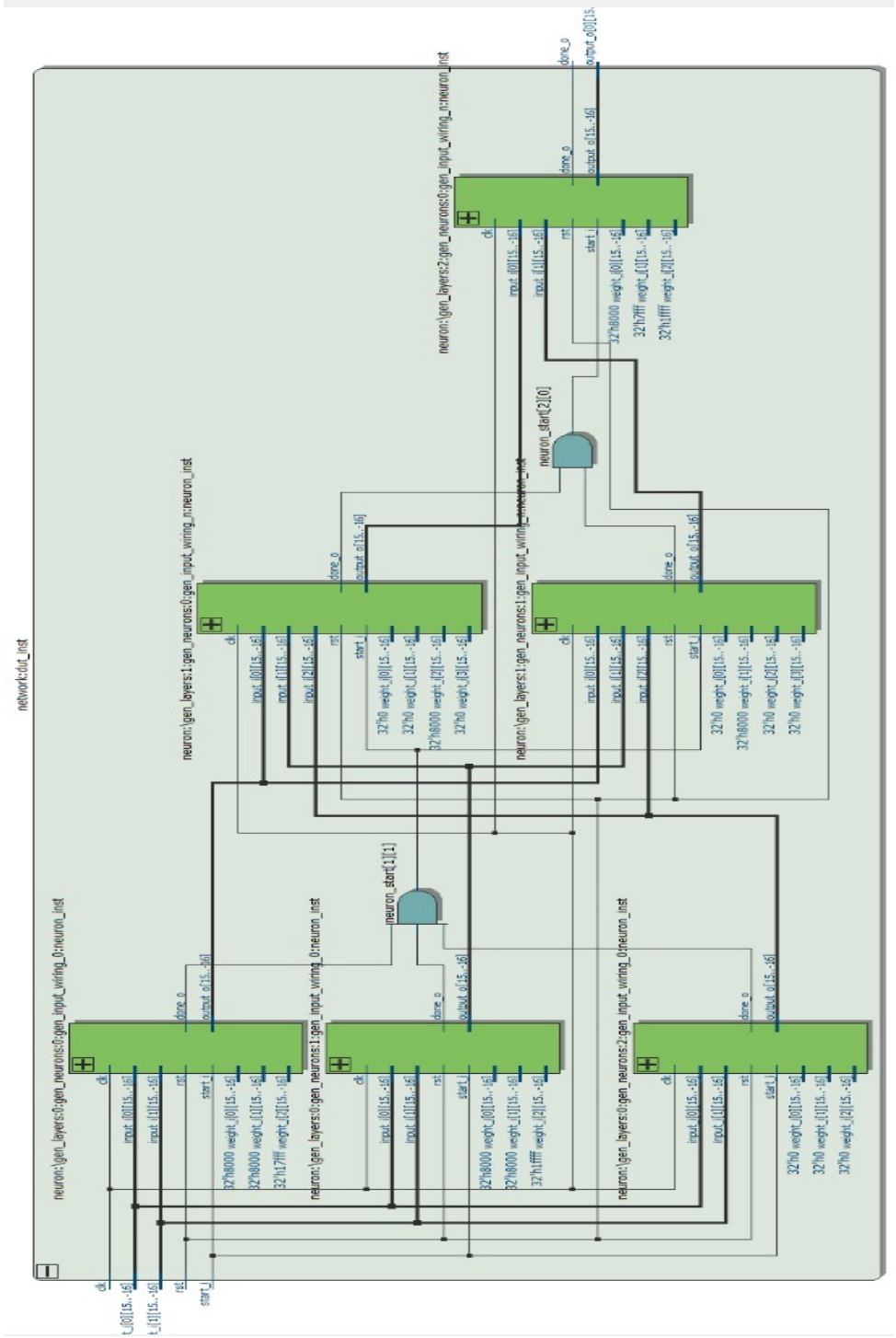

Figura 9. Hardware Flexível sintetizado para a rede 2, 3, 2, 1.

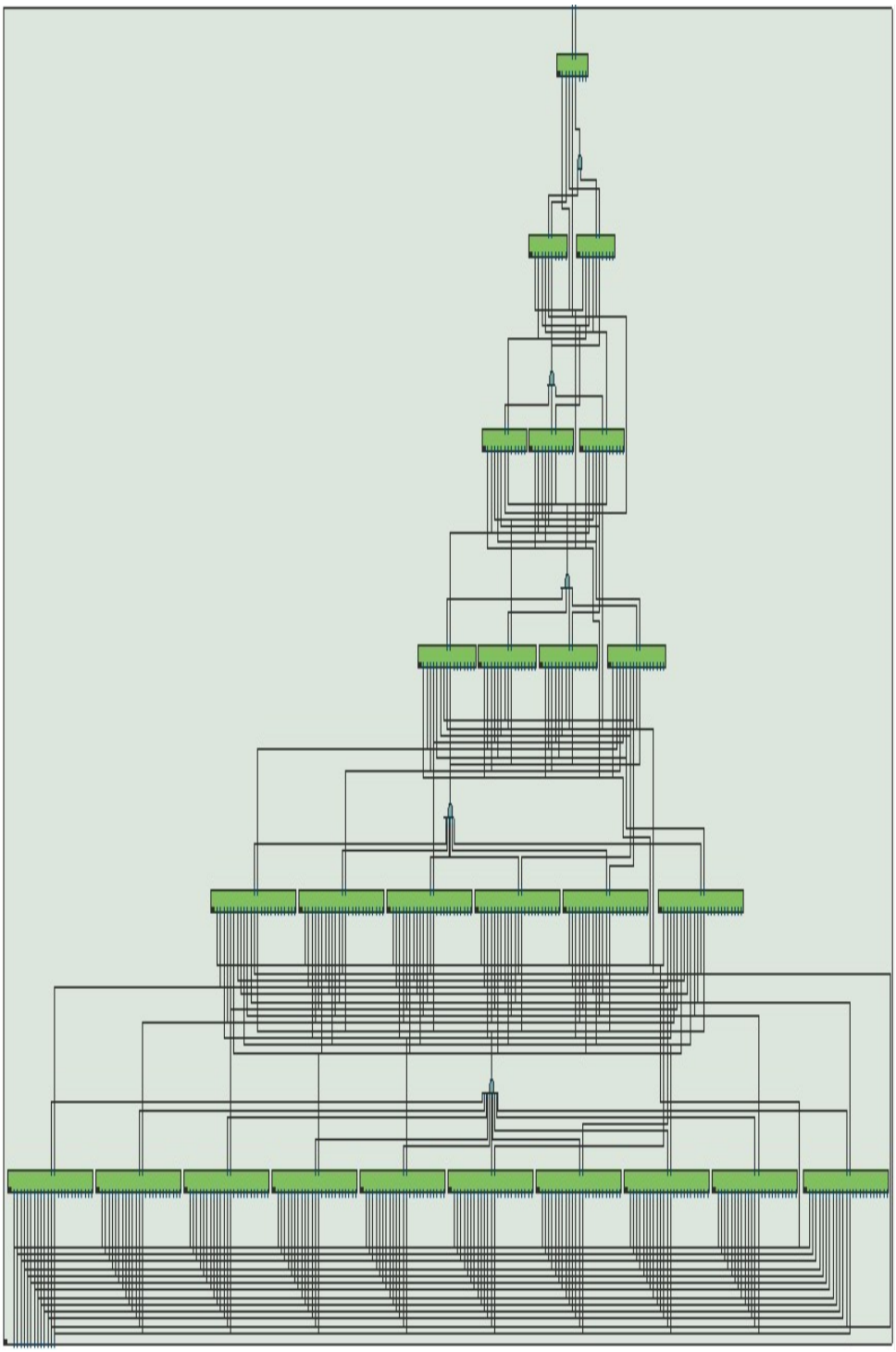

Figura 10. Hardware Flexível sintetizado para a rede 10-10-6-4-3-2-1.

5.4. Arquitetura Estática da Rede Neural

O projeto utilizado como referência foi sintetizado e simulado, também, na ferramenta Quartus, e teve seus dados de utilização obtidos para fins comparativos. Dessa forma, é possível estabelecer uma análise entre os processos a fim de identificar possíveis vantagens e desvantagens, considerando que, neste trabalho, foi realizada a adaptação para um modelo parametrizável. O trabalho de referência foi configurado para a mesma topologia XOR 2–2–1 (duas entradas, uma camada intermediária com dois neurônios e uma camada de saída com um neurônio)

e direcionado ao mesmo dispositivo (EP4CGX22CF19C6), garantindo condições equivalentes de comparação. Os resultados reportados pelo Quartus para o modelo *não parametrizável* encontram-se na Tabela 7. A partir desses dados, foi construído o gráfico comparativo de barras mostrado na Figura 11.

Tabela 7. Utilização de Recursos Arquitetura não Flexível 2-2-1.

Recurso	Utilizado	Total disponível	Utilização
Total logic elements	3 595	109 424	2%
Total combinational functions	3 594	109 424	2%
Dedicated logic registers	15	109 424	<1%
Total registers	15	–	–
Total pins	100	508	19%

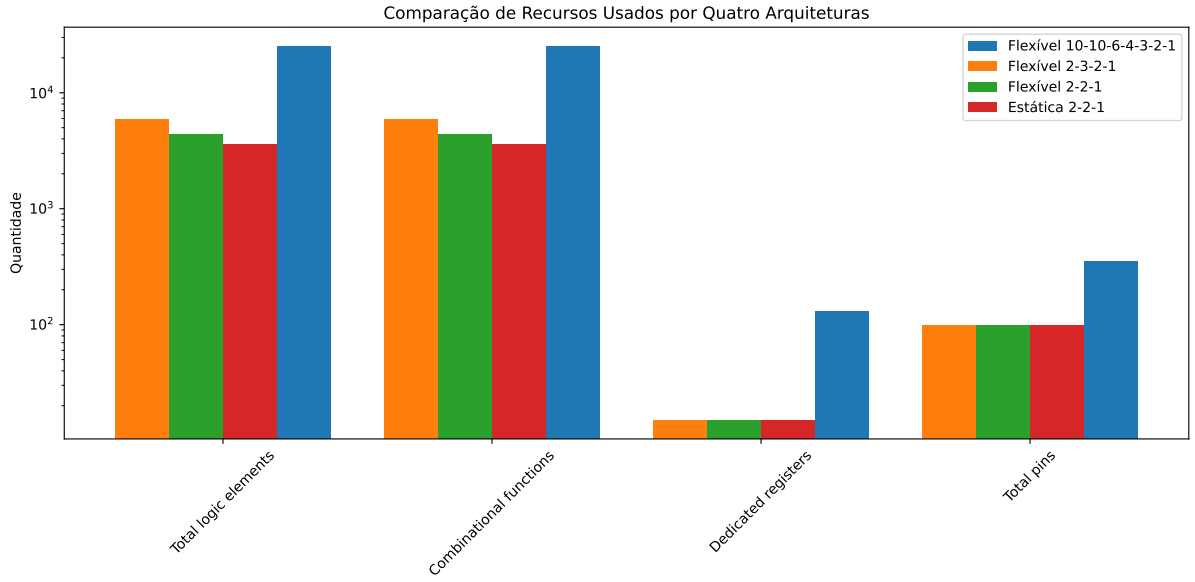

Figura 11. Gráfico Comparativo entre Arquiteturas

Comparando com a versão *parametrizável* (Tabela 4), observa-se um aumento não significativo, mantendo a porcentagem de recursos utilizados em 2%. Ao comparar como a rede de 26 neurônios (Tabela 6), houve um incremento de recursos, principalmente, em *logic elements* (de 2% para 23%) e em *registers* (de 15 para 130). Esse acréscimo é esperado e decorre, sobretudo, de:

- Lógica adicional de indexação e *slicing* para varrer camadas e neurônios com base em vetores de parâmetros (`NEURONS_PER_LAYER_C` e `WEIGHTS_C`);
- Multiplexação e `generate` condicionais para selecionar ativações e tamanhos de entrada por instância de neurônio;
- Controle sequencial (FSM) comum às instâncias, dimensionado para tamanhos genéricos ao invés de dimensões fixas.

Em contrapartida, o modelo parametrizável oferece maior reuso e escalabilidade, permitindo alterar topologia e pesos sem reescrita estrutural do código. Contudo, os resultados mostram que esse ganho de flexibilidade vem acompanhado de um aumento progressivo no custo em

hardware, especialmente em arquiteturas com maior número de neurônios. Assim, embora o dispositivo-alvo ainda apresente margem suficiente para operar as topologias avaliadas, o crescimento de área observado na rede de 26 neurônios indica que a expansão para configurações maiores exige cautela. Deve-se considerar antecipadamente o impacto sobre elementos lógicos, registradores e roteamento, uma vez que expansões adicionais — como aumento do número de neurônios, inclusão de memória dedicada a pesos ou estágios extras de *pipelining* — podem aproximar o projeto dos limites de capacidade do FPGA.

6. Conclusões

Este trabalho demonstrou a eficácia de uma metodologia parametrizável para a implementação de redes neurais em hardware. A proposta resolveu o desafio central da rigidez de designs estáticos (identificado na Seção 2), por meio de uma arquitetura VHDL genérica (Seção 4) que utiliza parâmetros `generic` e laços `for...generate` para a construção procedural da rede.

O sucesso da arquitetura foi validado pelos resultados da Seção 5. A **Validação Funcional** comprovou que a rede gerada (para o problema XOR 2-2-1) produziu saídas idênticas, bit-a-bit, ao modelo de referência em Python, garantindo a correção lógica do hardware. Mais importante, a **Validação da Síntese** demonstrou visualmente que o mesmo código-fonte VHDL, ao se alterar apenas o parâmetro `NEURONS_PER_LAYER`, gerou hardwares distintos, provando o sucesso da parametrização.

Entretanto, a flexibilidade proposta introduz *trade-offs* importantes que devem ser considerados. A análise empírica mostrou que, embora o acréscimo moderado de neurônios não impacte significativamente o consumo de recursos — como observado nas redes 2-2-1 (3 neurônios) e 2-3-2-1 (6 neurônios), ambas mantendo cerca de 2% de utilização dos elementos lógicos — o crescimento mais agressivo da arquitetura leva a um aumento substancial na demanda por hardware. No caso da rede 10-10-6-4-3-2-1, com 26 neurônios, o consumo atingiu 23% dos recursos lógicos no modelo flexível, enquanto o modelo base utilizado como referência permaneceu próximo de 2%. Esses resultados evidenciam que o *trade-off* entre flexibilidade e custo computacional torna-se mais pronunciado à medida que o número de neurônios cresce, uma vez que cada nova instância adiciona blocos de processamento e conexões adicionais. Para arquiteturas maiores, o congestionamento de roteamento passa a se tornar um gargalo dominante, dada a expansão combinatória das interconexões em topologias totalmente conectadas

A arquitetura mitiga parcialmente a explosão de área ao reutilizar multiplicadores através de Máquinas de Estados (FSM), mas essa decisão impõe uma penalidade de latência, tornando o processamento sequencial dentro de cada camada. Por fim, o sistema limita-se à etapa de inferência, dependendo de treinamento *offline*.

Apesar destes custos, conclui-se que o aumento de área é justificado pelo ganho crucial em flexibilidade, eliminando a manutenção manual de código. Visando a reprodutibilidade, todo o código-fonte e scripts de validação foram disponibilizados publicamente em um repositório digital[1]. O trabalho consolida, portanto, uma base sólida para a prototipagem rápida de aceleradores de redes neurais em hardware.

Referências

[Antunes and Podobas 2025] Antunes, P. and Podobas, A. (2025). FPGA-based neural network accelerators for space applications: A survey. *arXiv preprint arXiv:2504.16173*. Disponível

[1]Disponível em: `https://github.com/IsraelRAlencar/neuron_network_vhdl` e `https://redmine.lscad.facom.ufms.br`. Acesso em: 20 nov. 2025.

em: <https://arxiv.org/abs/2504.16173>. Acesso em: 2 fev. 2025. Levantamento sobre aceleradores de redes neurais em FPGA para aplicações espaciais.

[Campos et al. 2024] Campos, N., Edirisinghe, E., Chesnokov, S., and Larkin, D. (2024). Fast generation of custom floating-point spatial filters on FPGAs. *arXiv preprint arXiv:2409.05837*. Disponível em: <https://arxiv.org/abs/2409.05837>. Acesso em: 05 set. 2025. Propõe o uso de DSLs e ponto flutuante customizado para acelerar a prototipagem de hardware.

[Costa 2023] Costa, D. (2023). Fpga linear regression. GitHub Repository. Acessado em Outubro 20, 2025. Implementação VHDL de Regressão Linear em FPGA.

[de Assis Pedrobon Ferreira et al. sd] de Assis Pedrobon Ferreira, W., Grout, I., and da Silva, A. C. R. (s.d.). Implementação de um algoritmo de regressão linear em hardware utilizando operações aritméticas de ponto fixo. *Não especificado no documento (possivelmente revista ou conferência)*. Artigo sobre regressão linear em FPGA com VHDL e ponto-fixo.

[Fernández-Caro 2023] Fernández-Caro, A. G. (2023). Implementation of a neural network in fpga: Investigating the viability of lenet-5 network implementation using field-programmable gate arrays. Trabalho de conclusão de curso (engenharia eletrônica), Universitat Politècnica de Catalunya (UPC), Barcelona, Spain. Análise da implementação manual da rede LeNet-5 em FPGA e discussão sobre limitações de recursos de hardware estático.

[Gautam et al. 2025] Gautam, A., Date, P., Kulkarni, S., Patton, R., and Potok, T. (2025). NeuroCoreX: An open-source FPGA-based spiking neural network emulator with on-chip learning. *arXiv preprint arXiv:2506.14138*. Disponível em: <https://arxiv.org/abs/2506.14138>. Acesso em: 4 mar. 2025. Apresenta uma arquitetura flexível para SNNs com aprendizado STDP em hardware.

[Ghaffari and Savaria 2020] Ghaffari, A. and Savaria, Y. (2020). CNN2Gate: Toward designing a general framework for implementation of convolutional neural networks on FPGA. *arXiv preprint arXiv:2004.04641*. Disponível em: <https://arxiv.org/abs/2004.04641>. Acesso em: 05 abril. 2025. Proposta de framework automatizado para conversão de modelos de alto nível (como Keras/PyTorch) para FPGA.

[Giasemis et al. 2025] Giasemis, F. I., Lončar, V., Granado, B., and Gligorov, V. V. (2025). Comparative analysis of FPGA and GPU performance for machine learning-based track reconstruction at LHCb. *arXiv preprint arXiv:2502.02304*. Disponível em: <https://arxiv.org/abs/2502.02304>. Acesso em: 05 set. 2025. Compara a eficiência energética de FPGAs e GPUs na inferência de MLPs para física de altas energias.

[Liu et al. 2022] Liu, K., Börjeson, E., Häger, C., and Larsson-Edefors, P. (2022). FPGA implementation of multi-layer machine learning equalizer with on-chip training. *arXiv preprint arXiv:2212.03515*. Disponível em: <https://arxiv.org/abs/2212.03515>. Acesso em: 12 out. 2025. Demonstra a viabilidade de treinamento (backpropagation) completo em FPGA.

[Ortiz Arciniega et al. 2019] Ortiz Arciniega, J. L., Carrió, F., and Valero, A. (2019). FPGA implementation of a deep learning algorithm for real-time signal reconstruction in radiation detectors under high pile-up conditions. *arXiv preprint arXiv:1903.02439*. Disponível em: <https://arxiv.org/abs/1903.02439>. Acesso em: 9 set. 2025. Apresenta uma implementação de rede neural em VHDL para processamento de sinais em tempo real, substituindo DSPs tradicionais.

[Silva 2019] Silva, G. d. J. C. d. (2019). Implementação de redes neurais artificiais em hardware para inferência. Trabalho de conclusão de curso (engenharia de computação), Universidade

Federal de Santa Maria (UFSM), Santa Maria, RS. Proposta de implementação modular de redes neurais em VHDL focada em flexibilidade, validada com a função XOR.

[Slavescu and Bastone 2023] Slavescu, A. and Bastone, L. (2023). Hardware implemented neural network. GitHub Repository. Acessado em Outubro 20, 2025. Implementação VHDL de uma Rede Neural MLP com aprendizado on-chip.