PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA

# AMBIENTE PARA EXPLORAÇÃO DE CNNS EM NÍVEL RTL

**TÁRSIO ONOFRIO CARDOSO DA SILVA**

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do grau de Bacharel em Engenharia de Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Fernando Gehm Moraes

**Porto Alegre**
**2023**

**AGRADECIMENTOS**

Gostaria de deixar aqui minha gratidão às pessoas que contribuíram para o desenvolvimento deste TCC.

Agradeço à minha família, à minha amada esposa Adriana, às minhas preciosas filhas Isabela e Aurora. À minha mãe, Cecília e às minhas tias, Ângela e Vera que sempre cuidaram de mim. E à minha avó, Lúcia que sempre sonhou com esse momento.

Gostaria de agradecer ao meu orientador, Fernando Gehm Moraes, pelo apoio, atenção e inúmeros incentivos. Obrigado por toda a paciência, calma e conselhos.

Agradeço a Leonardo Juracy pelas explicações sobre sua tese e por todo o apoio.

Agradeço à PUCRS, a todos os meus professores e professoras, também agradeço a toda a equipe de laboratórios e secretaria que me acompanharam durante todos esses anos. Todos sempre foram muito atenciosos e pacientes.

Agradeço a Vinicius Souza pelo apoio nessa etapa final, auxiliando na síntese e implementação do projeto em FPGA.

Quero agradecer a todos os amigos e colegas que tive durante todos esses anos, incluindo aqueles que hoje estão mais distantes.

E agradeço a Deus, a ele toda honra, glória e louvor.

# AMBIENTE PARA EXPLORAÇÃO DE CNNS EM NÍVEL RTL

## RESUMO

Este trabalho de conclusão de curso (TCC) aborda o desenvolvimento de um acelerador de hardware para aprendizado de máquina, com foco em redes neurais convolucionais (CNNs). O problema central abordado é a necessidade de acelerar o processamento de CNNs em hardware dedicado, visando melhorar a eficiência e o desempenho dessas redes. O objetivo estratégico deste trabalho é desenvolver um ambiente de exploração para CNNs em nível RTL, permitindo a análise e otimização de desempenho. Para atender a este problema, foi utilizado como base o acelerador 2D WS sistólico desenvolvido no grupo de pesquisa. Esse acelerador, denominado CONVWS, foi projetado para realizar operações de convolução em CNNs de apenas 1 camada, utilizando uma arquitetura de memória específica, com módulos de memória para pesos e viés, bem como para os mapas de características. Assim para prover um ambiente de exploração para CNNs, o trabalho contou com três frentes de desenvolvimentismo. A primeira foi relacionada à arquitetura de memória, que separou pesos e bias dos mapas de características (IFMAP e OFMAMP), requerendo domínio de memórias embarcadas em dispositivos FPGAs e modificações no acelerador de referências para adequar o mesmo a esta nova arquitetura. A segunda frente de trabalho foi relacionado à interconexão de diversos aceleradores, de forma a implementar um CNN com diversas camadas. Finalmente a terceira frente de trabalho correspondeu ao projeto e integração das camadas *max polling* e *fully connected*, permitindo o desenvolvimento de redes completas. É importante destacar que o código VHDL do acelerador é parametrizável e integrado ao TensorFlow, permitindo a exploração de arquiteturas de CNNs. O trabalho foi validado por simulação RTL, e prototipado em dispositivos FPGAs. Resultados apresentam dados de desempenho e ocupação de área no FPGA.

**Palavras-Chave:** Aceleradores de hardware para CNN, CNN, FPGAs, arquiteturas de memória.

# ENVIRONMENT FOR EXPLORATION OF CNNS AT THE RTL LEVEL

## ABSTRACT

This Bachelor Thesis explores the development of a hardware accelerator for machine learning, focusing on convolutional neural networks (CNNs). The central issue addressed is the need to accelerate the processing of CNNs on dedicated hardware, aiming to enhance the efficiency and performance of these networks. The strategic objective of this work is to develop an exploration environment for CNNs at the RTL level, allowing performance analysis and optimization. To address this issue, the 2D WS systolic accelerator developed in the research group was used as the reference design. This accelerator, named CONVWS, was designed to perform convolution operations in CNNs of only one layer, using a specific memory architecture with memory modules for weights, bias, and feature maps. This work had three development fronts to provide an environment for CNNs architecture exploration. The first was related to memory architecture, which separated weights and bias from feature maps (IFMAP and OFMAMP), requiring knowledge of embedded memories in FPGA devices and modifications in the reference accelerator to adapt it to this new architecture. The second front was related to the interconnection of several accelerators to implement a CNN with multiple layers. Finally, the third front corresponded to the design and integration of the max pooling and fully connected layers, enabling the development of complete networks. It is important to highlight that the VHDL code of the accelerator is parameterizable and integrated with TensorFlow, allowing the exploration of CNN architectures. The work was validated through RTL simulation and prototyped on FPGA devices. Results present performance data and FPGA area occupancy.

**Keywords:** Hardware accelerators for CNN, CNN, FPGAs, memory architectures.

# LISTA DE FIGURAS


Figura 1.1 – Acelerador 2D WS a esquerda de [4] em azul e o mesmo acelerador com alterações deste trabalho em verde. . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
Figura 1.2 – Desenvolvimento deste trabalho em verde em relação ao acelerador 2D WS de [4] em azul. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
Figura 2.1 – Ilustração do processo de convolução [4]. . . . . . . . . . . . . . . . . . . . . . . 14
Figura 3.1 – Exemplo de código para o *TensorFlow*. . . . . . . . . . . . . . . . . . . . . . . . . 22
Figura 3.2 – Acelerador 2D WS, com interconexão às interfaces de memória [4]. . 23
Figura 3.3 – Máquina de estados de controle da convolução WS . . . . . . . . . . . . . . 25
Figura 3.4 – FSM de busca de dados para o acelerador WS. . . . . . . . . . . . . . . . . . 26
Figura 3.5 – Arquitetura de memória da arquitetura de referência. . . . . . . . . . . . . . 27
Figura 3.6 – Partição da memória de entrada do acelerador de referência. . . . . . . 28
Figura 3.7 – Multiplexadores para Controle de Acesso à Memória . . . . . . . . . . . . . 29
Figura 3.8 – *Test bench* da convolução do trabalho de referência . . . . . . . . . . . . . 30
Figura 4.1 – Partição de endereços para a memória de pesos e bias (MEMWGHT). 33
Figura 4.2 – Partição de endereços para a memória de características (MEMF-MAP). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
Figura 4.3 – Lógica para acesso às memórias do CONVWS. . . . . . . . . . . . . . . . . . 34
Figura 4.4 – Arquitetura do *test bench* com duas memórias. . . . . . . . . . . . . . . . . 35
Figura 4.5 – Módulo CORE1. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
Figura 4.6 – Detalhe de $M_f$ e $M_w$, multiplexadores das memórias MEMFMAP de entrada e MEMWGHT no CORE1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
Figura 4.7 – Detalhe de $M_{of}$ multiplexador da MEMFMAP de saída no *Core* . . . . . 38
Figura 4.8 – Fluxograma da geração da descrição RTL. . . . . . . . . . . . . . . . . . . . . . 42
Figura 4.9 – Core Serial. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
Figura 4.10 – CNN Simple11 (CNNSM). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
Figura 4.11 – Macro BRAM_SINGLE_MACRO. . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
Figura 4.12 – Arquitetura de encapsulamento das BRAMs - módulo `bram_single`. . 53
Figura 4.13 – Arquitetura de acesso ao encapsulamento das BRAMs. . . . . . . . . . . 55
Figura 4.14 – Particionamento do barramento `address` para seleção de módulo `bram_single` e geração do endereço interno da BRAM. . . . . . . . . . . . . . . . . 56
Figura 4.15 – Forma de onda da BRAM em modo `WRITE_FIRST` [15]. . . . . . . . . . . . . 56
Figura 4.16 – Forma de onda da BRAM em modo `READ_FIRST` [15]. . . . . . . . . . . . . 56

Figura 4.17 – Comportamento do módulo `memory` sem registrar sinais (*Select* e *data_output*) e registrando ($Select_{FF}$ e $data_output_{FF}$), usando como base a saída de duas `bram_single`, $DO_1$ e $DO_2$. . . . . . . . . . . . . . . . . . . . . . . . . . . 57
Figura 4.18 – Listas mostrando a transformação dos dados passando pelas três operações: complemento de dois, hexadecimal e ordenação. . . . . . . . . . . . . 59
Figura 4.19 – Geração dos valores utilizados para inicialização das BRAMs. . . . . . 60
Figura 4.20 – Valores utilizados para inicialização das BRAMs, com largura de 255 bits (64 valores hexadecimais). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
Figura 4.21 – Máquina de estados da *maxpooling2d*. . . . . . . . . . . . . . . . . . . . . . . . . 64
Figura 4.22 – Fully connected. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
Figura 4.23 – Máquina de estados da arquitetura *fully_connected* . . . . . . . . . . . . . . 66
Figura 4.24 – Máquina de estados do ACCEL . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
Figura 5.1 – Exemplo de uma CNN para executar o *dataset* CIFAR [4]. . . . . . . . . . 69
Figura 5.2 – Diagrama de conjuntos com a média de uso de LUTS e FFs por módulo. Nome de cada módulo em negrito. . . . . . . . . . . . . . . . . . . . . . . . . . 74
Figura 5.3 – Gráfico da relação entre RAMB36 e LUTs para cada instância do módulo de Controle e Acesso as BRAM. . . . . . . . . . . . . . . . . . . . . . . . . . . 78
Figura 5.4 – Planta baixa no FPGA Artix 7 para os modelos de CNN explorados. . 79

# LISTA DE TABELAS

Tabela 4.1 – Tabela de conexões entre CORESR na CNNSM . . . . . . . . . . . . . . . . 45
Tabela 4.2 – Configurações para BRAM 18Kb. . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
Tabela 4.3 – Configurações para BRAM 36Kb. . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
Tabela 4.4 – Ordem de leitura dos endereços da *maxpooling2d* para uma imagem com tamanho ($image_{size}$)de 6 × 6 e pooling com tamanho ($kernel_{size}$) igual a 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
Tabela 5.1 – Quantidade de BRAMs e parâmetros gerados para as DNN. . . . . . . . 71
Tabela 5.2 – Benchmark da simulação de cada CNN no Modelsim em milisegundo (ms) pressupondo ACC com clock de 100 MHz em relação a GPU (Nvidia GeForce GTX 980 Ti e Intel® Core™ i5-2320 CPU @ 3.00GHz × 4) e uma CPU (Intel(R) Core(TM) i9-7940X CPU @ 3.10GHz). O resultado do $batch_{10}$ o tempo médio das 10 amostras. . . . . . . . . . . . . . . . . . . . . . . . . . . 71
Tabela 5.3 – Benchmark da simulação no Modelsim em milisegundo (ms) pressupondo clock de 100 MHz para ACC na execução das operações de convolução. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
Tabela 5.4 – Utilização de recursos da FPGA para cada CNN. . . . . . . . . . . . . . . . 75
Tabela 5.5 – Utilização de recursos da FPGA pelo módulo CNNSR. . . . . . . . . . . . 75
Tabela 5.6 – Utilização de recursos da FPGA por instância de CORESR em cada CNN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
Tabela 5.7 – Utilização de recursos da FPGA por instância de CONVWS por CORESR e CNN. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
Tabela 5.8 – Utilização de recursos da FPGA pela MEMFMAP do módulo CNNSR para cada modelo de CNN. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
Tabela 5.9 – Utilização de recursos da FPGA por instância de MEMFMAP de entrada por CORESR e CNN. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
Tabela 5.10 – Utilização de recursos da FPGA por MEMWGHT por CORESR e CNN. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
Tabela 5.11 – Comparação entre CNNs implementadas em FPGAs [11]. . . . . . . . . 80

## SUMÁRIO


**1 INTRODUÇÃO . . . 10**

1.1 MOTIVAÇÃO . . . 10

1.2 OBJETIVOS . . . 11

1.3 ORGANIZAÇÃO DO DOCUMENTO . . . 12

**2 CONCEITOS BÁSICOS RELACIONADOS A CNNS E ACELERADORES DE HARDWARE . . . 13**

2.1 DEEP LEARNING . . . 13

2.2 ACELERADORES DE HARDWARE . . . 14

2.2.1 ASICS . . . 15

2.2.2 FPGAS . . . 19

**3 ARQUITETURA DE REFERÊNCIA . . . 21**

3.1 MODELO TENSORFLOW . . . 21

3.2 ARQUITETURA DO ACELERADOR . . . 22

3.3 *DATAFLOW WEIGHT STATIONARY – WS* . . . 24

3.4 MÓDULOS DO ACELERADOR MODIFICADOS NO CONTEXTO DO TCC . . . 26

3.4.1 ESTRUTURA DE MEMÓRIA DO ACELERADOR DE REFERÊNCIA . . . 26

3.4.2 LÓGICA PARA CONTROLE DE ACESSO À MEMÓRIA . . . 28

3.4.3 *TEST BENCH* . . . 30

**4 PROJETO DO ACELERADOR DE HARDWARE PARA APRENDIZADO DE MÁQUINA . . . 31**

4.1 ALTERAÇÕES NA ARQUITETURA DE REFERÊNCIA . . . 31

4.1.1 MEMÓRIAS MEMWGHT E MEMFMAP . . . 32

4.1.2 LÓGICA PARA ACESSO ÀS MEMÓRIAS . . . 33

4.1.3 *TEST BENCH* . . . 34

4.2 MÓDULO *CORE* . . . 34

4.2.1 CORE 1-LAYER . . . 35

4.2.2 *TEST BENCH* . . . 38

4.3 EXECUÇÃO DE MÚLTIPLAS CAMADAS . . . 39

4.3.1 SUPORTE PARA CONFIGURAÇÃO VARIÁVEL . . . 39

4.3.2 ALGORITMO DE GERAÇÃO DOS DADOS . . . 40

4.3.3 *TEST BENCH* . . . . . . . . . . 42
4.4 CNN SIMPLE . . . . . . . . . . 43
4.4.1 MEMÓRIA . . . . . . . . . . 43
4.4.2 CORE SERIAL . . . . . . . . . . 44
4.4.3 CNN . . . . . . . . . . 45
4.5 MEMÓRIAS BRAMS EM DISPOSITIVOS FPGAS XILINX . . . . . . . . . . 46
4.5.1 INSTANCIAÇÃO E IMPLEMENTAÇÃO DE BRAMS . . . . . . . . . . 47
4.5.2 ARQUITETURA DAS MEMÓRIAS BRAMS . . . . . . . . . . 51
4.5.3 ALGORITMO DE GERAÇÃO DA INICIALIZAÇÃO DAS BRAMS . . . . . . . . . . 58
4.6 CAMADAS MAX POOLING E FULLY CONNECTED . . . . . . . . . . 62
4.6.1 *MAX POOLING 2D* — MP2D . . . . . . . . . . 63
4.6.2 *FULLY CONNECTED* — FC . . . . . . . . . . 65
4.7 ACELERADOR EM FPGA . . . . . . . . . . 67

**5 RESULTADOS . . . . . . . . . . 69**
5.1 CONJUNTO DE DADOS . . . . . . . . . . 69
5.2 MODELOS . . . . . . . . . . 69
5.3 BENCHMARK . . . . . . . . . . 71
5.4 USO DE RECURSOS DA FPGA . . . . . . . . . . 73
5.5 COMPARAÇÃO COM TRABALHOS RELACIONADOS . . . . . . . . . . 79

**6 CONCLUSÃO E TRABALHOS FUTUROS . . . . . . . . . . 81**

**REFERÊNCIAS . . . . . . . . . . 83**

# 1. INTRODUÇÃO

As redes neurais convolucionais (CNNs) têm se mostrado extremamente eficazes em diversas aplicações de aprendizado de máquina, abrindo caminho para avanços significativos em áreas como reconhecimento de imagens, processamento de linguagem natural e análise de dados. Essas redes possuem uma arquitetura especializada que permite a extração de características relevantes dos dados de entrada, tornando-as ideais para lidar com problemas complexos e de grande escala.

Com o advento da Internet das Coisas (IoT) e o desenvolvimento de carros autônomos, a importância das CNNs se torna ainda mais evidente. Em IoT, por exemplo, as CNNs podem ser utilizadas para o reconhecimento de objetos em tempo real, permitindo que dispositivos inteligentes tomem decisões com base nas informações capturadas por sensores. Já nos carros autônomos, as CNNs são essenciais para a detecção de pedestres, veículos e obstáculos, garantindo a segurança e a eficiência do sistema.

No entanto, é importante destacar que o desempenho das CNNs está diretamente relacionado ao poder de processamento disponível. Com o aumento da complexidade e profundidade destas redes, a exigência por recursos computacionais escala de maneira significativa. Neste ponto, o papel do hardware dedicado para aceleração de CNNs torna-se preponderante para garantir um desempenho otimizado na fase de inferência [6].

O hardware específico para CNNs é projetado com o objetivo de acelerar as operações matemáticas intensivas executadas por estas redes, ao mesmo tempo, em que otimiza o acesso à memória externa do acelerador. Estes aceleradores têm a capacidade de processar cálculos em paralelo, capitalizando a natureza altamente paralelizável das CNNs. Eles podem ainda ser otimizados para diminuir a latência e o consumo energético, potencializando a eficiência geral do processamento e os tornando fundamentais para viabilizar aplicações avançadas de aprendizado de máquina.

## 1.1 Motivação

Diante da importância das CNNs e da necessidade de hardware dedicado para alcançar um desempenho satisfatório, surge a motivação para o desenvolvimento de aceleradores de hardware especializados para essas redes. Esses aceleradores têm o potencial de impulsionar o avanço das aplicações de aprendizado de máquina, permitindo o processamento rápido e eficiente de grandes volumes de dados.

Além disso, o uso de hardware dedicado para CNNs pode trazer benefícios significativos em termos de desempenho e eficiência energética. Ao utilizar recursos computacio-

nais otimizados para as operações específicas das CNNs, é possível obter resultados mais precisos em um tempo menor, além de reduzir o consumo de energia.

## 1.2 Objetivos

O objetivo estratégico deste trabalho[1] é desenvolver um ambiente de exploração para redes neurais convolucionais (CNNs) em nível RTL, permitindo a análise e otimização de desempenho.

Para alcançar esse objetivo estratégico, foram estabelecidos os seguintes objetivos específicos:

1. Dominar a arquitetura de aceleração de referência, denominado de CONVWS, proposta em [4]. Este acelerador executa a convolução sobre uma camada de uma CNN, utilizado uma matriz 2D de dimensão 3x3, com memórias comportamentais (descritas na forma de um vetor).
2. Dominar a utilização de blocos de memória presentes nos dispositivos FPGAs, denominados de BRAM.
3. Alterar a arquitetura de memória do CONVWS visando a utilização de BRAMs. A Figura 1.1 apresenta à esquerda os principais blocos do CONVWS. A figura da esquerda mostra que neste TCC alteramos o bloco de controle da memória (“Mem contr”), adicionando um novo de conjunto de registradores (“REG”) e um novo controlador de memória visando o uso de BRAMs.

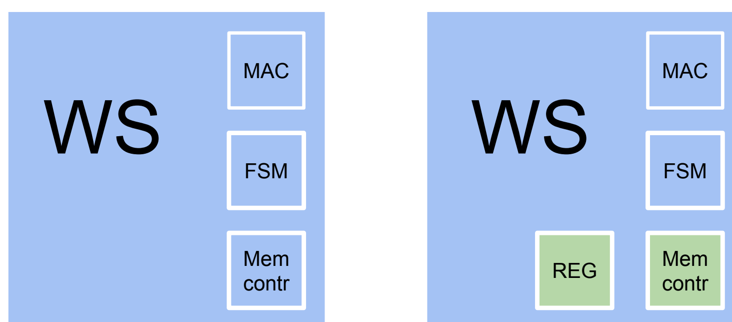



Figura 1.1 – Acelerador 2D WS a esquerda de [4] em azul e o mesmo acelerador com alterações deste trabalho em verde.

4. Desenvolver uma CNN completa, através da integração em série de diversos módulos CONVWS. A Figura 1.2 apresenta esta rede completa, composto por várias camadas (“CORE”), interconectadas em série.

[1]Todas as contribuições deste trabalho são encontradas no repositório do GitHub: https://github.com/tarsioonofrio/acc_dse_env/tree/484fa709cfe5597e464643d2bc0e20e2b14412e6.

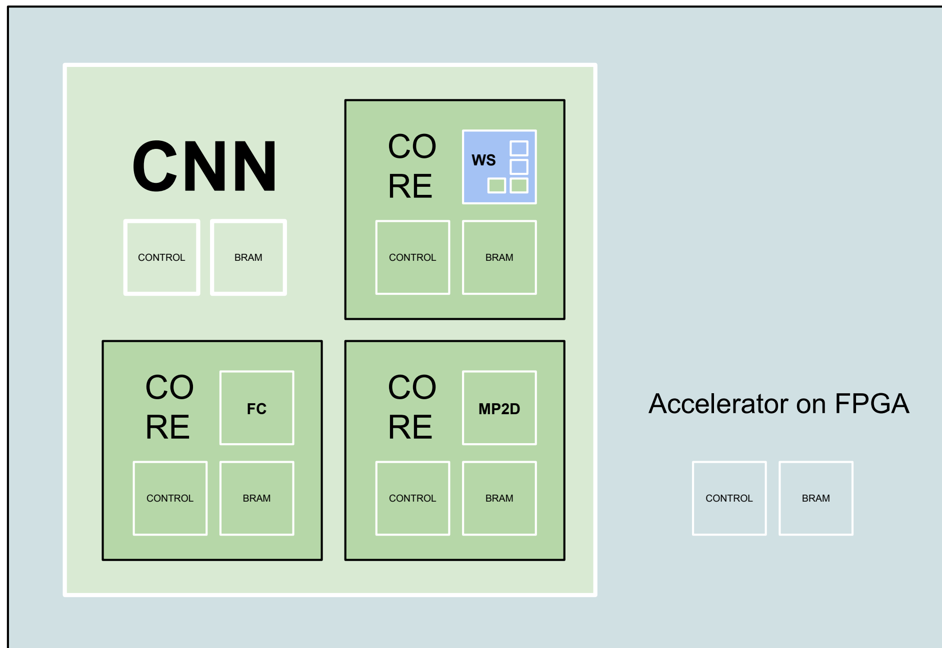



Figura 1.2 – Desenvolvimento deste trabalho em verde em relação ao acelerador 2D WS de [4] em azul.

5. Desenvolver uma ferramenta que permita a inicialização das BRAMs conforme os dados extraídos do treinamento da CNN no *framework* TensorFlow.
6. Avaliar por simulação uma rede composta por diversas camadas, validando o hardware por simulação, utilizando por referência os valores gerados pelo *framework* TensorFlow.
7. Desenvolver as camadas *max polling* e *fully connected*, não desenvolvidas no trabalho de referência.
8. Validar a CNN desenvolvida em dispositivos FPGAs

## 1.3 Organização do Documento

Este TCC é organizado como segue:

- O Capítulo 2 apresenta os conceitos básicos necessários à compreensão deste trabalho.
- O Capítulo 3 detalha a arquitetura de aceleração de referência, proposta em [4]. *Este Capítulo atende ao primeiro objetivo específico*.
- O Capítulo 4 corresponde à principal à contribuição deste TCC, o qual é o projeto de um acelerador de hardware para aprendizado de máquina, tendo por base a aceleração de referência. *Este Capítulo atende aos objetivos específicos 2 a 8*.
- O Capítulo 5 apresenta resultados de simulação e prototipação.
- O Capítulo 6 conclui este trabalho e aponta direções para trabalhos futuros.

# 2. CONCEITOS BÁSICOS RELACIONADOS A CNNS E ACELERADORES DE HARDWARE

## 2.1 Deep Learning

As redes neurais profundas *feedforward* (DNN), são modelos de aprendizado profundo. Essas redes têm o propósito de aproximar uma função, mapeando uma entrada em uma categoria. Elas são chamadas de *feedforward* devido à forma unidirecional como a informação flui, sem realimentações. Essas redes, quando estendidas para incluir conexões de feedback, tornam-se redes neurais recorrentes. As redes *feedforward* têm relevância crucial no campo do aprendizado de máquina, sendo a base de diversas aplicações comerciais, incluindo redes convolucionais para reconhecimento de objetos.

Redes neurais convolucionais (CNNs) são uma forma especializada de rede neural projetada para processar dados com uma topologia conhecida e semelhante a uma grade. Exemplos incluem dados de séries temporais (grade 1D) e dados de imagem (grade 2D). As CNNs incluem três tipos principais de camadas: convolução, *pooling*, e *fully connected* [2].

A convolução discreta é representada como uma multiplicação matricial, com certas restrições impostas a alguns elementos. A matriz de convolução geralmente é esparsa, devido ao tamanho do *kernel* ser geralmente menor do que a imagem de entrada.

A equação 2.1 apresenta formalmente o método para alcançar um resultado de um mapa de características de saída $O$ (OFMAP). A operação de convolução utiliza o mapa de características de entrada $i$ IFMAP e uma matriz com os pesos $w$. A partir daí, cada janela de pesos $w$ é processada em conjunto (e deslocada ao longo) com o canal $c$ de entrada correspondente, criando um novo conjunto de mapas de características $o$. Posteriormente, um vetor de vieses (*bias*) $b$ é incorporado ao mapa de características. Nesse contexto: $w$ e $h$ representam a posição horizontal e a posição vertical, respectivamente; C indica o número total de canais de entrada e de filtro, W e H se referem ao tamanho do filtro e $s$ é o *stride* [4].

$$O_{cwh} = b_c + \sum_{c=0}^{C-1} \sum_{w=0}^{W-1} \sum_{h=0}^{H-1} (i_{s_w+w, s_h+h} * w_{cwh}) \tag{2.1}$$

A Figura 2.1 ilustra o processo de convolução. Nesta figura temos os mapas de características de entrada $i$ IFMAP e saída $O$ IOFMAP com três canais. Os filtros possuem dimensão 3x3.

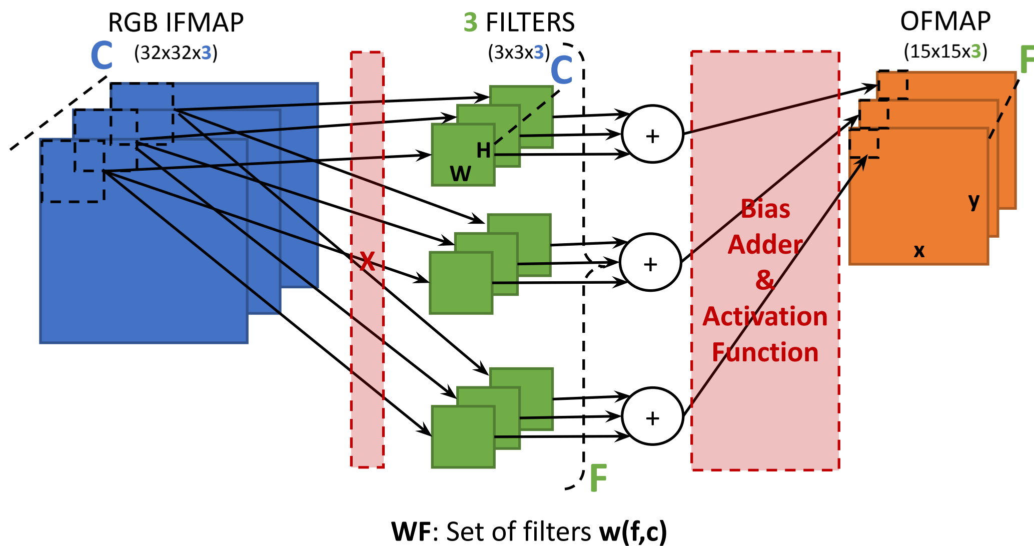



Figura 2.1 – Ilustração do processo de convolução [4].

A camada *fully connected* ou *linear* é a operação do produto vetorial entre duas matrizes, equação 2.2, onde $i$ são as *features* de entrada (IFMAP), $w$ são os pesos, $b$ é o viés, e $O$ são as *features* de saída.

$$O_b = iw^T + b = \sum_{j=i}^{n-1} b + i_j w_j = b_1 + i_1 w_1 + b_2 i_2 w_2 + \cdots + b_n i_n w_n \tag{2.2}$$

O *pooling* contribui para que a representação da rede se torne praticamente invariante a pequenas alterações na entrada. Ou seja, pequenas variações na entrada não modificam significativamente a maioria das saídas agrupadas [2]. A operação de *max pooling* seleciona os maiores valores em dada janela, sendo representada pela Equação 2.3.

$$O_{hw} = \max_{h,w} \left( i_{sh \to sh+h, sw \to sw+w} \right) \tag{2.3}$$

Onde $O_{hw}$ é OFMAP do *pooling*, $i_{sh \to sh+h, sw \to sw+w}$ é o intervalo da posição do IFMAP, $s$ é o *stride*, que determina o deslocamento da janela de *pooling* pela matriz, $h$ e $w$ são as variáveis que percorrem a extensão da janela de *pooling*, e o operador max seleciona o maior valor dentro dessa janela.

## 2.2 Aceleradores de Hardware

Essa seção se dedica a revisão de conceitos básicos em aceleradores de Hardware para CNNs em ASICs [9] e FPGAs [11].

### 2.2.1 ASICs

Nessa seção, conforme [9], revisamos conceitos básicos acerca da implementação de aceleradores CNN baseados em ASIC, focando na otimização do tempo de computação, acesso à memória e redução do consumo de memória, sem considerar projetos em FPGAs.

#### Referências no projeto de aceleradores de CNN

Cada acelerador CNN pode ser abstraído na forma de uma arquitetura de referência, caracterizada por múltiplos elementos de processamento (PEs), onde partes do cálculo são mapeadas para cada PE. Esses PEs transferem dados e resultados entre si, e de uma camada da CNN para a próxima. Cada PE é composto por uma ALU, uma unidade de controle e um arquivo de registro, com buffers globais para ativações de entrada/saída e pesos. No entanto, como o armazenamento integrado geralmente não é suficiente, uma memória externa é necessária para manter todos os dados. Além do buffer global, cada PE armazena os dados em registradores locais. Para transmitir dados diretamente de um PE para outro existe uma rede inter-PE.

As arquiteturas podem ser classificadas com base na disposição dos PEs e no tipo de fluxo de dados explorado. A maioria possui uma disposição 1D ou 2D de PEs, com uma rede inter-PE de conexões ponto a ponto ou todos conectados a um buffer global [9].

#### Redução do tempo de computação

Com o aumento da complexidade algorítmica das CNNs, o tempo de cálculo aumentou drasticamente. As formas intuitivas de reduzir o tempo de cálculo incluem o uso de recursos paralelos, redução do número de cálculos, redução do tempo de execução de cada cálculo envolvido em uma operação de convolução ou o uso de ciclos ociosos em um cálculo para algum trabalho útil.

A operação de uma camada convolucional é inerentemente paralela. No entanto, como uma CNN é uma rede *feedfoward*, cada camada é dependente de dados de sua camada anterior, impossibilitando executá-las em paralelo. As diferentes camadas de uma CNN podem ser encadeadas sujeitas a restrições de memória.

Existem três tipos de paralelismo: inter-output, inter-kernel e intra-kernel. O paralelismo intra-kernel realiza as operações de multiplicação e adição de uma operação de convolução em paralelo. O paralelismo inter-kernel produz as ativações de um mapa de recursos de saída em paralelo. O paralelismo inter-output produz as ativações de vários mapas de recursos de saída em paralelo.

A ideia chave na redução de cálculos baseada em padrões de convolução é uma troca entre tempo e espaço. Reutilizando resultados parciais dos filtros, reduzindo a quantidade de cálculo ao custo de aumentar o espaço de armazenamento. No entanto, esta concepção aumenta o consumo de energia através de buscas de buffer adicionais, mas também economiza energia ao reduzir o número de cálculos.

Outra forma de reduzir o número de cálculos é remover aqueles cujo resultado é zero (ou abaixo de um certo limite). Essa abordagem economiza energia e tempo. Economiza energia porque há a opção de desligar rapidamente PEs quando um cálculo não precisa ser realizado. Economiza tempo quando um PE tem um grande número de multiplicações para fazer, e os PEs não precisam completar suas tarefas simultaneamente. Há um benefício de desempenho porque a multiplicação é muito mais lenta do que a verificação de bits nas máscaras.

Na redução de cálculos com base em previsão a ideia central é identificar se uma ativação de uma camada anterior de CNN contribui significativamente para a camada seguinte. Uma das propostas realizar os cálculos dos bits de ordem mais alta e mais baixa separadamente, usando os primeiros para prever as ativações que seriam ineficazes nas camadas ReLU e Pooling. Isso economiza energia, mas não reduz o tempo de cálculo. Uma maneira de reduzir o tempo de cálculo seria aumentar a flexibilidade do acesso aos dados, mas isso aumentaria a pressão no sistema de memória. Uma alternativa seria classificara e reorganizar os pesos. Embora essas técnicas economizem energia e operações, elas exigem mais espaço de memória e podem aumentar a latência.

A ideia central é que a remoção de cálculos ineficazes pode deixar as Unidades de Processamento (PEs) ociosas. Uma alternativa é usar essas PEs para trabalhos úteis, reduzindo o tempo de cálculo. Uma alternativa é explorar eficientemente os ciclos de cálculo com pesos de filtro zero, realizando cálculos com pesos não zero que estavam programados para um tempo futuro. Há também um algoritmo adaptativo que mapeia dinamicamente as dimensões para a matriz PE de forma a maximizar a utilização das PEs [9].

*Dataflow*

A reutilização temporal reaproveita dados já buscados da memória off-chip os armazenando em buffers on-chip. Isso reduz o número de acessos à memória off-chip, com reduções demonstradas de até 500 vezes. A paralelização de diferentes iteradores de loop em PEs paralelas permite a exploração de diferentes tipos de paralelismo, definindo o padrão de acesso aos dados.

São descritos quatro *data flow*. *Weight stationary* (WS) explora a reutilização de pesos, enquanto o *Output Stationary* (OS) reutiliza as saídas parciais. *No Local Reuse* (NLR) não mantém dados, peso, entrada e saída estacionários em um arquivo de registro local. *input stationary* (IS), mantém as ativações de entrada estacionárias no nível de regis-

tradores. Finalmente, a arquitetura de reutilização “all-in-on chamada *Row stationar* (RS) reutiliza todos os tipos de dados (pesos, ativações e somas parciais) no arquivo de registro. Apesar da reutilização de dados diminuir substancialmente o número de acessos à DRAM e, portanto, o consumo de energia, essa técnica depende fortemente da reutilização de buffers on-chip, o que aumenta a área para estruturas de armazenamento on-chip.

A reutilização espacial dos dados é uma técnica que aproveita a localidade dados, roteando dados entre elementos de processamento para evitar acessos custosos à memória. A eficácia do roteamento depende de um equilíbrio entre largura de banda e latência, e é afetada pelo congestionamento na rede, que é influenciado pelo número de acessos que os elementos de processamento fazem para buscar pesos de filtros e ativações de entrada da memória.

Dados ineficazes são valores próximos a zero que não afetam os cálculos e, portanto, podem ser descartados, economizando tempo e energia. Existem várias técnicas para lidar com esses dados. Uma delas é gerar índices para dados efetivos, evitando a necessidade de carregar dados ineficazes da memória. Outra abordagem é usar técnicas de codificação para calcular bit de vetores para ativações e pesos, permitindo a detecção de ineficácia em tempo real e armazenamento de dados de forma compactada [9].

Redução do consumo de memória

O aumento no tamanho dos modelos de Redes Neurais Convolucionais exige mais memória, dificultando o armazenamento de todos os dados nas caches on-chip e prejudicando o desempenho. Para minimizar essa demanda, são sugeridas estratégias como a redução da precisão de entradas e pesos, a não armazenagem de dados intermediários e ineficazes, visando a diminuição do armazenamento de dados on-chip.

Várias técnicas comprimem dados de entrada e pesos de filtro para armazenamento eficiente. Usando um método de particionamento, os pesos são decompostos em uma média e diferenças, economizando espaço. Embora seja necessário mais tempo para descomprimir os dados, o espaço de armazenamento é geralmente reduzido. Além disso, os dados intermediários são armazenados substituindo padrões frequentes por identificadores menores, melhorando a latência e diminuindo os requisitos de armazenamento.

Técnicas recentes propõem converter a codificação de valores para reduzir o número de bits necessários, possivelmente à custa da precisão. Isso resulta em melhor desempenho computacional e menor espaço de memória, mas pode comprometer a precisão. Para redes neurais convolucionais, essa abordagem pode ser usada efetivamente para comprimir partes do modelo minimizando a redução da precisão. Inicialmente, era usado uma precisão fixa para toda a rede, mas pesquisas posteriores revelaram as vantagens de desempenho e armazenamento da redução da precisão, começando a usar apenas a precisão mínima necessária por camada [9].

Projetos industriais de aceleradores de CNN

Tendências recentes na indústria favorecem o uso de paradigmas SIMD (Single Instruction, Multiple Data), MIMD (Multiple Instruction, Multiple Data) e VLIW (Very Long Instruction Word) na construção de aceleradores de CNN. Estes projetos preferem memória on-chip por conta de seu menor consumo de energia e latência. Na arquitetura SIMD, as unidades de processamento são organizadas em uma matriz que é alimentada por um buffer on-chip global.

Alguns projetos recentes introduziram um nível adicional na hierarquia de memória on-chip, resultando em reduções significativas no consumo de energia. Outros implementaram um processador VLIW seguindo o paradigma SIMD, aproveitando a arquitetura de comunicação sistólica para economizar energia e largura de banda.

Há também projetos industriais recentes que adotam o paradigma MIMD, onde todas as unidades de processamento são processadores completos com sua própria memória local para armazenamento [9].

Discussão

A precisão de um acelerador de CNN é influenciada por otimizações que reorganizam cálculos efetivos sem alterar o resultado. Alterações como mudança dos operandos, redução da precisão dos operandos, truncamento dos operandos, entre outros, podem impactar a precisão. Perdas de precisão são geralmente inferiores a 3% e mais pronunciadas se pequenos valores de peso são removidos.

Observou-se que o fluxo de dados RS tem menor consumo enérgico por MAC devido ao alto grau de reutilização de dados, enquanto o fluxo NLR tem maior consumo por MAC, devido ao alto número de acessos à DRAM. A eficiência energética aumenta à medida que se aumenta o tamanho da matriz PE e do buffer global, devido à diminuição de acessos redundantes à DRAM [9].

Desafios

O projeto de ASIC para aceleradores de CNN apresenta desafios devido à falta de flexibilidade após a fabricação. Determinadas decisões de projeto, como o tamanho do *array* PE e o grau de funcionalidade em um PE, precisam ser feitas com cuidado. Além disso, o rede de interconexão entre os PEs é altamente dependente do tipo de fluxo de dados suportado pelo array PE, afetando assim a área, energia e largura de banda do chip. Questões de memória, como a natureza dos buffers on-chip, seus tamanhos e bit-widths, também são preocupações chave, exigindo decisões de projeto precisas para evitar o desperdício de largura de banda e maximizar a exploração de localidade temporal. Por fim, um grau de reconfigurabilidade dinâmica pode ser necessário para lidar com uma grande vari-

edade de CNNs ou ANNs, apesar de isso poder aumentar a área e o consumo de energia devido à lógica e *lanes* extra [9].

### 2.2.2 FPGAs

Esta seção, conforme [11], apresenta uma visão geral da estrutura básica *Field Programmable Gate Arrays* (FPGAs) e como os métodos de aprendizado profundo podem se beneficiar das capacidades dos FPGAs. Por fim, destacam-se os desafios da implementação de redes de aprendizado profundo em FPGAs.

#### Desafios

A implementação de redes de aprendizado profundo, especialmente CNNs, em FPGAs apresenta desafios, incluindo a necessidade de uma quantidade significativa de armazenamento, largura de banda de memória externa e recursos computacionais na ordem de bilhões de operações por segundo. Modelos complexos ampliam essa questão e é esperado que futuros modelos de CNN se tornem mais complexos à medida que a quantidade de dados de treinamento continua a crescer e os problemas a serem resolvidos se tornam mais complexos.

Além disso, diferentes camadas nas CNNs têm diferentes características que resultam em diferentes requisitos de paralelismo e acesso à memória. Dessa forma, o acelerador CNN desenvolvido precisa ser projetado cuidadosamente para atender aos diferentes requisitos das camadas e precisa ser flexível para maximizar o desempenho para cada camada da CNN.

Conforme a tecnologia avança, os FPGAs continuam a crescer em tamanho e capacidades, tornando crucial ter mecanismos para atender aos requisitos de implementações eficientes de redes de aprendizado profundo. A superação das limitações dos recursos de hardware requer o reaproveitamento de recursos computacionais e o armazenamento de resultados parciais nas memórias internas. Adicionalmente, a redução do número de acessos à memória externa pode ser alcançada utilizando memória on-chip e explorando a reutilização de dados. Além disso, o grande número de pesos na camada totalmente conectada pode ser reduzido com um pequeno impacto na precisão [11].

#### Status Atual

Aceleração de redes neurais convolucionais (CNNs) implementadas em FPGAs envolve a otimização da operação de convolução (CONV), que consome cerca de 90% do tempo de cálculo. Tais otimizações são realizadas através da utilização de operações

paralelas de multiplicação-acumulação, projeto de padrões de acesso a dados para minimizar os requisitos de largura de banda da memória, maximizando a reutilização de dados e otimizando a quantidade de bits usados para representar mapas de características e pesos.

Para otimizar a paralelização das operações de convolução, diversas abordagens foram tentadas, incluindo a análise da carga de trabalho para determinar cálculos que podem ser estruturados como fluxos paralelos, a aplicação de transformações de Winograd para reduzir a complexidade computacional das camadas CONV e a utilização de múltiplos processadores de camada CONV para maximizar a utilização de recursos e melhorar o desempenho.

Diversas técnicas foram empregadas para melhorar a eficiência da memória e maximizar a utilização de recursos, como a criação de lotes por agrupamento de diferentes mapas de entrada de características e processamento em conjunto nas camadas totalmente conectadas (FC), a utilização de padrões de acesso complexos e a localização de dados para melhor reutilização de dados, e a otimização do comprimento fracionário para pesos e mapas de características em cada camada [11].

Recomendações

Para maximizar a efetividade de FPGAs na aceleração de CNNs, recomenda-se o desenvolvimento de um framework que inclui uma interface de usuário que permita especificar facilmente o modelo de CNN a ser acelerado, especificando parâmetros como o número de camadas de convolução e seus tamanhos, e o número de camadas totalmente conectadas, entre outras operações intermediárias. O framework deveria então realizar otimizações para encontrar o número mínimo de bits necessários para representar os pesos e mapas de características e o número de bits fracionários a serem usados para cada camada. Além disso, a otimização das camadas totalmente conectadas seria realizada para minimizar os requisitos de memória.

O framework proposto deveria ser baseado no desenvolvimento de uma arquitetura de hardware escalável que funcione para qualquer plataforma FPGA dada e que atinja um aumento de velocidade com a disponibilidade de recursos mais altos. Com base nos recursos disponíveis, o framework faria otimizações para maximizar o paralelismo e a reutilização de dados, dadas as limitações de recursos. Depois disso, geraria automaticamente o modelo de CNN que se ajusta à plataforma FPGA dada e permitiria que o usuário avaliasse o desempenho com base na biblioteca de aplicativos escolhida. Adicionalmente, o framework deveria ter a opção de gerar medidas de desempenho com base em diferentes métricas de desempenho selecionadas pelo usuário. Além disso, ele forneceria outras métricas de projeto, como utilização de recursos, tamanhos de memória e largura de banda, e dissipação de energia [11].

# 3. ARQUITETURA DE REFERÊNCIA

O presente trabalho ter por base o acelerador de convolução proposto na Tese de Doutorado de Leonardo Rezende Juracy [4]. A Tese propõe métodos para realizar uma exploração do espaço de projeto (DSE — do inglês, *design space exploration*) rápida e precisa para aceleradores de aprendizado de máquina, levando em conta diferentes arquiteturas de Redes Neurais Convolucionais (CNNs), usando o framework TensorFlow [12]. O método é abrangente na estimativa de potência, desempenho e área (PPA), permitindo ao projetista selecionar os parâmetros mais relevantes para a implementação de um acelerador de hardware. Os parâmetros de hardware considerados incluem o número de aceleradores em paralelo, o tipo de acelerador e o fluxo de dados.

Para atingir o objetivo estratégico, a Tese estabelece várias metas específicas, incluindo a integração de um *framework* de alto nível (TensorFlow) com uma biblioteca de aceleradores, a implementação de diferentes aceleradores de hardware para CNNs, e um método para comparar diferentes tipos de aceleradores. Além disso, a Tese apresenta a definição e execução de um fluxo de síntese física para a biblioteca de aceleradores de hardware, a definição de um método para extrair dados de PPA e a execução de DSE no *framework* de alto nível.

A Tese apresenta quatro contribuições originais para a área de aceleradores de hardware para Redes Neurais Convolucionais (CNN):

1. Utilização do TensorFlow como um framework front-end para realizar a exploração rápida e precisa do espaço de projeto para esses aceleradores, usando dados da síntese física dos aceleradores, não apenas de componentes básicos.
2. Desenvolvimento de uma biblioteca de aceleradores de hardware CNN, detalhando a arquitetura do hardware.
3. Um método para comparar diferentes aceleradores de hardware para CNNs, levando em consideração parâmetros como o nó de tecnologia, frequência e tipo de memória.
4. Apresentação de um método analítico para realizar DSE, integrado ao TensorFlow, para estimar área, desempenho e potência de maneira rápida e precisa.

## 3.1 Modelo TensorFlow

O uso do TensorFlow, utilizado para modelar a arquitetura de CNNs permite a exploração de sua estrutura, a extração de valores de pesos e bias da rede selecionada, e a obtenção de valores de saída para validar a simulação RTL (modelo *gold*).

A implementação dos modelos de CNN compreende duas fases: treinamento e inferência [3]. A fase de treinamento define os valores de pesos e *bias*. A fase de inferência

usa pesos e *bias* previamente calculados para classificar ou prever valores de saída usando entradas desconhecidas. As vantagens dos modelos de redes neurais em relação a questões de classificação levaram à expansão de estruturas que auxiliam os desenvolvedores a construir seus modelos, oferecendo mecanismos necessários para treinamento e inferência. Exemplos de *frameworks* incluem Caffe [1], Pytorch [10] e TensorFlow [12]. Esses *frameworks* fornecem bibliotecas para implementar aplicações de Machine Learning, incluindo CNNs, que permitem a realização de fases de treinamento e inferência de maneira simplificada, baseadas em linguagens de programação de alto nível como Python. Além disso, esses *frameworks* suportam as funções mais comuns de CNNs, como convolução, *max pooling* e ReLU [7]. A Figura 3.1 mostra um exemplo de um código TensorFlow.

```
# Clearup everything before running
keras.backend.clear_session()

# Create model
model = keras.models.Sequential()

# Add layers
model.add(keras.layers.Conv2D(16, (3,3), strides=(2, 2), activation='relu', input_shape=(28, 28, 1)))
model.add(keras.layers.Conv2D(8,  (3,3), strides=(1, 1), activation='relu'))
model.add(keras.layers.Conv2D(3,  (3,3), strides=(2, 2), activation='relu'))
model.add(keras.layers.Conv2D(1,  (3,3), strides=(1, 1), activation='relu'))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(10, activation='softmax'))

# Build model and print summary
model.build(input_shape=featureShape)
model.summary()
```

Figura 3.1 – Exemplo de código para o *TensorFlow*.

No contexto deste trabalho, após a fase de treinamento, é realizada a exportação dos *bias*, pesos (*weights*) e características (*features*), e é realizada a quantização destes dados para serão usados na simulação RTL do acelerador convolucional. Este trabalho utiliza uma quantização baseada em números inteiros para evitar operações de ponto flutuante, reduzindo a complexidade da implementação do hardware.

## 3.2 Arquitetura do Acelerador

O arranjo dos componentes aritméticos (MACs — *multiply–accumulate*) em relação à interconexão dos mesmos pode ser classificado em duas categorias: 1D e 2D. Na arquitetura 1D, os MACs estão interconectados sequencialmente, com cada MAC tendo, no máximo, dois vizinhos, resultando em uma transferência de dados em cadeia. Por outro lado, o arranjo 2D permite organizar os MACs na forma de uma matriz.

Além da classificação espacial (1D e 2D), os aceleradores também são classificados quanto ao fluxo de dados, ou seja, a forma na qual são carregados os valores nos buffers internos do acelerador. Os fluxos de dados mais comumente empregados são: *Weight*

*Stationary* (**WS**), *Input Stationary* (**IS**), *Output Stationary* (**OS**), *No Local Reuse* (**NLR**), *Row Stationary* (**RS**) e *Fine-Grained* (**FG**) [9, 13].

A trabalho de referência [4] aponta que o arranjo 2D sistólico, com *dataflow* WS, é a arquitetura de acelerador que apresenta o melhor compromisso entre área, consumo e desempenho.

A Figura 3.2 ilustra a arquitetura do acelerador sistólico 2D, com *dataflow* WS [1]. A arquitetura consiste em interfaces externas, registradores internos, além de um núcleo aritmético com uma matriz $3 \times 3$ fixa, contendo 3 multiplicadores, 6 MACs e 3 somadores. O processo começa com o carregamento dos valores de peso e *bias* em buffers. A convolução é controlada por uma máquina de estados finita (*Control FSM*). O acelerador requer sete ciclos para a leitura dos dados, que é controlada por uma FSM de busca (não incluída na Figura), com a computação da convolução sendo executada em paralelo à leitura da memória. Posteriormente, executa-se a ReLU como função de ativação. A arquitetura assume filtros de peso $3 \times 3$ e passo (*stride*) igual a 2.

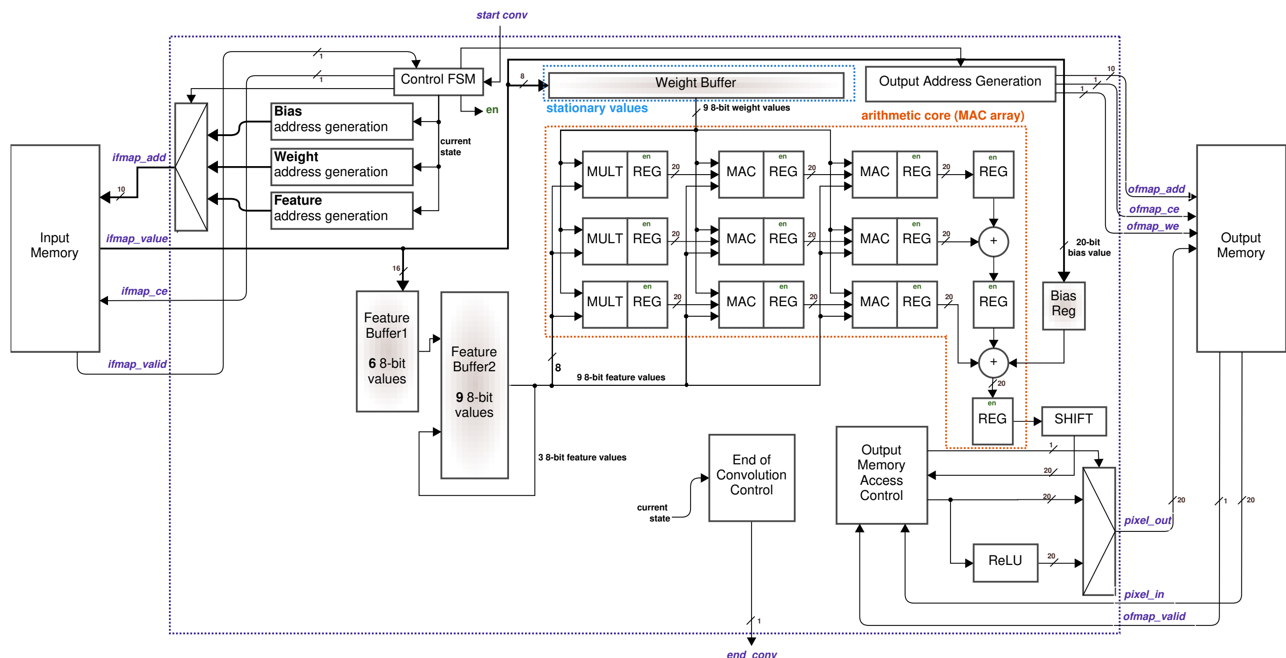



Figura 3.2 – Acelerador 2D WS, com interconexão às interfaces de memória [4].

A inicialização envolve o carregamento dos valores dos pesos e *bias* nos respectivos buffers. Após a ativação do sinal `start_conv`, a convolução é executada, até finalizar a escrita do OFMAP na memória externa e o envio do sinal `end_conv`.

Conforme ilustrado na Figura 3.2, a construção de aceleradores convolucionais requer a implementação de três módulos principais: o primeiro é a memória de entrada (*input memory*), a qual armazena o IFMAP, os pesos dos filtros e os valores de *bias*, sendo

[1] Descrição VHDL:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/rtl/convolution/syst2d_ws.vhd

caracterizada como uma memória de leitura apenas; o segundo é o núcleo convolucional, encarregado de realizar a operação de convolução; finalmente, o terceiro módulo é a memória OFMAP (*output memory*), onde são armazenados os valores parciais e completos resultantes da convolução.

O núcleo convolucional, que executa a convolução, contém buffer de entrada, uma lógica de controle de acesso à memória de entrada, uma matriz MAC, uma função de ativação e uma lógica de controle de saída. A função de ativação não linear adotada é a ReLU devido à sua implementação de hardware mais simples.

Descreve-se abaixo as portas de entrada e saída do acelerador:

- `ifmap_add` e `ofmap_add`: correspondem aos endereços de memória para IFMAP e OFMAP, respectivamente;
- `ifmap_valid` e `ofmap_valid`: são sinais associados à latência da memória, sinalizando quando um dado da memória está apto para uso;
- `ifmap_ce` e `ofmap_ce`: controlam o acesso às memórias, regulando o acesso às mesmas;
- `ifmap_value`: representa os dados provenientes da memória IFMAP, com o sinal `ifmap_valid` indicando a disponibilidade desses dados para consumo;
- `pixel_in`: são os dados originários da memória OFMAP, sendo a disponibilidade para consumo indicada pelo sinal `ofmap_valid`;
- `pixel_out`: se refere aos dados que serão armazenados na memória OFMAP, sendo utilizados para o armazenamento de valores de somas parciais produzidos durante a operação de convolução;
- `start_conv`: é o sinal de entrada que dispara o início da operação de convolução;
- `end_conv`: funciona como um sinal de saída, indicando a conclusão de toda a operação de convolução.

## 3.3 *Dataflow Weight Stationary – WS*

Revisamos neste TCC apenas o *dataflow* WS, dado que ele possui o melhor compromisso entre área ($\mu m^2$), consumo (*nJ*) e desempenho (ciclos) [4, 5].

O Algoritmo 3.1 apresenta o fluxo de dados para o acelerador WS. A principal característica é o uso do mesmo conjunto de filtros de pesos para toda a janela de entrada, reduzindo assim a necessidade de acessos constantes à memória para recuperar pesos diferentes (linhas 5-6 do algoritmo). O laço interno, linhas 7-11, executa todas as convoluções de um canal de entrada (*c*), armazenando o resultado na memória OFMAP. O laço externo, linhas 4-12, controla qual canal está sendo processado.

1: **Input** $C$ input channels $F$ output channels
2: **Output** *Out*
3: **for** $f$ in $F$ **do**
4: **for** $c$ in $C$ **do**
5: **Read** weight filter set $w(f, c)$ from input memory
6: **Store** filter set in the input buffer // weight stationary
7: **for** $I(i)(j)$ in IFMAP($c$) **do**
8: **Read** a window $I(i)(j)$ from IFMAP
9: $p \leftarrow convolution(I(i)(j), w(f, c))$
10: $Out[f][x][y] \leftarrow O[f][x][y] + p$
11: **end for**
12: **end for**
13: **end for**

Algoritmo 3.1 – Pseudo código do *dataflow* WS ([4]).

Este acelerador, apresentado na Figura 3.2, é controlado por uma FSM, apresentada na Figura 3.3. Esta FSM opera da seguinte forma:

- Inicialmente, no estado **WAIT START**, a FSM aguarda o sinal de início da convolução.
- No estado **READ BIAS**, o acelerador lê um valor da memória IFMAP.
- Após a leitura do *bias*, no estado **READ WEIGHT**, o acelerador lê os pesos. Este estado que caracteriza essa arquitetura como *weight stationary*.
- No estado **START MAC** os MACs iniciam a convolução de um determinado canal.
- No estado **WAIT CONV**, é aguardada o fim das convoluções em um dado canal. Após isso, retorna para **READ BIAS**.
- Quando todos os filtros são lidos pelo acelerador, a convolução termina e retorna ao estado **WAIT START**.

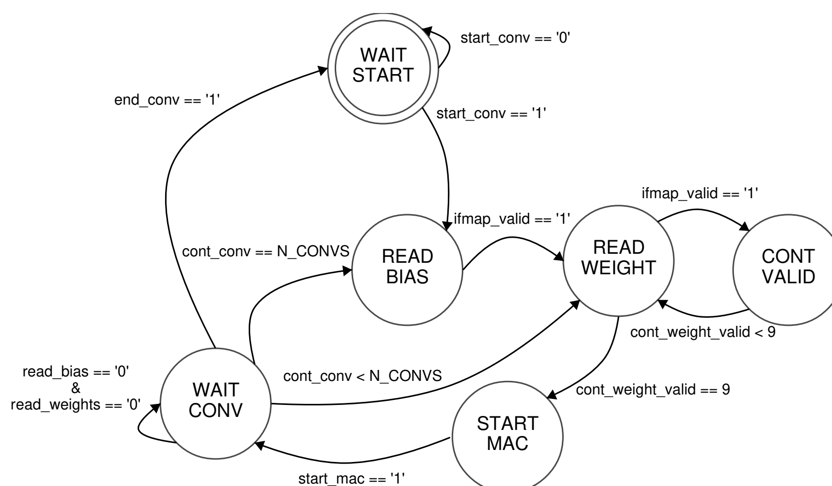



Figura 3.3 – Máquina de estados de controle da convolução WS

A transição do estado **START MAC** para **WAIT CONV** inicia a execução da "FSM de busca", responsável por calcular as convoluções e buscar os valores do IFMAP. A FSM de Busca, detalhada na Figura 3.4, executa o núcleo do Algoritmo 3.1, ou seja, o loop entre

as linhas 5-9. Após a ativação do sinal `start_mac`, o acelerador começa a buscar valores IFMAP da memória. Os estados **FECTH IFMAP** e **CONT VALID** leem valores IFMAP, calculam o valor de convolução parcial, armazenando-o na memória OFMAP ou em um buffer. Em seguida, o estado **UPDATE_ADD** gera um novo endereço IFMAP até o final do cálculo do canal.

No final do cálculo de um canal, é necessário ler um novo valor de *bias* ou um novo conjunto de pesos (sinais `read_bias` e `read_weights`). Nesse caso, a FSM de Busca retorna ao estado **IDLE**, liberando a FSM de Controle. A Figura 3.4 também mostra o esquema de reutilização com base no *stride*. A FSM de Busca lê 6 valores IFMAP para cada operação de convolução em vez de 9, que é a janela IFMAP usada nas convoluções.

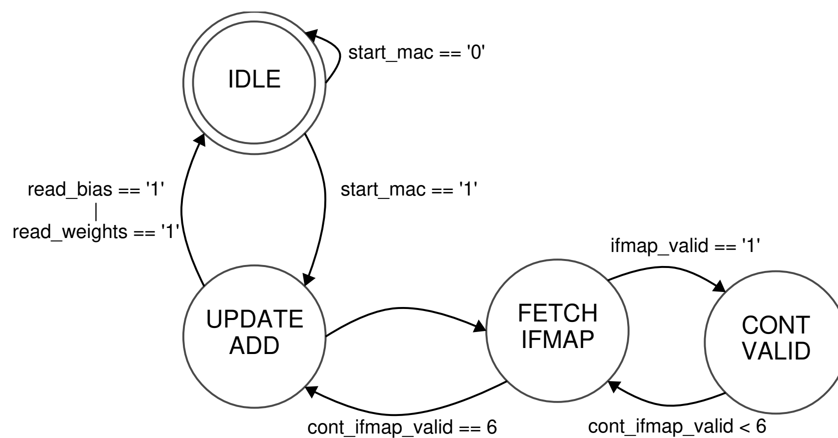



Figura 3.4 – FSM de busca de dados para o acelerador WS.

## 3.4 Módulos do Acelerador Modificados no Contexto do TCC

Esta seção apresenta os módulos do acelerador que necessitaram ser alterados no contexto deste TCC.

### 3.4.1 Estrutura de Memória do Acelerador de Referência

A arquitetura apresentada na Figura 3.5 é uma memória comportamental [2]. Isto se deve ao fato dela ser modelada usando estruturas de dados de alto nível e operações que se concentram no comportamento funcional desejado, em vez de uma implementação de hardware específica.

[2]Descrição VHDL:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/rtl/components/mem.vhd

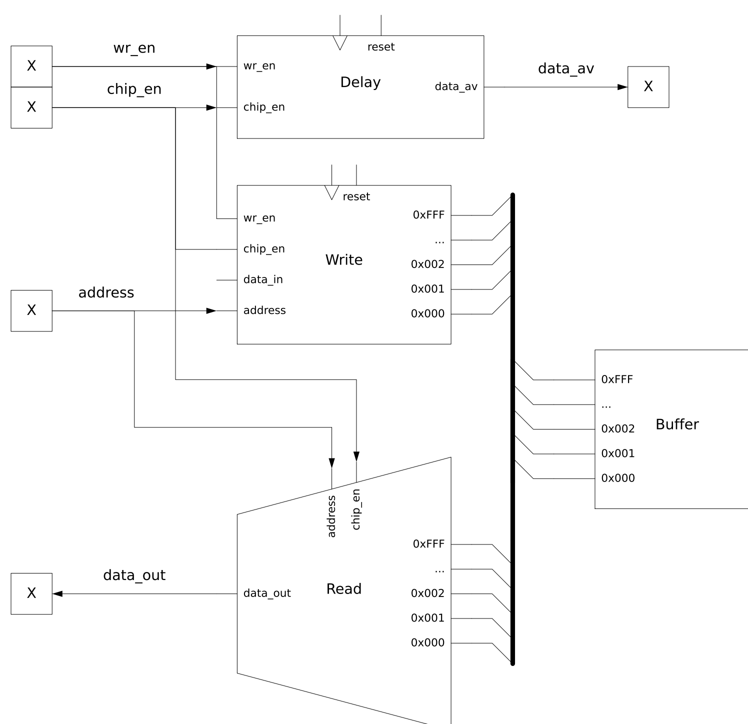



Figura 3.5 – Arquitetura de memória da arquitetura de referência.

A arquitetura de memória é composta por um *buffer* de registradores, um processo de escrita, *Write*, um multiplexador que controla a saída, *data_out*, e uma unidade de controle de atraso, *Delay Control*. As operações de leitura e escrita na memória são implementadas como processos VHDL, que são ativados por um sinal de relógio. O processo de escrita na memória, *Write*, verifica os sinais `chip_en` e `wr_en`, e se ambos estão ativos, o conteúdo do vetor de entrada `data_in` é escrito no *Buffer* no endereço especificado pelo *address*. A operação de leitura da memória é realizada por meio de uma atribuição de sinal concorrente, um multiplexador, que atualiza o valor de `data_out` dependendo dos sinais `chip_en` e *address*.

O módulo *Delay Control* possui duas funções. A primeira é garantir a sincronização entre a leitura de uma informação e o módulo que solicitou essa informação. A segunda é prover a capacidade de simular a latência inerente ao funcionamento da memória.

A Figura 3.6 apresenta o mapa de memória organizado em três seções: *bias*, *weights* e *feature map*, que são os componentes em uma operação de convolução.

- Seção bias: parte inferior do mapa de memória. Bias, ou viés, é um parâmetro em redes neurais que é usado para ajustar a saída da rede juntamente com os pesos. O viés está localizado nos endereços iniciais da memória, começando em `0x0000`, e ocupa uma quantidade de espaço correspondente à quantidade de filtros, indicada por $n_{filter}$.

- Seção weights (pesos): ocupa um espaço maior na memória. Os pesos se referem aos coeficientes dos filtros convolucionais, que são aprendidos durante o treinamento da rede neural. A quantidade de memória que os pesos ocupam é determinada pela dimensão dos filtros (largura x altura, neste caso $filter^2_{width}$), o número de canais de entrada da imagem ($n_{channel}$) e o número total de filtros na camada convolucional ($n_{filter}$).
- Seção feature map, ou mapa de características. Corresponde à entrada de uma camada convolucional. A quantidade de memória ocupada por uma *feature map* é determinada pelo quadrado da sua largura ($feature^2_{width}$) e o número de canais de entrada da imagem ($n_{channel}$).

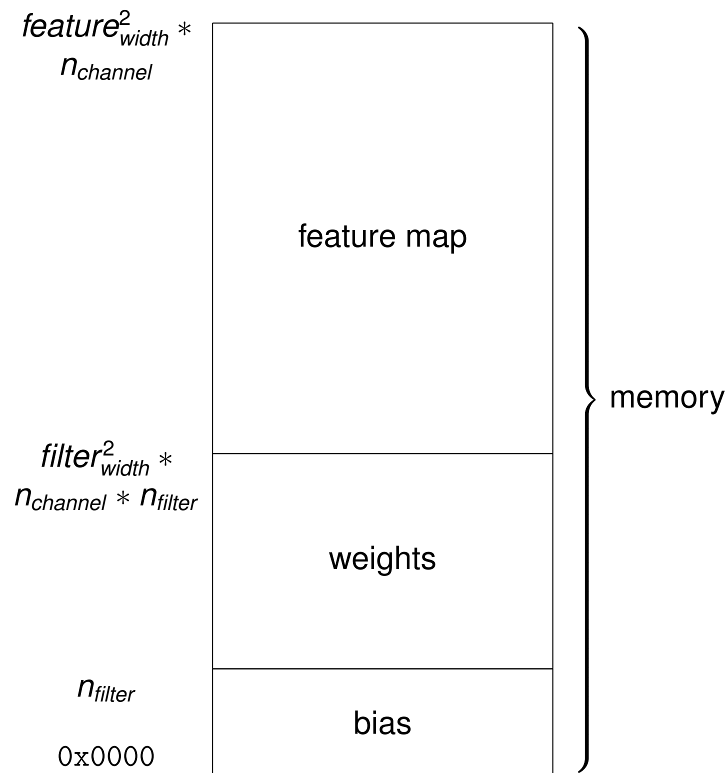



Figura 3.6 – Partição da memória de entrada do acelerador de referência.

Essas três seções representam a organização da memória em uma camada convolucional de uma CNN, em que cada seção do mapa de memória é reservada para um tipo específico de dado necessário para a operação de convolução.

### 3.4.2 Lógica para Controle de Acesso à Memória

Os multiplexadores apresentados na Figura 3.7 são responsáveis por selecionar o endereço de memória que será lido, `inmem_address`. Por meio de uma série de condições, estabelece-se qual valor será atribuído ao endereço de memória.

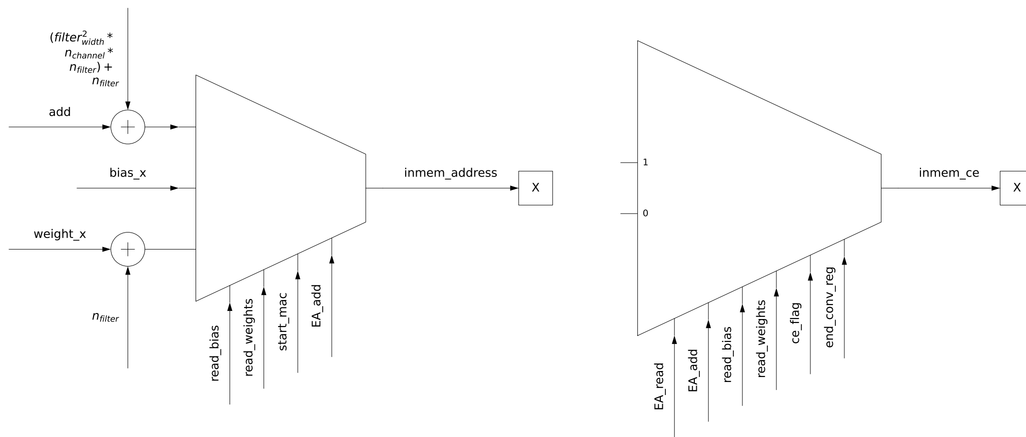



Figura 3.7 – Multiplexadores para Controle de Acesso à Memória

Para acessar os *bias*, o circuito verifica a ativação da leitura do *bias*, ou seja, se `read_bias` está alto. Caso afirmativo, o endereço do *bias*, `bias_x`, é atribuído ao endereço de memória, `inmem_address`, indicando que o *bias* deve ser acessado.

O acesso aos pesos é feito em blocos de 9 palavras (tamanho da janela de convolução). Se o estado correspondente à leitura dos pesos estiver ativo, `read_weights=1`, ou caso tenha sido iniciada uma operação de multiplicação e acumulação (MAC), `start_mac=1`, o valor do endereço relativo ao peso e o deslocamento referente ao total de pesos, representado por `weight_x` + $n_{filter}$, é atribuído a `inmem_address`.

O segundo multiplexador apresentado na Figura 3.7 apresenta o comportamento que define o estado do sinal de controle de leitura para a memória, `inmem_ce`. Estão envolvidas quatro condições, que compreendem seis sinais. Quando `inmem_ce=1` é realizada uma operação na memória.

Quatro condições desabilitam o acesso à memória (`inmem_ce=0`):

- FSM de controle do acelerador estiver no estado de espera para iniciar uma operação de leitura, **WAITSTART** (Figura 3.3).
- FSM de busca (Figura 3.4) estiver no estado de atualização do endereço (**UPDATEADD**).
- Flag de controle de leitura (`ce_flag`) estiver ativo.
- Registrador de término da convolução (`end_conv_reg`) estiver ativo.

Dessa maneira, caso qualquer uma dessas condições seja atendida, o `inmem_ce` assume o valor '0', desativando a operação na memória. Em contrapartida, se nenhuma dessas condições for satisfeita, o `inmem_ce` se mantém '1', permitindo a operação na memória.

### 3.4.3 *Test bench*

O *test bench* [3] é composto pela convolução, uma memória de entrada — *inmem*, uma memória para as características de saída, *ofmap*, um buffer com o os dados corretos, *gold*, e um processo que compara os dados de saída com o *gold*.

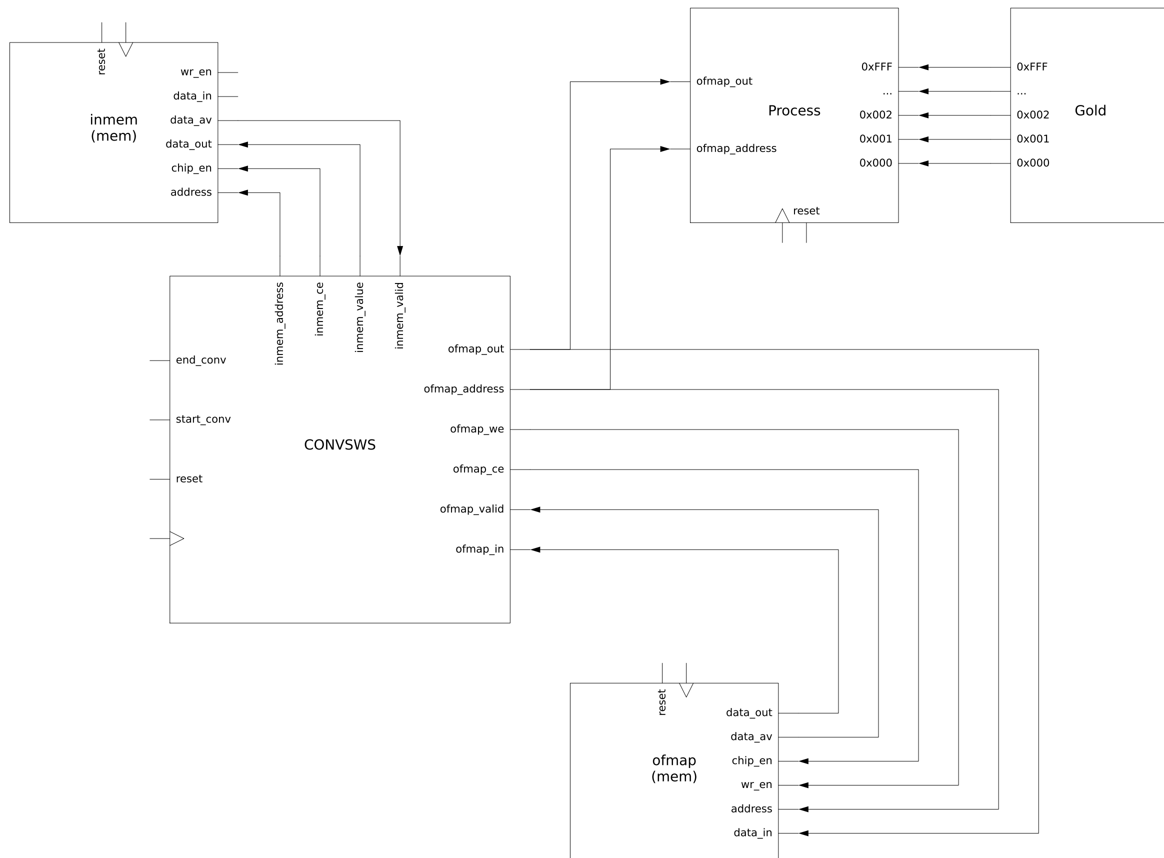



Figura 3.8 – *Test bench* da convolução do trabalho de referência

Este *test bench* por ter as saídas esperadas armazenadas no módulo *gold* permite a validação da convolução de forma automatizada, sem necessitar recorrer a formas de onda.

[3] Descrição VHDL:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/tb/tb_rtl.vhd

# 4. PROJETO DO ACELERADOR DE HARDWARE PARA APRENDIZADO DE MÁQUINA

Este Capítulo corresponde à principal à contribuição deste TCC, que é o projeto de um acelerador de hardware para aprendizado de máquina, tendo por base o acelerador 2D WS sistólico apresentando no capítulo anterior (Figura 3.2), o qual o denominamos **CONVWS**. Este Capítulo é organizado em 7 seções, seguindo o fluxo de desenvolvimento deste TCC:

- Seção 4.1 apresenta a primeira alteração realizada na arquitetura de referência, a qual corresponde à alteração da configuração das memórias CONVWS, criando duas memórias, uma para pesos e viés (MEMWGHT), e outra para os mapas de características (MEMFMAP).
- Seção 4.2 apresenta o módulo *Core*, o qual encapsula os módulos necessários para executar as operações de uma camada da rede neural, incluindo as memórias MEMFMAP e MEMWGHT.
- Seção 4.3 apresenta a parametrização do CONVWS para que seja possível utilizar o mesmo em múltiplas camadas da CNN.
- Seção 4.4 simplifica módulo *Core* pela remoção da MEMFMAP de saída, permitindo a conexão dos módulos de aceleração em série. O novo módulo, com várias convoluções conectadas em série, é denominado de *CNN Simple*.
- Seção 4.5 apresenta a substituição das memórias comportamentais (declaradas como vetores) para memórias BRAMs, que são os módulos de memória disponíveis nos FPGAs Xinlix. Este trabalho foi um dos mais complexos no contexto deste TCC, dada a complexidade de se gerar os arquivos de configuração e inicialização das BRAMs.
- Seção 4.6 apresenta o projeto das camadas *max polling* e *fully connected*, não desenvolvidos no trabalho de referência. Com a integração destes módulos torna-se possível a implementação de uma CNN completa em dispositivos FPGAs.
- Seção 4.7 apresenta o projeto implementado em um dispositivo FPGA Xilinx, visando a validação do projeto em hardware en não apenas por simulação RTL.

## 4.1 Alterações na Arquitetura de Referência

A divisão da memória de entrada em duas, denominadas “memória de pesos e viés” (MEMWGHT) e “memória de características” (MEMFMAP), apresenta vantagens em

termos de eficiência e implementação. Uma das principais razões para essa divisão é a facilidade de geração de Blocos de Memória (BRAMs) para dados em duas memórias separadas. A MEMWGHT é utilizada exclusivamente para operações de leitura, enquanto a MEMFMAP permite tanto operações de leitura quanto de escrita. A separação dessas memórias permite otimizar o desempenho e a utilização dos recursos de hardware disponíveis, resultando em uma implementação mais eficiente.

Ao analisar estrutura da memória de entrada (Seção 3.4.1), avaliamos duas opções de implementação: uma preservando a arquitetura de convolução original, e a outra propondo sua alteração.

A primeira opção envolve a conexão de um demultiplexador nas portas de saída (`inmem_ce` e `inmem_address`) da memória de entrada (`inmem`). Isso direcionaria o endereço e a habilitação de leitura para a memória de pesos (e viés) ou para a memória de características. Além disso, seria necessário que esse demultiplexador recebesse sinais de escrita para novos pesos, viés e características oriundos de uma camada anterior. Essa abordagem, no entanto, impõe a necessidade de introduzir sinais adicionais de controle para alternar entre endereços e dados de pesos e características.

Embora a segunda opção exija a alteração da arquitetura da CONVWS, optou-se por esta implementação por sua simplicidade e por não ter como requisito sinais adicionais de controle de escrita.

### 4.1.1 Memórias MEMWGHT e MEMFMAP

A Figura 4.1 apresenta a partição de endereços para a MEMWGHT. Esta memória armazena os coeficientes dos filtros convolucionais, ou pesos, e os parâmetros de ajuste de saída da rede, denominado viés. A porção superior desta memória, rotulada como “weight”, é destinada aos pesos e ocupa um espaço determinado pela dimensão dos filtros (representada por $filter^2_{width}$), o número de canais de entrada da imagem ($channel_n$) e o número total de filtros na camada convolucional ($filter_n$). A parte inferior desta memória, rotulada como “bias”, é destinada ao viés e está localizada nos endereços iniciais da memória, a partir de `0x0000`, ocupando um espaço correspondente ao número de filtros ($filter_n$).

A Figura 4.2 apresenta a partição de endereços para a MEMFMAP, utilizada para armazenar a entrada de uma camada convolucional, conhecida como mapa de características (*feature map*) de entrada (IFMAP) ou saída (OFMAP). A quantidade de memória necessária é determinada pela multiplicação da altura pela largura do feature map ($feature^2_{width}$), que sempre tem a mesma medida, e pelo número de canais de entrada da imagem ($n_{channel}$). Este mapa de características é armazenado nos endereços iniciais da memória, a partir de `0x0000`.

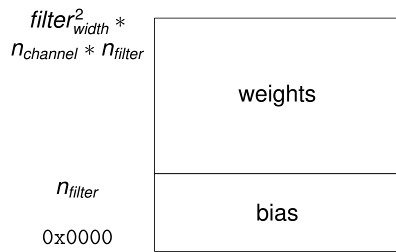



Figura 4.1 – Partição de endereços para a memória de pesos e bias (MEMWGHT).

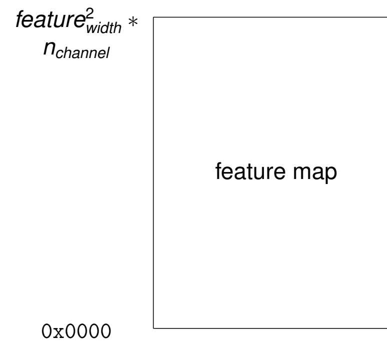



Figura 4.2 – Partição de endereços para a memória de características (MEMFMAP).

### 4.1.2 Lógica para acesso às Memórias

O primeiro multiplexador do CONVWS[1] apresentado na Figura 4.3 determina o endereço de memória correspondente aos pesos, identificado como `iwght_address`. Esta lógica é semelhante ao primeiro multiplexador da arquitetura anterior (Figura 3.7), preservando a primeira e segunda condições. Contudo, há algumas diferenças. A terceira condição, que corresponde à leitura das características de cada convolução, ou seja, `EA_add`, não está presente. Ademais, o deslocamento que considerava a totalidade dos endereços de pesos e do viés foi removido.

Quanto ao multiplexador responsável por definir o endereço para a leitura das características, `ifmap_address` refere-se a um estado associado à leitura dos dados da convolução (`EA_add`). Importante notar que o deslocamento que levava em consideração a totalidade da memória de pesos e do viés foi removido, dado que as características agora estão posicionadas desde o início endereços.

Os multiplexadores que habilitam a leitura da memória de pesos e características também é semelhante à lógica anterior onde algumas das condições são mutuamente exclusivas de um ou outro enquanto outras são comuns a ambos. O multiplexador `iwght_ce` tem as mesmas condições do multiplexador anterior em relação aos sinais `EA_read`, `ce_flag` e `end_conv_reg`. Já o multiplexador com saída na porta `ifmap_ce` tem as mesmas condições do multiplexador anterior em relação aos sinais `EA_add`, `read_bias`, `read_weights`, `ce_flag` e `end_conv_reg`.

[1] Descrição VHDL: https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/rtl/convolution/syst2d_ws_split.vhd

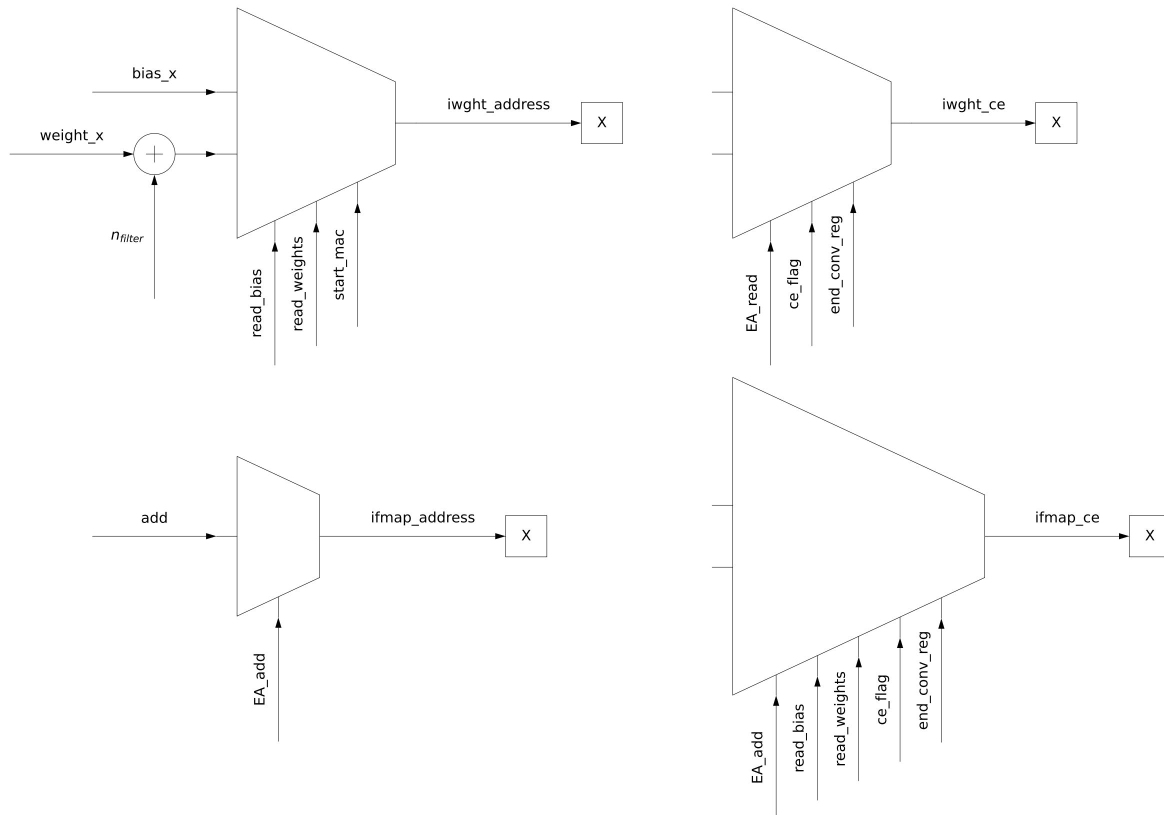



Figura 4.3 – Lógica para acesso às memórias do CONVWS.

### 4.1.3 *Test bench*

O *test bench*[2] também foi alterado para suportar duas memórias. A Figura 4.4 apresenta a nova arquitetura de teste. Observar que há um processo que lê os dados gerados e os comparada com a referência *gold*, acelerando o processo de verificação. Esta modificação reflete a divisão da memória de entrada em entidades de memória separadas dedicadas aos mapas de características (MEMFMAP) e à outra memória para os pesos e o viés (MEMWGHT).

## 4.2 Módulo *Core*

A segunda etapa do trabalho corresponde à implementação do módulo *Core* (CORE), o qual encapsula os módulos necessários para executar as operações de uma camada da

[2]Descrição VHDL:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/tb/tb_rtl_split.vhd

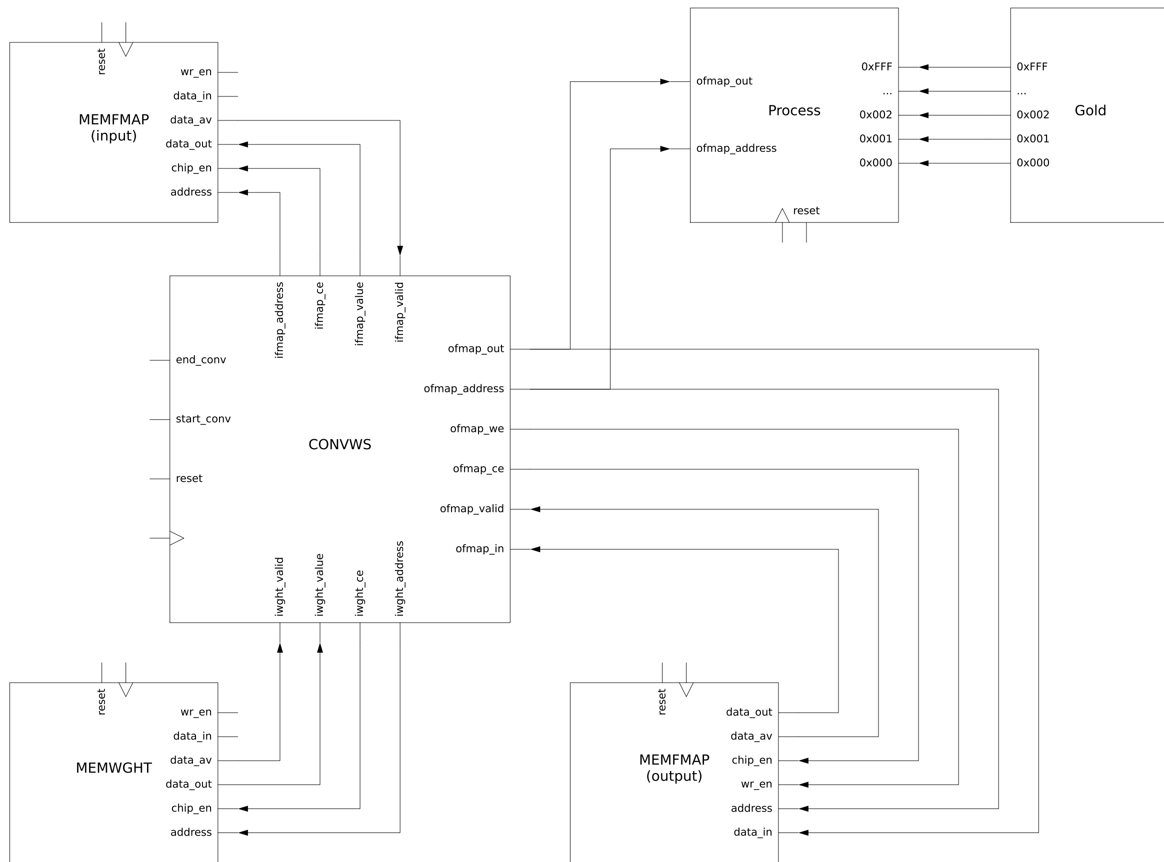



Figura 4.4 – Arquitetura do *test bench* com duas memórias.

rede neural. Esses elementos incluem as memórias MEMFMAP e MEMWGHT, bem como o acelerador original, o que engloba a convolução, *pooling* e camadas totalmente conectadas.

### 4.2.1 Core 1-Layer

O objetivo com o *Core 1-Layer*[3] (CORE1), apresentado na Figura 4.5, é criar um módulo que possa ser instanciado *n* vezes. Após a conclusão de cada operação, os dados são movidos por para a próxima camada. O processamento é armazenado na MEMFMAP de saída.

O CORE1 é uma evolução da organização dos módulos existentes no *test bench* da arquitetura de referência (Figura 3.8) e das alterações propostas na Figura 4.4. Foram removidos o processo *test* e *gold*, e foram adicionados multiplexadores para controlar entrada e saída de dados na memória e na convolução.

[3]Descrição VHDL:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/rtl/core/core.vhd

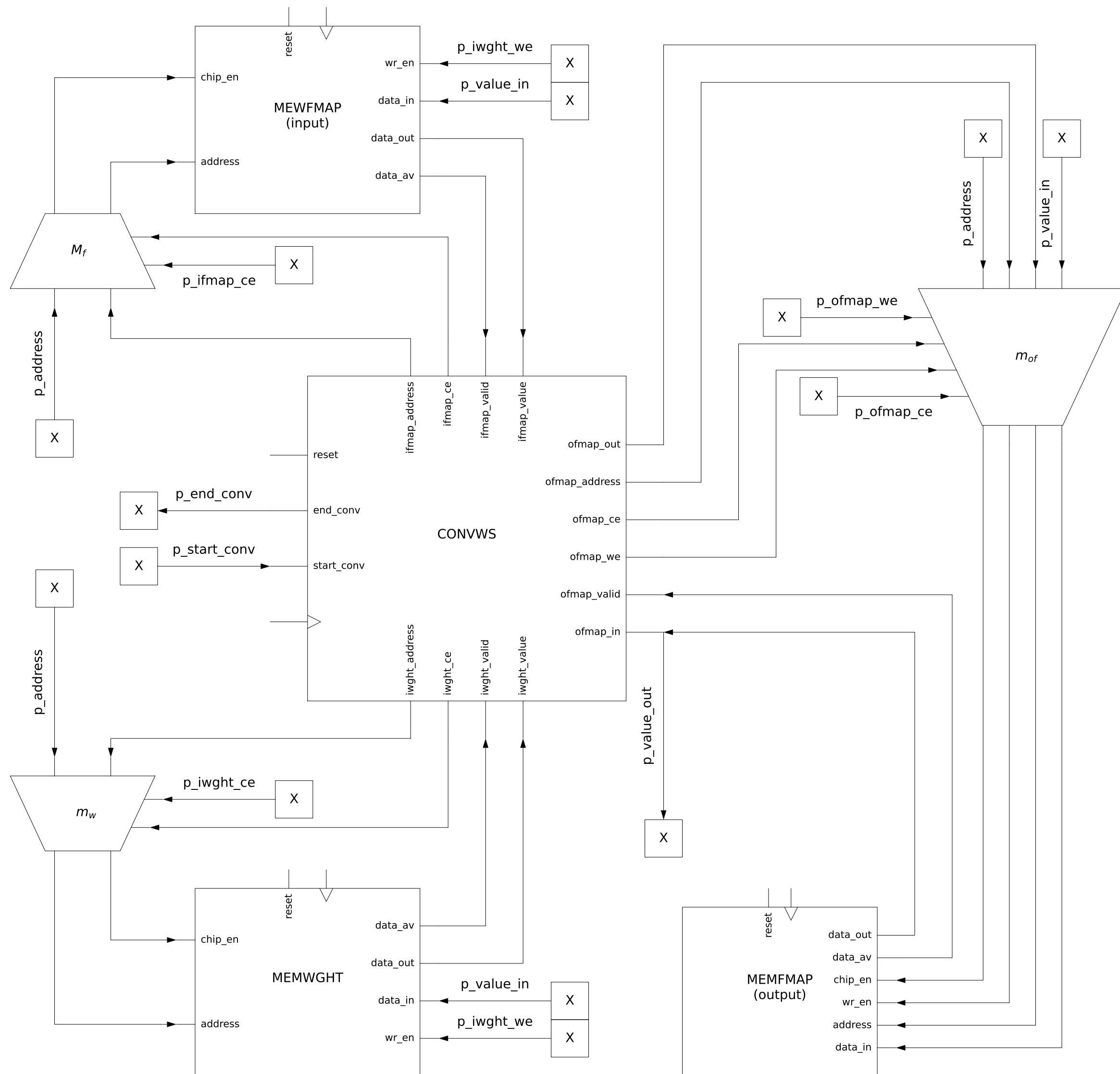



Figura 4.5 – Módulo CORE1.

As principais características a respeito do comportamento das memórias MEMWGHT e MEMFMAP:

- Tanto a convolução quanto um módulo que esteja instanciando o CORE1 podem escrever nas memórias, mas somente a convolução pode ler os dados.
- Em caso de solicitação de escrita para o CORE1 não há suporte para a escrita de dados diferentes nas duas memórias, ou seja, o mesmo dado será escrito em ambas as memórias, pois só há uma porta de entrada de dados.
- Não é possível solicitar leitura dessas memórias, suas portas de saída não estão conectadas a porta de saída do CORE1.

A leitura e escrita nas memórias MEMWGHT e MEMFMAP de entrada são controladas pelos multiplexadores $M_f$ e $M_w$, apresentados na Figura 4.6. Será usado o multiple-

xador de MEMFMAP de entrada ($M_f$) como exemplo, mas o mesmo serva para MEMWGHT ($M_w$):

- Para que o *chip enable* da MEMFMAP seja habilitado basta que o *chip enable* da convolução (`ifmap_ce`) ou o *chip enable* do *Core* (`p_ifmap_ce`) estejam altos.
- O endereço multiplexado virá do *Core* (`p_address`) caso seu *chip enable* (`p_ifmap_ce`) esteja alto, caso contrário virá da convolução (`ifmap_address`).

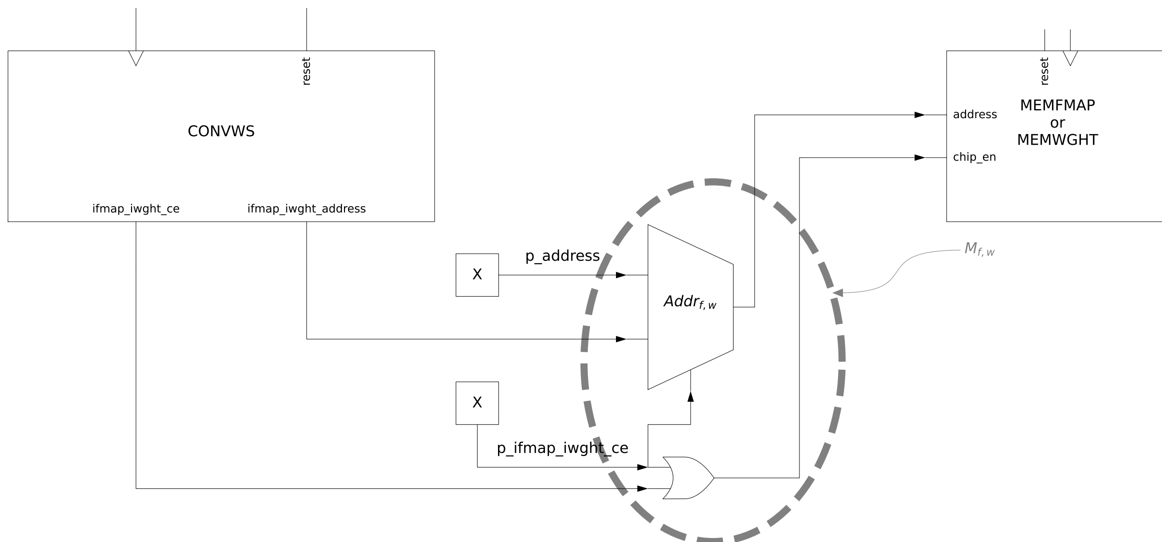



Figura 4.6 – Detalhe de $M_f$ e $M_w$, multiplexadores das memórias MEMFMAP de entrada e MEMWGHT no CORE1

Os sinais de habilitação de escrita, `wr_en`, e entrada de dados, `data_in`, para ambas as memórias vêm das portas de entrada do *Core*, respectivamente, `p_ifmap_we` ou `p_iwght_we`, e `p_value_in` como pode ser visto na Figura 4.5.

O controle da escrita e leitura da MEMFMAP saída feito pelo do multiplexador $M_{of}$ pode ser visto em detalhe na Figura 4.7. O objetivo de $M_{of}$ é permitir que tanto a CONVWS quanto as portas do *Core* tenham acesso de forma organizado e sem conflitos ao MEMFMAP de saída.

A habilitação de leitura e escrita na MEMFMAP de saída é decidida por dois operadores *OR*. Caso `p_ofmap_ce` do *Core* ou `ofmap_ce` de CONVWS estejam com o sinal alto então a escrita é habilitada na MEMFMAP, `chip_en`. A lógica para a habilitação de escrita é semelhante: caso `p_ofmap_we` do *Core* ou `ofmap_we` de CONVWS estejam com o sinal alto então a escrita é habilitada na MEMFMAP, `wr_en`.

A entrada de dados e do endereço são multiplexadas por $M_{Data}$ e $M_{Addr}$ respectivamente. Caso $M_{Data}$ receba `p_ofmap_ce` e `p_ofmap_we` do *Core* com sinal alto então será concatenado `ofmap_pad`, sinal com todos os bits em zero com a largura de bits do *Carry*, com `p_value_in`, porta de entrada de dados do CORE1, em `data_in` de MEMFMAP de

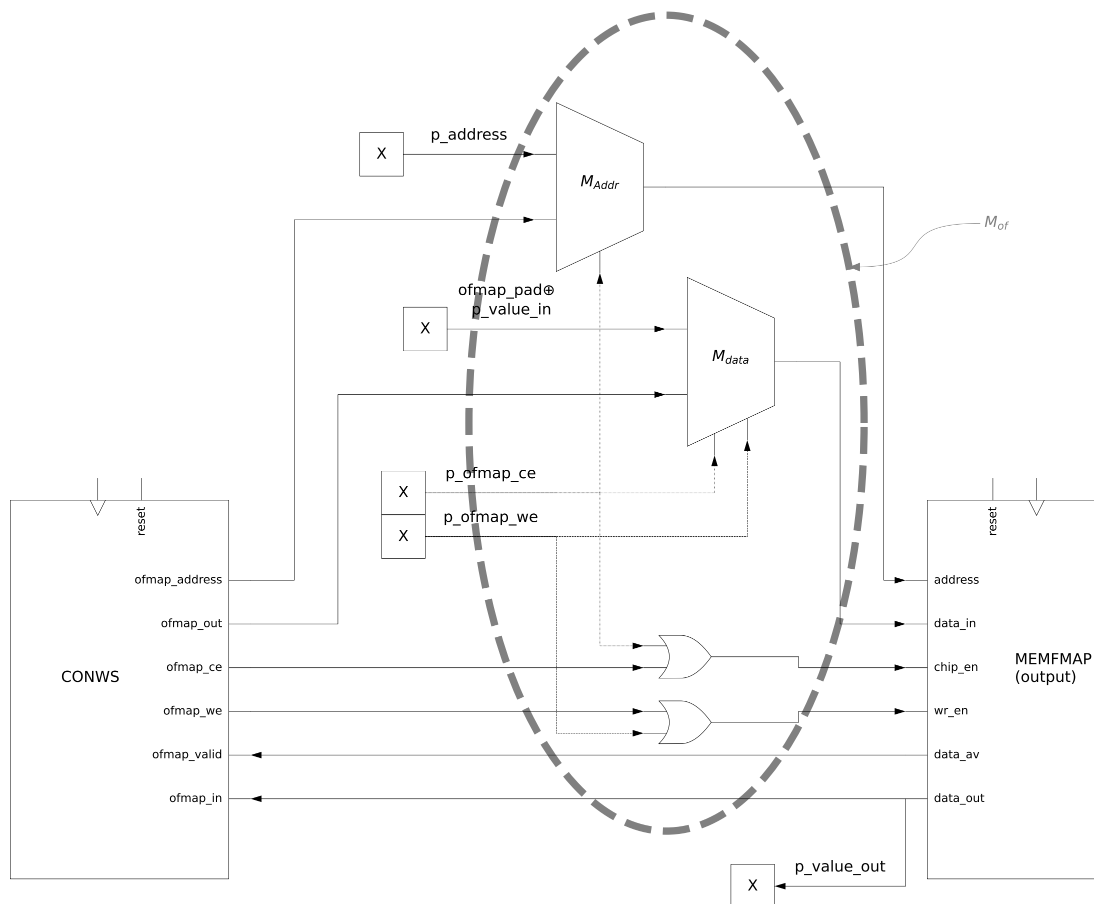



Figura 4.7 – Detalhe de $M_{of}$ multiplexador da MEMFMAP de saída no *Core*

saída. Se $M_{Addr}$ receber `p_ofmap_ce` alto então a MEMFMAP receberá o endereço do *Core* caso contrário receberá da CONVWS.

### 4.2.2 *Test bench*

O *test bench*[4] associado ao CORE1 distingue-se dos demais por utilizar uma descrição não sintetizável. Essa abordagem, que emprega o comando *process* com instruções sequências, foi adotada para simplificar o desenvolvimento. A expectativa é que, posteriormente, esta descrição seja convertida para uma versão sintetizável que utilize uma máquina de estados.

Os elementos presentes neste *test bench* incluem o *process* não sintetizável, o CORE1, os buffers com pesos, viés, características e a saída *gold*, além dos sinais que

[4]Descrição VHDL:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/tb/tb_rtl_core.vhd

conectam todos esses elementos. O principal objetivo desse *test bench* é efetuar testes em todos os módulos da arquitetura. Por esta razão, as memórias estão inicialmente vazias, sem quaisquer dados que serão posteriormente escritos no CORE1.

Após a conclusão da operação de CONVWS, as características finais resultantes, ou OFMAP, são lidas e validadas. Essa validação é realizada por meio de uma comparação com o *gold*.

O algoritmo 4.1 apresenta uma descrição passo-a-passo das operações executadas pelo *test bench*. Este algoritmo oferece uma perspectiva de alto nível das etapas seguidas para garantir o funcionamento adequado do CORE1.

1: **send** signal to enable write in *iwght* and disable to *ifmap*
2: **for** *i* in 0 to $filter^2_{width} * n_{channel} * n_{filter} + n_{filter}$ **do**
3:     **write** weight and bias data to $address_i$ in *iwght*
4: **end for**
5: **send** signal to enable write in *ifmap* and disable to *iwght*
6: **for** *i* in 0 to $x^2_{size} * n_{channel}$ **do**
7:     **write** feature data to $address_i$ in *ifmap*
8: **end for**
9: **send** signal to start convolution and to disable write in *ifmap* and *iwght*
10: **wait** end of convolution
11: **for** *i* in 0 to $convs_{perline} * n_{filter}$ **do**
12:     **read** features for $address_i$ in *ofmap*
13:     **wait** valid signal from *ofmap*
14:     **compare** received feature data with $gold_i$
15: **end for**

Algoritmo 4.1 – Algoritmo do *test bench*.

## 4.3 Execução de múltiplas camadas

O objetivo dessa seção é apresentar a parametrização CONVWS para que seja possível utilizar o mesmo em múltiplas camadas da CNN.

### 4.3.1 Suporte para configuração variável

No trabalho de referência, a configuração das arquiteturas CONVWS era realizada por meio de *generics* na descrição VHDL, os quais determinavam parâmetros como o número de convoluções e o tamanho do IFMAP. No entanto, da forma que é empregada tal abordagem tornava inflexível a parametrização da arquitetura CONVWS, que era especifi-

cada e fixada de acordo com as necessidades de uma única camada convolucional da CNN para cada simulação.

Para a arquitetura CONVWS[5] ser capaz de executar uma operação de diferentes camadas da CNN, tornou-se necessário remover alguns *generics*, como aqueles que definem número de convoluções, e substituí-los por registradores. Entretanto, as informações relevantes precisam ser calculadas previamente e, posteriormente, enviadas à arquitetura CONVWS. A implementação de uma nova porta na CONVWS para receber esses dados foi um requisito surgido dessa nova estrutura, embora sua representação não seja ilustrada nas figuras, por não se considerar que contribua significativamente para a compreensão. Todas as operações entre *generics* que envolve um dos itens na lista abaixo foram removidas da descrição e transformadas em dados pré-calculados:

- `n_filter`: número de filtros convolucionais aplicados, sendo também o número de canais de entrada.
- `n_channel`: número de canais de saída.
- `x_size`: tamanho da imagem ou FMAP.
- `convs_per_line`: convoluções executadas para cada linha de uma imagem ou FMAP.

O número máximo de bits que esses registradores podem ter, ou seu valor máximo, é definido pela escolha do maior valor que o registrador pode assumir entre todas as camadas. Por exemplo, se `n_channel` tiver os valores 3, 16 e 32 para diferentes camadas, então o valor máximo que este registrador poderá assumir será definido como 32.

Os dados pré-calculados são processados para cada uma das camadas convolucionais da CNN e salvos em formato texto em um *package* VHDL.

### 4.3.2 Algoritmo de geração dos dados

No escopo deste trabalho, uma fase crítica engloba a geração de dados adequados para operar todas as camadas convolucionais. Até o presente momento, o framework em uso gera e armazena os dados necessários para uma única camada de cada vez. Temos o objetivo de automatizar a geração de informações de configuração do CONVWS. Nessa etapa, para cada camada, são gerados automaticamente quatro componentes fundamentais: o arquivo de configuração da Rede Neural Convolucional (CNN), os *generics*, os mapas de características de entrada e saída, bem como os pesos e o viés.

[5]Descrição VHDL:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/rtl/convolution/syst2d_ws_split_multi.vhd

Em paralelo, uma meta secundária é a otimização do carregamento de bibliotecas. Foi identificado que a carga das bibliotecas TensorFlow e Keras apresenta tempo significativo de execução, atingindo cerca de dois minutos para conclusão, considerando as configurações atuais. Esse tempo elevado de carregamento tem influência direta na execução dos testes, estendendo a duração dos mesmos de alguns segundos para vários minutos. Isso acarreta atrasos consideráveis no desenvolvimento do projeto, situação que estamos buscando solucionar.

Anteriormente, um único *script* era responsável por executar tanto o treinamento[6] quanto a geração do hardware auxiliar[7]. No entanto, tornou-se evidente que essas são duas tarefas distintas. Como exemplo, um único treinamento pode requerer vários hardwares, e reciprocamente, um único hardware pode ser aplicado em diversos treinamentos. Assim, foi tomada a decisão de dividir o *script* original em dois, um dedicado exclusivamente ao treinamento e outro à geração do hardware. Consequentemente, também foram separadas as configurações para o treinamento e para o hardware, cada uma com seus respectivos parâmetros, os quais são armazenados em arquivos JSON independentes.

Adicionalmente, para evitar conflitos entre os pacotes de memória, pesos, mapa de características e *gold*, que estão associados a diferentes camadas durante o teste, um novo método para exportar esses dados foi implementado. Os dados de memória são exportados para pacotes VHDL. No entanto, todos os pacotes de cada tipo (pesos, mapa de características e gold) compartilham a mesma nomenclatura. Considerando que o VHDL não permite o uso de pacotes distintos com o mesmo nome, duas opções foram inicialmente avaliadas: (1) manter a mesma nomenclatura do pacote e criar uma nova biblioteca ou (2) carregar os dados nos sinais a partir da leitura de arquivos em disco.

A primeira opção revelou-se impraticável, uma vez que demandaria a criação de tantas bibliotecas quanto o número de camadas existentes na CNN, além de necessitar da distinção manual de cada biblioteca. Assim, a segunda alternativa foi escolhida, visando manter a flexibilidade no desenvolvimento e execução do projeto.

Com a adoção desse processo temos como resultado a geração de:

- Para cada camada da CNN temos os arquivo com as parametrizações, mapa de características, pesos e viés, e GOLD todos estes em VHDL e texto, e *generics* de simulação e sintetização.
- Para *Core* temos o arquivo com as parametrizações em texto e VHDL, os *generics* de simulação e síntese de todos os valores máximos da CNN.

[6] Código: https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/apps/cnn-train-cifar10.py

[7] Código: https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/apps/rtl-generate.py

A solução adotada, apresentada na Figura 4.8 , envolve o carregamento de dados para a memória a partir do disco, permite adaptar-se às alterações no número de camadas, evitando a criação de múltiplas bibliotecas. Esse procedimento otimiza o processo de teste e simplifica a manipulação de dados de diferentes camadas.

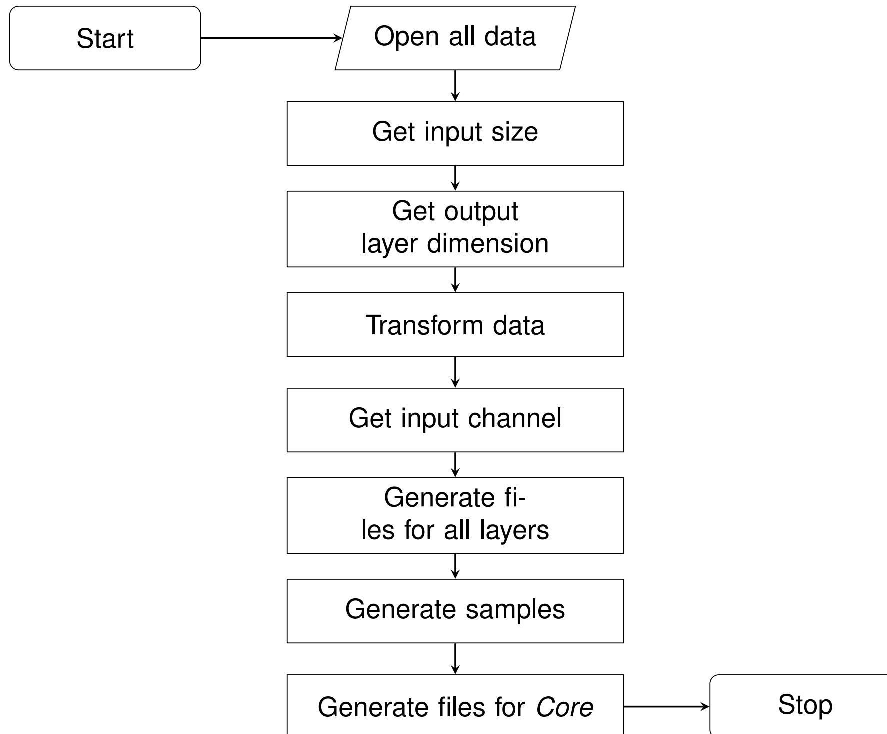



Figura 4.8 – Fluxograma da geração da descrição RTL.

### 4.3.3 *Test bench*

O *test bench* do *Core Multi*[8] (Core que pode executar diferentes camadas em função da parametrização), apresentado no Algoritmo 4.2, tem por objetivo testar a execução de múltiplas camadas em um único *Core Multi*.Também utiliza comandos não sintetizáveis e apresenta os mesmos elementos.

Diferentemente do *test bench* anterior os dados de cada camada (viés, pesos, características e configurações) são carregados do disco dentro de *buffers* e após isso se dá o processo de escrita dos dados no CORE1.

[8]Descrição VHDL:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/tb/tb_rtl_core_multi.vhd

```
1:  read features_0 from file_0
2:  for n in 0 to n_layer - 1 do
3:      read weights_n, config_n and gold_n from file_n
4:      send signal to enable write in iwght and disable to ifmap
5:      for i in 0 to filter^2_width * n_channel * n_filter + n_filter from config_n do
6:          write weights_ni data to address_i in iwght
7:      end for
8:      send signal to enable write in ifmap and disable to iwght
9:      for i in 0 to x^2_size * n_channel from config_n do
10:         write features_ni data to address_i in ifmap
11:     end for
12:     send signal to start convolution and to disable write in ifmap and iwght
13:     wait end of convolution
14:     for i in 0 to convs_perline * n_filter from config_n do
15:         read features f for address_i from ofmap
16:         wait valid signal from ofmap
17:         write feature f to address_i in features_n
18:         compare received feature f with gold_ni
19:     end for
20: end for
```

Algoritmo 4.2 – Algoritmo do *test bench* do *Core Multi*.

## 4.4 CNN Simple

A arquitetura CORE1 (4.2.1) prevê a utilização de duas memórias de características, sendo uma para entrada e outra para saída, o que leva a uma duplicação da alocação de memória necessária. Observando a propriedade que as características de entrada ($ifmap_{n+1}$) da camada uma posterior ($layer_{n+1}$) coincidem com as características de saída ($ofmap_n$) da camada anterior ($layer_n$) e que esse padrão se repete nas camadas subsequentes, surge a possibilidade de projetar um módulo que se restrinja ao uso de uma única memória para as características.

Em decorrência dessa abordagem, introduziu-se a arquitetura da Rede Neural Convolucional (CNN) denominada *CNN Simple*, que executa convoluções em série (vários módulos replicados). Essa arquitetura termina em uma MEMFMAP em sua saída, local onde os resultados finais das operações são armazenados.

### 4.4.1 Memória

A fim de simplificar o processo de carregamento de dados nas IFMAPs de cada CORESR durante a execução da CNN, foram implementadas algumas modificações na

arquitetura da memória comportamental, *memory*[9]. Essas alterações foram feitas com o objetivo de permitir que a memória comportamental carregasse os dados a partir de um arquivo em disco, em vez de utilizar um pacote VHDL. Tal estratégia foi escolhida pelo mesmo motivo demonstrado em relação à geração dos dados na Seção 4.3.

### 4.4.2 Core Serial

O *Core Serial*[10] (CORESR), Figura 4.9, passou por duas principais alterações, uma delas já mencionada, remoção do MEMFMAP de saída e consequentemente a alteração em $M_{of}$ que agora multiplexa os dados para a portas de saída.

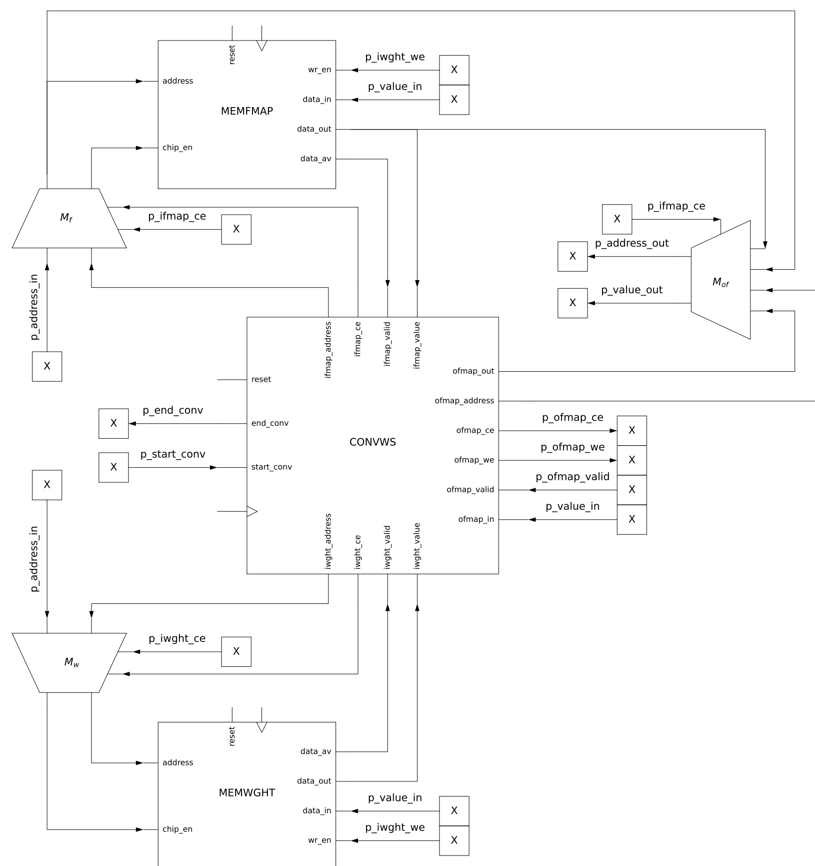



Figura 4.9 – Core Serial.

[9]Descrição VHDL:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/rtl/components/mem_file.vhd

[10]Descrição VHDL:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/rtl/core/core_serial.vhd

### 4.4.3 CNN

$M_{of}$ é controlado pelo *chip enable* do CORESR (`p_ifmap_ce`) e direciona para a porta de saída de dados do CORESR (`p_value_out`). Quando *chip enable* for alto, é direcionado o valor de saída(`data_out`) da memória MEMFMAP. Em situações alternativas, quando *chip enable* for baixo será multiplexado o valor de saída (`ofmap_out`) da CONVWS. Ou seja, quando *chip enable* está baixo, posição *default*, o CORESR está escrevendo dados de OFMAP. Quando está alto, foi solicitada leitura do MEMFMAP.

As outras portas da CONVWS, *chip enable*, habilitação de escrita, *valid* e valor de entrada estão conectadas agora diretamente as portas do CORESR.

A CNN Simple[11] (CNNSM), Figura 4.10, é uma arquitetura que conecta sequencialmente um número *n* de CORESR de tal forma que os sinais de saída de $CORESR_1$ sejam os sinais de entrada de $CORESR_{n-1}$, e que os sinal de entrada de $CORESR_n$ sejam os sinais de saída de $CORESR_{n-1}$ e assim sucessivamente. As portas se conectam em pares conforme pode ser visto na Tabela 4.1.

| | |
|---|---|
| Fim da convolução | → Início da convolução |
| Endereço de saída | → Endereço de entrada |
| MEMFMAP chip enable | → MEMFMAP chip enable |
| MEMFMAP habilitação de escrita | → MEMFMAP habilitação de escrita |

Tabela 4.1 – Tabela de conexões entre CORESR na CNNSM

Há duas exceções nesse arranjo: as portas *valid* e o primeiro (1) e último (*n*) CORESR. As portas de entrada da *CNNSM* se conectam as portas entrada do $CORESR_1$, e as portas de saída do $CORESR_n$ se conectam ao conjunto de controles da MEMFMAP e as portas de saída da *CNNSM*. As portas *valid* tem comportamento inverso das restantes, sai do *CORESR* posterior e vai para o anterior, ou seja, o sinal do $CORESR_n$ vai para $CORESR_{n-1}$ e não o contrário.

A arquitetura da CNNSM é caracterizada também pela presença de multiplexadores, portas lógicas e uma MEMFMAP. Em cada CORESR que compõe a CNNSM, existe um multiplexador responsável pelo controle da entrada de dados. O MEMFMAP está conectado ao último CORESR e às portas de saída da CNNSM. A leitura e a escrita nesta memória são reguladas por um conjunto de portas lógicas e um multiplexador.

[11] Descrição VHDL:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/rtl/cnn/simple.vhd

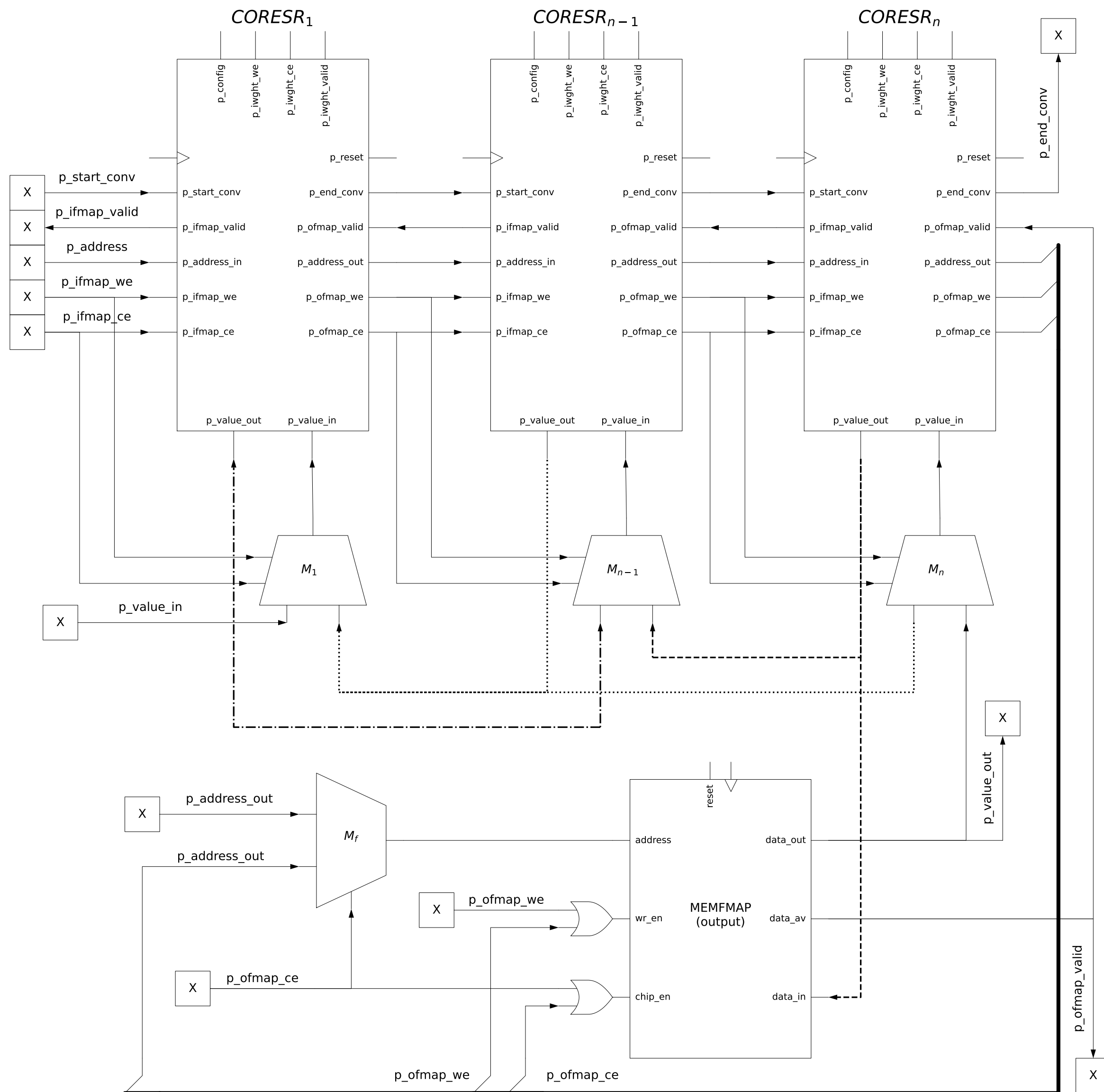



Figura 4.10 – CNN Simple11 (CNNSM).

## 4.5 Memórias BRAMs em Dispositivos FPGAs XILINX

Até esta etapa do projeto, todos os desenvolvimentos foram testados e validados em ambiente de simulação RTL (simulador Modelsim). Uma vez o projeto validado por simulação, inicia-se a etapa de síntese em dispositivos FPGA. A simulação RTL nos permite testar o conceito e a funcionalidade das nossas soluções, mas é somente ao executar as arquiteturas desenvolvidas no dispositivo no FPGA que podemos verificar seu real desempenho. Podemos então ajustar nossas soluções, otimizando-as para o melhor desempenho possível.

No entanto, o processo de portar uma descrição RTL para um dispositivo FPGA requer adaptações na descrição original. Um dos principais obstáculos que encontramos é o armazenamento dos dados de entrada, dos pesos e vieses no FPGA. FPGAs de baixa e média densidade não dispõem de registradores suficientes para essa finalidade.

No entanto, os FPGAs oferecem memórias embarcadas, denominadas de BRAMs (*Block Random Access Memory*). Essas estruturas de memória são ideais para o armazenamento dos dados da CNNSM, apresentando velocidade e eficiência satisfatórios em relação às outras formas de armazenamento, como memórias externas. Portanto, para superar o desafio do armazenamento de dados, desenvolvemos um conjunto de algoritmos que converte os dados do TensorFlow para o formato requerido para inicializar as BRAMs, permitindo assim que estas sejam utilizadas para a prototipação no dispositivo FPGA.

Nesta etapa, é importante compreender não apenas o processo de conversão e armazenamento dos dados, mas também a arquitetura da memória e como a conexão com as BRAMs é estabelecida. A discussão sobre esses pontos é necessária, pois eles são necessárias para a execução do nosso sistema. Deste modo, a essa seção é dedicada à explicação das decisões de projeto, à exploração do desenvolvimento e aplicação desse código de conversão para BRAM, da arquitetura de memória implementada e da conexão com as BRAMs. Além disso, também discutimos os benefícios e melhorias que essas técnicas proporcionam ao nosso projeto.

### 4.5.1 Instanciação e Implementação de BRAMs

Nesta subseção, detalhamos os métodos para instanciar e implementar os blocos de memória BRAMs em nosso projeto.

#### Inferência de BRAMs

A **inferência automática** de BRAMs pela ferramenta de síntese da Xilinx proporciona otimização de recursos, flexibilidade, portabilidade e simplicidade de desenvolvimento. Otimiza recursos através de uma alocação eficiente de blocos de memória no FPGA. A flexibilidade é evidenciada na possibilidade de criar memórias de tamanhos e configurações variadas. A portabilidade é garantida pelo mapeamento automático da ferramenta de síntese das BRAMs inferidas para os recursos de memória do dispositivo. Por fim, a simplicidade resulta do uso de abstrações de alto nível em linguagens de descrição de hardware, como VHDL e Verilog.

No entanto, a inferência automática de BRAMs apresenta desvantagens. Primeiramente, o controle do projetista sobre a implementação específica das memórias pode ser limitado, afetando a escolha do tipo de BRAM, a organização da memória e a otimização

manual de desempenho e consumo de energia. Além disso, variações no desempenho e na utilização de recursos podem ocorrer, dependendo das heurísticas e algoritmos utilizados pela ferramenta de síntese. A depuração e análise de problemas também se tornam mais complexas, uma vez que o mapeamento das BRAMs inferidas para os recursos de hardware é gerenciado pela ferramenta de síntese e pode não ser diretamente observável pelo projetista e nem ser utilizada na simulação. Por último, embora a inferência de BRAMs melhore a portabilidade entre dispositivos do mesmo fabricante, pode haver problemas de compatibilidade ao se trabalhar com FPGAs de diferentes fabricantes, exigindo adaptações ou refatorações na descrição HDL.

Gerador de Memórias

Uma opção à inferência automática é utilizar um **software gerador de memórias** na ferramenta de síntese da Xilinx [16] – *Block Memory Generator v8.4*, o qual também apresenta diversas vantagens. Primeiramente, o controle sobre a implementação das memórias é possível, incluindo a escolha do tipo de BRAM, a organização da memória e a otimização manual de desempenho. Essa abordagem permite maior eficiência e desempenho no projeto, uma vez que os projetistas podem criar memórias personalizadas que atendem às necessidades específicas de cada aplicação. Ademais, a depuração e análise de problemas tornam-se mais simplificadas, devido à maior visibilidade e controle sobre a implementação das BRAMs e seus respectivos recursos de hardware.

No entanto, algumas desvantagens estão associadas ao uso de um software gerador de memórias. A complexidade do processo de desenvolvimento pode ser aumentada, exigindo maior conhecimento e compreensão das especificidades da implementação das memórias e do software. Além disso, o tempo de desenvolvimento é estendido, uma vez que os projetistas precisam configurar e ajustar o gerador para cada CNN especificamente. A portabilidade entre dispositivos e famílias de FPGAs pode ser reduzida, já que as memórias geradas são específicas para uma plataforma ou fabricante. E por fim da mesma forma que memória inferida automaticamente não é possível usar esse recurso no simulador.

Instanciar BRAMs com descrição RTL

Uma terceira opção é a **instanciação manual** das BRAMs diretamente com descrição RTL. Comparada à inferência ou ao uso de geradores, oferece controle preciso sobre a implementação das memórias, permitindo explicitar a organização da memória, parâmetros e configurações desejadas. Também propicia maior controle sobre a utilização dos recursos de memória e otimização do desempenho, resultando em desempenho mais previsível que a inferência automática. Permite, ainda, a utilização em simuladores, facilitando a depuração e correção de erros, e promovendo entendimento sobre o comportamento das memórias durante o desenvolvimento.

Entretanto, tal abordagem acarreta desafios. Aumenta a complexidade do projeto, dificulta o desenvolvimento, limita a flexibilidade e a reutilização da descrição, além de demandar maior tempo devido à necessidade de configuração e ajuste manual das memórias.

Escolheu-se neste projeto a instanciação manual de BRAMs, apesar dos desafios. O uso de BRAMs em simulações, facilitando a depuração e correção de erros, foi fator relevante na decisão.

Os dados armazenados nas BRAMS, como os mapas de características *feature map* os pesos (*weights*) e viés (*bias*), sofrem alterações constantes. Essas mudanças são ocasionadas por diferentes arquiteturas de CNN e novos treinamentos realizados na mesma arquitetura. Adicionalmente, as CNNs demandam o uso de centenas de blocos de memória. Dessa forma, a instanciação manual das BRAMs torna-se inviável, dada a complexidade e a dinamicidade envolvidas no gerenciamento das memórias para esse tipo de aplicação.

A **solução** adotada para realizar a instanciação direta de BRAMs na descrição RTL foi a realização de um software para gerar as descrições VHDL com a instanciação das BRAMS de acordo com os parâmetros da CNN. Este software é uma contribuição significativa para o grupo de pesquisa, uma vez que os membros do grupo passam a contar com um software desenvolvido especificamente para geração de descrição para instanciar BRAMs em VHDL a partir de valores de entrada. Essa ferramenta permite uma maior flexibilidade na configuração das memórias, possibilitando a adaptação do projeto para atender às necessidades específicas da pesquisa em questão. Desse modo, a instanciação das BRAMs com descrição HDL, apesar de seus desafios, foi considerada a melhor abordagem para o contexto deste trabalho.

Formas para Instanciar Memórias BRAM

A Xilinx provê dois métodos para a instanciação direta de BRAMs: macros ou primitivas. Cada uma dessas alternativas possui vantagens e desvantagens, as quais são discutidas a seguir.

A utilização de macros na instanciação de BRAMs possui como vantagem a reutilização da descrição. As macros podem ser reutilizadas em projetos que usam diferentes FPGAs do mesmo fabricante, o que torna a descrição mais concisa e fácil de manter. Além disso, as macros permitem a abstração de detalhes de implementação das BRAMs, proporcionando maior clareza e facilitando a leitura da descrição. No entanto, a depuração de problemas em um projeto que utiliza macros pode ser mais desafiadora, pois as macros podem ocultar erros ou gerar comportamentos inesperados.

A instanciação das BRAMs pela primitiva possui como vantagen a explicitação da implementação, que facilita a compreensão e a leitura da descrição. Além disso, a depuração de problemas pode ser mais fácil com a instanciação pelo nome do bloco, já que os erros e os comportamentos inesperados são mais facilmente identificáveis na descrição.

Por outro lado, a instanciação das BRAMs pela primitiva pode levar a uma maior complexidade na descrição, tornando o desenvolvimento e a manutenção mais desafiadores. Além disso, o uso do nome do bloco implica em menor reutilização da descrição.

A documentação da Xilinx [14] provê modelos para a instanciação de macros, como a `BRAM_SINGLE_MACRO`, cujo símbolo é apresentado na Figura 4.11. Os dispositivos FPGA contêm vários blocos de memória RAM que podem ser configurados como memórias RAM/ROM de uso geral de 36Kb ou 18Kb.

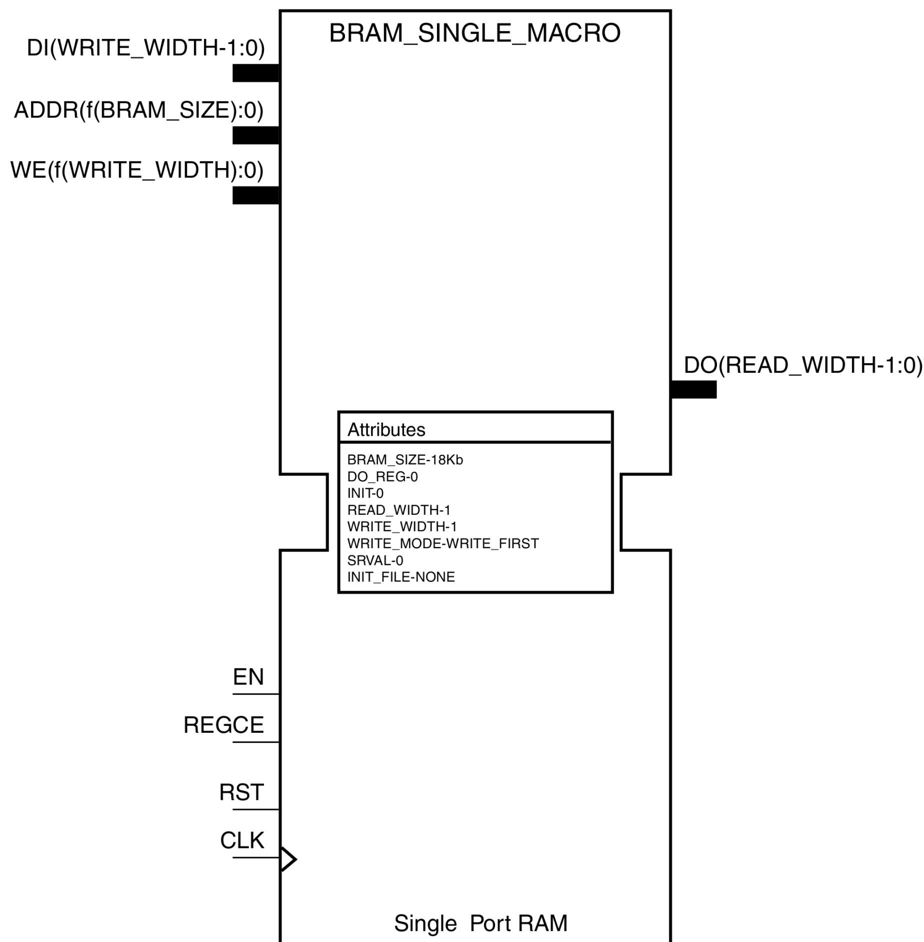



Figura 4.11 – Macro BRAM_SINGLE_MACRO.

**Optou-se** pela instanciação de BRAMs por meio de macros em detrimento da abordagem baseada em primitivas, tendo em vista a simplicidade e a rapidez proporcionadas por essa técnica. A descrição 4.1 apresenta esta forma de instanciar BRAMs[12]. A instanciação por meio de primitivas apresenta maior complexidade e menor reutilização de descrição, fatores que podem impactar negativamente o processo de desenvolvimento deste trabalho.

[12]Descrição VHDL:
https://raw.githubusercontent.com/tarsioonofrio/acc_dse_env/484fa709cfe5597e464643d2bc0e20e2b14412e6/apps/rtl_output/default/default/bram/bram_36Kb.vhd

Código 4.1 – Instanciação de BRAM com macros.

```
1   entity bram_single is
2       generic (  DEVICE     : string := "7SERIES";
3                  BRAM_NAME  : string := "default"
4               );
5       port ( RST, CLK, EN, WE  : in std_logic;
6              DI   : in std_logic_vector(36-1 downto 0);
7              ADDR : in std_logic_vector(9-1 downto 0);
8              DO   : out std_logic_vector(36-1 downto 0)
9             );
10   end bram_single;
11
12    architecture a1 of bram_single is
13      signal bram_wr_en     : std_logic_vector(8-1 downto 0);
14      signal bram_addr      : std_logic_vector(9-1 downto 0);
15      signal bram_di        : std_logic_vector(44-1 downto 0);
16      signal bram_do        : std_logic_vector(44-1 downto 0);
17      constant bram_par     : std_logic_vector(8-1 downto 0) := "00000000";
18
19      begin
20         bram_wr_en <= (others => '1') when WE = '1' else (others => '0');
21         bram_addr <= ADDR(9-1 downto 0);
22         bram_di <= bram_par & DI;
23         DO <= bram_do(36-1 downto 0);
24
25        MEM_IWGHT_LAYER0_INSTANCE0 : if BRAM_NAME = "iwght_layer0_instance0" generate
26          BRAM_SINGLE_MACRO_inst : BRAM_SINGLE_MACRO
27          generic map (
28              BRAM_SIZE => "36Kb",             -- Target BRAM, "18Kb" or "36Kb"
29              DEVICE => DEVICE,                -- Target Device: "VIRTEX5", "7SERIES", "VIRTEX6, "SPARTAN6"
30              DO_REG => 0,                     -- Optional output register (0 or 1)
31              INIT => X"000000000000000000",   -- Initial values on output port
32              INIT_FILE => "NONE",
33              WRITE_WIDTH => 44, -- 0, -- Valid values are 1-72 (37-72 only valid when BRAM_SIZE="36Kb")
34              READ_WIDTH => 44, -- 0,  -- Valid values are 1-72 (37-72 only valid when BRAM_SIZE="36Kb")
35              SRVAL => X"000000000000000000",  -- Set/Reset value for port output
36              WRITE_MODE => "WRITE_FIRST",     -- "WRITE_FIRST", "READ_FIRST" or "NO_CHANGE"
37              -- The following INIT_xx declarations specify the initial contents of the RAM
38              INIT_00 => X"0000894000000000ffffa57fffffffff000004fe00000000ffff05bffffffff",
39              INIT_01 => X"000186900000000000112a0000000000008a4000000000000001d00000000",
40              INIT_02 => X"000013d200000000ffffebafffffffffffdeecffffffffffffcf0ffffffff",
41              ....
42              INIT_7F => X"0000000000000000000000000000000000000000000000000000000000000000",
43
44              -- The next set of INITP_xx are for the parity bits
45              INITP_00 => X"0000000000000000000000000000000000000000000000000000000000000000",
46              INITP_01 => X"0000000000000000000000000000000000000000000000000000000000000000",
47              ...
48              INITP_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
49              INITP_0F => X"0000000000000000000000000000000000000000000000000000000000000000"
50          )
51          port map (
52              DO => bram_do,       -- Output data, width defined by READ_WIDTH parameter
53              ADDR => bram_addr,  -- Input address, width defined by read/write port depth
54              CLK => CLK,     -- 1-bit input clock
55              DI => bram_di,       -- Input data port, width defined by WRITE_WIDTH parameter
56              EN => EN,       -- 1-bit input RAM enable
57              REGCE => '1', -- 1-bit input output register enable
58              RST => RST,     -- 1-bit input reset
59              WE => bram_wr_en        -- Input write enable, width defined by write port depth
60          );
61      end generate MEM_IWGHT_LAYER0_INSTANCE0;
62  ...
63  end a1;
```

### 4.5.2 Arquitetura das memórias BRAMs

Esta seção aborda a análise da arquitetura de memórias BRAMs, focando na compreensão dos bits de paridade. Os bits de paridade compreendem parte dos bits de dados armazenados nos blocos de memória BRAM. Identificou-se que durante a leitura de dados

de uma BRAM ocorre a transmissão conjunta de bytes de paridade e dados na mesma porta de saída, uma característica não claramente descrita na documentação quando se emprega macros.

A Tabela 4.2[13] apresenta esta análise para a macro `BRAM_SINGLE_MACRO` (linha 28 da descrição 4.1), configurada para 18 Kb. A coluna `Memory` corresponde à largura do barramento de dados sem os bits de paridade. As colunas relativas à `BRAM` correspondem à configuração dos blocos de memória. A coluna `address` corresponde à largura do barramento de endereços, o que resulta em uma memória com `depth` posições endereçáveis. A coluna `we` corresponde à largura do barramento para habilitar a escrita. A coluna `parity` ao número de bits presentes no barramento de dados referente aos bits de paridade. Por exemplo, caso a largura do barramento de dados seja entre 17 e 32 bits, podemos ter no barramento de dados ente 19 bits (17 bits e 2 de paridade) a 36 (32 bits e 4 de paridade). A Tabela 4.3 apresenta análise similar para memória de 36 Kb.

Tabela 4.2 – Configurações para BRAM 18Kb.

| Memory | BRAM | | | | |
|---|---|---|---|---|---|
| word width (bits) | word width (bits) | depth | address | we | parity |
| 17–32 | 19–36 | 512 | 9 | 4 | 4 |
| 9–16 | 10–18 | 1024 | 10 | 2 | 2 |
| 5–8 | 5–9 | 2048 | 11 | 1 | 1 |
| 3–4 | 3–4 | 4096 | 12 | 1 | 0 |
| 2 | 2 | 8192 | 13 | 1 | 0 |
| 1 | 1 | 16384 | 14 | 1 | 0 |

Tabela 4.3 – Configurações para BRAM 36Kb.

| Memory | BRAM | | | | |
|---|---|---|---|---|---|
| word width (bits) | word width (bits) | depth | address | we | parity |
| 33–64 | 37–72 | 512 | 9 | 8 | 8 |
| 16–32 | 19–36 | 1024 | 10 | 4 | 4 |
| 9–16 | 10–18 | 2048 | 11 | 2 | 2 |
| 5–8 | 5–9 | 4096 | 12 | 1 | 1 |
| 3–4 | 3–4 | 8192 | 13 | 1 | 0 |
| 2 | 2 | 16384 | 14 | 1 | 0 |
| 1 | 1 | 32768 | 15 | 1 | 0 |

A análise sugere que, para o limite superior, os valores são maximizados: a largura total é de 72 bits, com 64 bits dedicados a dados e 8 a paridade. Em contraste, para o limite inferior, os valores são minimizados: a largura total é de 1 bit, com 1 bit dedicado a dados e nenhum a paridade. Para uma largura total de 2 bits e de 3 a 4 bits, acredita-se que o

[13] Ambas tabelas foram encontradas na página 30 de [15]

bit de paridade é nulo. Para o intervalo subsequente, de 5 a 9 bits, a análise sugere que há um bit adicional disponível para paridade, devido ao valor máximo possível. No intervalo remanescente de 10 a 18 bits, com base no valor máximo de 18, deduz-se que há 2 bits disponíveis para paridade. No último intervalo não definido, de 19 a 36 bits, estima-se que há 4 bits disponíveis para paridade.

## Arquitetura de Encapsulamento das BRAMs

A arquitetura da entidade que encapsula as BRAMs, `bram_single` (Descrição 4.1)[14] é apresentada na Figura 4.12. Este módulo controla a instanciação de cada macro e foi projetado com o objetivo de encapsular os blocos de memória e sinais de tal forma que a arquitetura externa precise receber o mínimo possível de alterações, dependendo da configuração das BRAMs, como largura de dados e endereços ou tipo de memória, se 18 Kb ou 36 Kb.

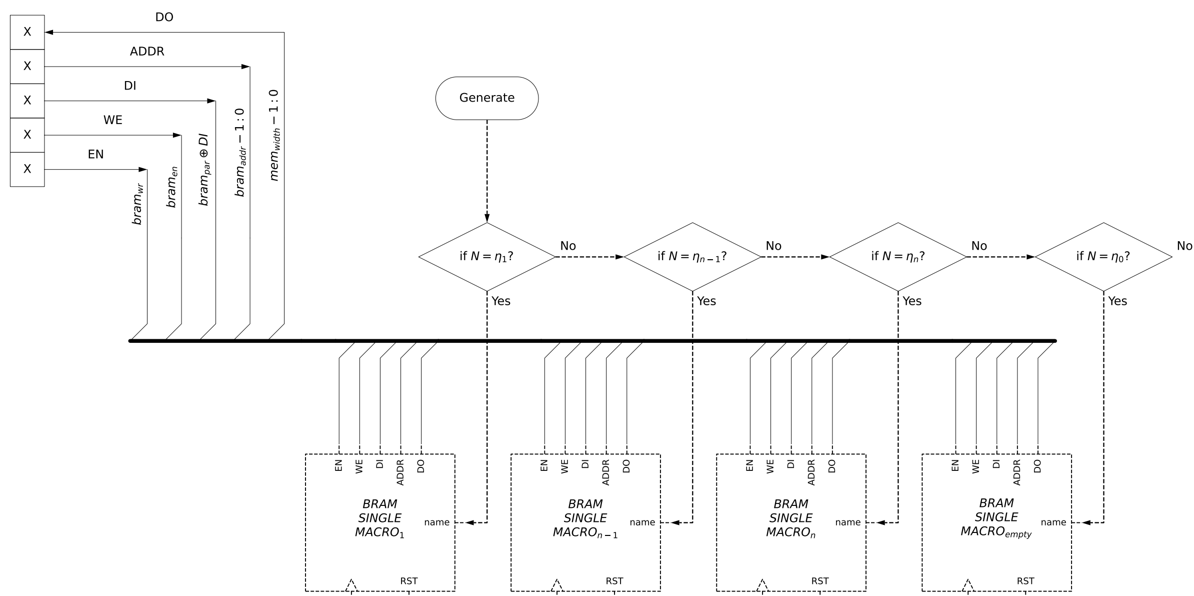



Figura 4.12 – Arquitetura de encapsulamento das BRAMs - módulo `bram_single`.

Este módulo faz operações para aumentar ou reduzir a largura de bits dos sinais de entrada ou saída:

- Os barramentos de dados de entrada e saída da BRAM tem uma largura de bits equivalente à paridade mais dados (44 bits, sinais `bram_di` e `bram_do`, Descrição 4.1). Dado que não estamos utilizando os bits de paridade no projeto, os bits escritos na

[14]Descrição VHDL:
https://raw.githubusercontent.com/tarsioonofrio/acc_dse_env/484fa709cfe5597e464643d2bc0e20e2b14412e6/apps/rtl_output/default/default/bram/bram_36Kb.vhd

memória são concatenados com 8 bits em zero, e os dados de saída excluem os bits de paridade (linha 25).

- O sinal de escrita, `WE` possui 1 bit, sendo extendido para 4 bits, dado que temos 4 bits que controlam a escrita na BRAM.
- O sinal de habilitação de leitura, `EN`, não sofre nenhuma alteração.

A instanciação das macros é controlada por um operador que compara o nome a ser instanciado, $N$, com o nome da $BRAM_N$, $\eta_n$ que é único. Caso a comparação entre $N$ e $\eta_1$ sejam iguais a macro é instanciada, caso seja falso compara-se a próxima, $\eta_2$ e assim sucessivamente até se completarem todas $\eta_n$. Caso todas as comparações sejam falsas deverá ser instanciada a $BRAM_{empty}$ como todos os dados inicializados em zero.

O processo de geração de cada instância foi planejado como um operador ternário. Entretanto, dada as limitações da versão de VHDL usado nesse trabalho, o processo foi implementado como um operador de apenas dois argumentos. Devido a isso, a $BRAM_{empty}$ também só será instanciada caso $N$ seja $\eta_0$, ou seja, o nome da BRAM seja o nome definido da BRAM vazia.

Duas observações **importantes** sobre o módulo `bram_single` (Descrição 4.1):

- a BRAM "vazia" é o bloco de memória que recebe os dados de processamento de uma determinada camada, os quais são utilizados como memória de características (MEMFMAP) dento de cada CORESR;
- o módulo `bram_single` é gerado de forma automática, contendo todas as instanciações necessárias de módulos BRAM, porém este módulo gera apenas uma BRAM. Logo, este módulo é instanciado tantas vezes quanto necessárias, como será apresentado na seção seguinte.

### Arquitetura de Controle de Acesso as BRAMs

Em geral, uma BRAM não é capaz de armazenar todos os dados necessários para o modelo de uma CNN, sendo necessárias várias instâncias para comportar todos os dados. Assim é necessária uma arquitetura de controle de acesso e instanciação das $n$ `bram_single`, módulo é denominado `memory`, conforme apresentado na Figura 4.13. A descrição 4.2[15] apresenta o controle de habilitação de cada bloco de memória (linhas 1-4), e a instanciação dos módulos `bram_single`, com o nome do módulo a ser instanciado (linha 9). O módulo `memory` foi projetado de tal forma que a escrita e a leitura paralelas não sejam

[15]Descrição VHDL:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/rtl/components/mem_bram.vhd

suportadas, impondo assim restrições à execução de operações em mais de uma BRAM simultaneamente.

Código 4.2 – Instanciação dos blocos de memória no módulo `memory`.

```
LOOP_EN : for i in 0 to FBRAM_NUM -1 generate
   bram_chip_en(i) <= chip_en when i = bram_select else '0';
   bram_wr_en(i) <= wr_en when i = bram_select else '0';
 end generate;

 LOOP_MEM : for i in 0 to FBRAM_NUM -1 generate
     BRAM_SINGLE_INST: entity work.bram_single
     generic map (
       BRAM_NAME => BRAM_NAME & "_instance" & integer'image(i)
     )
     port map(
       CLK  => nclock,
       RST  => reset,
       EN   => bram_chip_en(i),
       WE   => bram_wr_en(i),
       DI   => data_in,
       ADDR => address(BRAM_ADDR-1 downto 0),
       DO   => bram_data_out(i)
     );
   end generate;
```

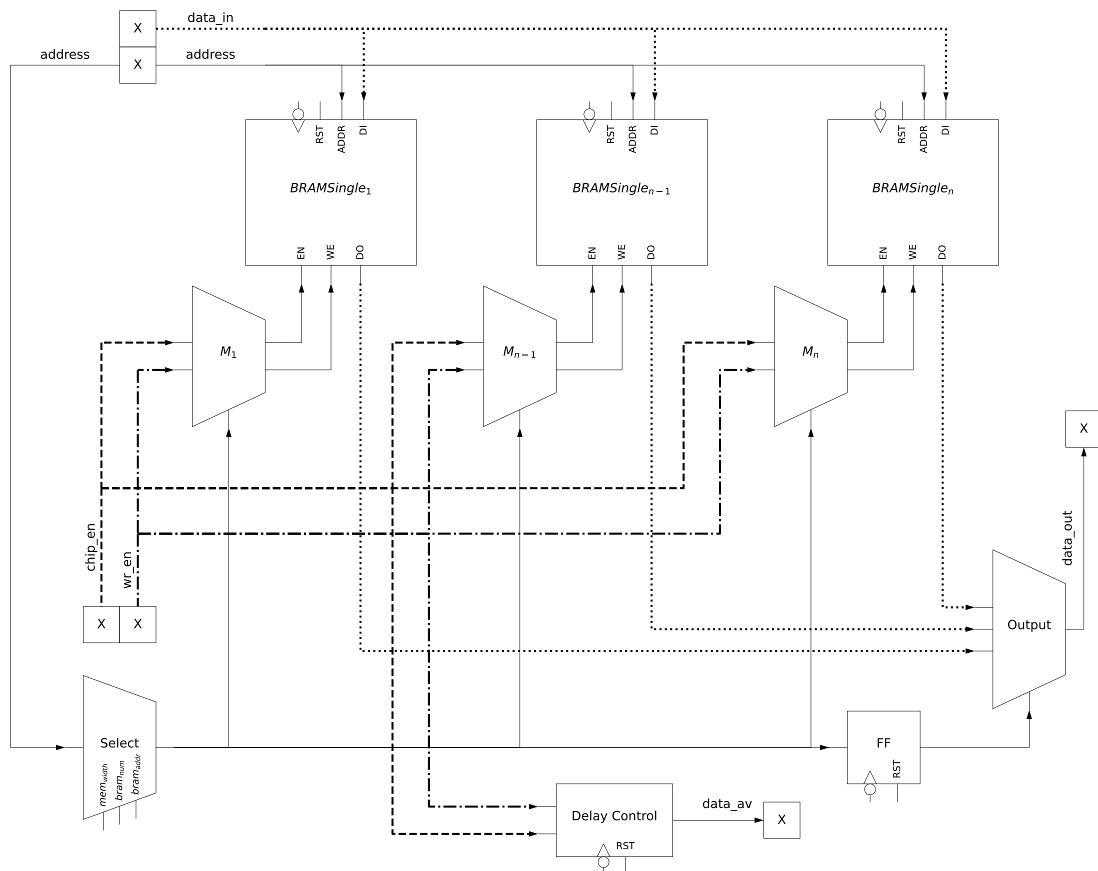



Figura 4.13 – Arquitetura de acesso ao encapsulamento das BRAMs.

Este módulo possui um multiplexador, denominado *Select*, que habilita para cada operação uma única BRAM por vez, efetuando a seleção entre as *n* BRAMs disponíveis. O método de seleção envolve a extração de uma partição do endereço (*address*) que varia entre a largura máxima, definida por $mem_{width}$, e a largura do endereço da BRAM, definida por $bram_{addr}$. Esta segmentação do endereço resulta na obtenção do endereço da `bram_single` selecionada para a operação atual conforme pode ser observado na Figura 4.14.

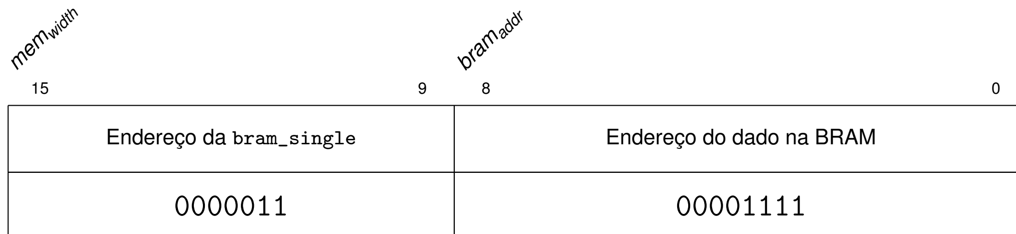



Figura 4.14 – Particionamento do barramento `address` para seleção de módulo `bram_single` e geração do endereço interno da BRAM.

A saída do multiplexador *Select* é enviado para um registrador e, em seguida, para o multiplexador que controla a saída de dados, denominado *Output*. Este multiplexador recebe o sinal de dados de saída de todas as BRAM, sendo a seleção feita pelo valor armazenado no registrador. A saída deste multiplexador é o barramento *data_out*.

As BRAMs da Xilinx usadas nesse trabalho requerem que o endereço fornecido esteja estável na borda de subida, conforme ilustrado nas Figuras 4.15 e 4.16. Entretanto, como fornecemos o endereço também na borda de subida, este endereço de fato só estará disponível na borda de subida seguinte. Com isto, obtemos uma latência de 2 ciclos para leitura, e no projeto do CONVWS estamos assumindo 1 ciclo de clock para esta latência. Este comportamento desincroniza a leitura dos dados.

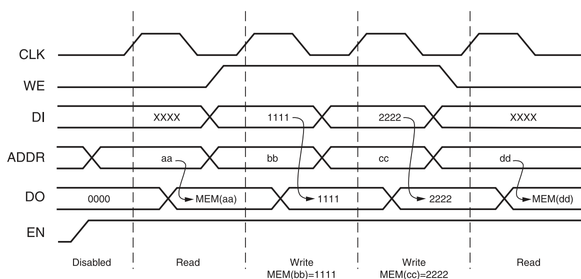



Figura 4.15 – Forma de onda da BRAM em modo `WRITE_FIRST` [15].

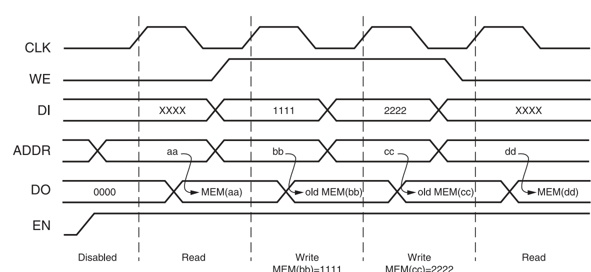



Figura 4.16 – Forma de onda da BRAM em modo `READ_FIRST` [15].

Para gerir tal situação, identificamos três estratégias, cada uma com seus próprios méritos e desvantagens. A primeira envolve aumentar o tempo de latência para dois ciclos, acomodando assim o atraso criado pelo comportamento das BRAMs e as características de leitura dos módulos desenvolvidos. Embora simples, basta alterar a parametrização, esta opção resultaria em um aumento de 100% no tempo de leitura.

A segunda opção propõe a modificação do CONVWS e outros módulos desenvolvidos no projeto para que suas operações de leitura ocorram na borda de subida. Essa seria a solução recomendada, pois usaria apenas uma fase de clock, evitando problema de sincronização. No entanto, a complexidade desta tarefa e o tempo disponível para o desenvolvimento deste trabalho tornam sua execução inviável.

A última alternativa é a implementação de dois clocks com fases deslocadas em 180°. Embora seja a solução simples, basta negar o clock e mapear o sinal resultante em `bram_single`, e possa ser implementada no curto prazo disponível para este trabalho, pode haver possíveis problemas de sincronização, como mencionado anteriormente. Devido à sua simplicidade e à rapidez com que pode ser implementada, optamos por essa solução.

É necessário registrar o sinal de *Select* pois pode ocorrer troca de BRAM, e acontecer leitura de módulo BRAM incorreto. Esse erro pode ser observado na Figura 4.17, no ciclo 4, marcado com uma linha vermelha comparando o *data_output*, sem registrador, com o sinal $data_output_{FF}$. Imediatamente antes da linha vermelha, *data_output* estava com os dados ($d1_1$) corretos no barramento. Entretanto, devido a troca de `bram_single`, o dado não fica disponível quando o clock sobe, momento que seria amostrado, ficando disponível $d2_0$. Esse erro não se observa em $data_output_{FF}$ dado que o registrador armazena o dado com o atraso de meio ciclo de clock (na borda de descida).

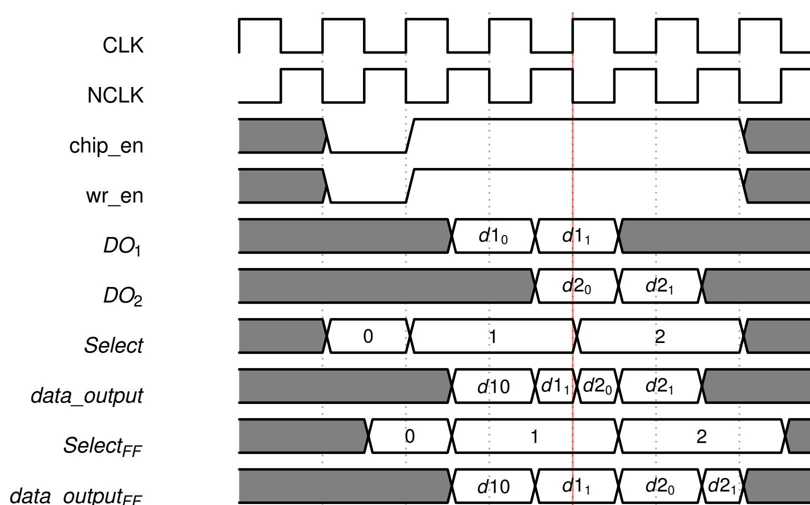



Figura 4.17 – Comportamento do módulo `memory` sem registrar sinais (*Select* e *data_output*) e registrando ($Select_{FF}$ e $data_output_{FF}$), usando como base a saída de duas `bram_single`, $DO_1$ e $DO_2$.

Os dados de entrada (*data_in*) e endereço (*address*) são enviados para os *n* módulos `bram_single`, através de barramentos. O controle de atraso (*Delay Control*) tem dupla função. A primeira é sincronizar a leitura de um dado com o módulo que requisitou o mesmo. A segunda função é permitir simular a latência da memória. Como neste projeto não necessitamos simular a latência da memória, o *Delay Control* apenas realiza a sincronização de leitura.

### 4.5.3 Algoritmo de Geração da Inicialização das BRAMs

Esta seção apresenta os algoritmos[16] que possibilitam a geração da inicialização das BRAMs conforme os parâmetros da rede CNN. Antes de explicarmos os algoritmos desenvolvidos, é necessário discutir como os dados são tratados previamente à inicialização das BRAMs: a transformação dos dados para representação em complemento de dois e a reordenação dos bytes.

Conforme discutido no capítulo anterior, as BRAMs requerem a inicialização com dados em formato hexadecimal, ou seja, não suportam o armazenamento de dados na representação decimal, conforme era utilizado no modelo de memória simulado, na qual os dados eram armazenados em um vetor de inteiros. A Equação 4.1 realiza a conversão de um número inteiro (positivo ou negativo) para seu equivalente em complemento de dois para representação em hexadecimal. Esta operação garante que o resultado esteja dentro do intervalo de um número representável com $n$ bits.

$$(x + 2^n) \mod (2^n) \tag{4.1}$$

Da mesma forma que tivemos dificuldades com o uso dos bits de paridade na palavra de dados da BRAM, a documentação é também omissa na forma como os bytes são ordenados. Para ilustrar esse problema, considere o seguinte exemplo: se o valor $6285_{10}$ ou $00000000188D_{16}$ (48 bits) for armazenado como tal em uma BRAM, quando este valor for recuperado da memória, ele será interpretado como $26993869455360_{10}$ ou $188D00000000_{16}$. Sendo assim, caso os inteiros em representação hexadecimal, tendo o décimo segundo bit como o mais significativo, seja representado como um vetor $H$ com $h$ elementos, teremos:

$$H = [h_{12}, h_{11}, h_{10}, h_9, h_8, h_7, h_6, h_5, h_4, h_3, h_2, h_1] = [h_{12}, ..., h_1] \tag{4.2}$$

E quando a leitura é executada temos:

$$H' = [h_4, h_3, h_2, h_1, h_{12}, h_{11}, h_{10}, h_9, h_8, h_7, h_6, h_5] = [h_4, ..., h_1, h_{12}, ..., h_5] \tag{4.3}$$

Portanto, a função de permutação $\pi$ que realiza essa transformação pode ser representada por:

[16] Código:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/apps/bram-generate.py

$$\pi = \{12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1\} \rightarrow \{4, 3, 2, 1, 12, 11, 10, 9, 8, 7, 6, 5\} \quad (4.4)$$

Assim, a Equação 4.4 representa uma operação de permutação aplicada a uma sequência ordenada de 12 nibbles (48 bits). Nesta permutação $\pi$ os quatro nibbles menos significativos (4, 3, 2, 1) são movidos para o início da sequência, e os oito nibbles restantes (12, 11, 10, 9, 8, 7, 6, 5) são deslocados para as posições menos significativas. Se estivéssemos utilizando 64 bits e não 48, esta operação corresponderia a inverter blocos de 32 bits.

A Figura 4.18 ilustra os dados antes e após a realização das transformações apresentadas anteriormente. Este processo é detalhado no Algoritmo 4.3. Este algoritmo segue uma sequência de passos para cada valor inteiro, inserindo-os na lista *list_int_hex*. Inicialmente, aplica a Equação 4.1 para a conversão do valor inteiro em seu correspondente em complemento de dois; logo após, a representação resultante é formatada como uma *string* hexadecimal. Subsequentemente, esta *string* é submetida a uma permutação conforme estabelecido pela Equação 4.4, que a reordena. No final do algoritmo, a *list_int_hex* contém as representações hexadecimal para todos os inteiros.

| | Integer (15 … 0) | Two's complement (15 … 0) | Hexadecimal (15 … 0) | Ordinated (15 … 0) |
|---|---|---|---|---|
| 0 | -4005 | 18446744073709547611 | fffffffffffff05b | fffff05bffffffff |
| 1 | 1278 | 1278 | 00000000000004fe | 000004fe00000000 |
| | ⋮ | ⋮ | ⋮ | ⋮ |
| N | -52 | 18446744073709551564 | ffffffffffffffcc | ffffffccffffffff |

Figura 4.18 – Listas mostrando a transformação dos dados passando pelas três operações: complemento de dois, hexadecimal e ordenação.

1: **start** an empty list called *list_int_hex*
2: **for** each *int* in *list_int* **do**
3: **call** Equação 4.1 (*two′scomplement*) in *int*
4: **format** *two′scomplement* as *hexadecimal*
5: **call** Equação 4.4 (*permutation*) in *hexadecimal*
6: **add** the *ordering* to *list_int_hex*
7: **end for**
8: **return** *list_int_hex*

Algoritmo 4.3 – Conversão de inteiros para formato de inicialização das BRAMs.

Na sequência, o algoritmo 4.4 apresenta o procedimento de agrupamento dos valores hexadecimais gerados pelo Algoritmo 4.3. Os valores para a inicialização das BRAMs correspondem aos valores *init_xx*, ou seja, $init_{01}$, $init_{02}$, $init_{n-1}$, $init_n$.

1: **start** an empty list called *list_init_xx*
2: **for** $i \leftarrow 0$ **to** length of *list_int_hex* **step** *values_init_xx* **do**
3:     **start** an empty list called *init_xx*
4:     **for** $j \leftarrow 0$ **to** *values_init_xx* **do**
5:         **get** *data* from $list_int_hex_{i+j}$
6:         **add** *data* to *init_xx*
7:     **end for**
8:     **reverse** *init_xx*
9:     **add** to *list_init_xx*
10: **end for**
11: **return** *list_init_xx*

Algoritmo 4.4 – Agrupamento dos hexadecimais por *init_xx*

Os dados são agrupados em sublistas de tamanho *values_init_xx*. Da mesma forma que as palavras de 64 bits necessitam de permutação de 32 bits, os blocos de inicialização possuem 256 bits (64 nibbles), também requerendo permutação de blocos de 32 bits (16 nibbles). A Figura 4.19 apresenta este processo de geração dos valores inicialização das BRAMs. A parte superior da Figura apresenta os valores obtidos pelo laço 4-7 do Algoritmo 4.4. A parte inferior da Figura apresenta os dados após a permutação (linha 8). A Figura 4.20 apresenta como os dados são inseridos no *list_init_xx*.

| 63 | 47 | 31 | 15 — 0 | |
|---|---|---|---|---|
| fffffffffffff05b | 00000000000004fe | 00000000000004fe | 0000000000000894 | $init_{00}$ |
| 000000000000001d | 00000000000008a4 | 000000000000112a | 0000000000001869 | $init_{01}$ |

| 63 | 47 | 31 | 15 — 0 | |
|---|---|---|---|---|
| 0000000000000894 | 00000000000004fe | 00000000000004fe | fffffffffffff05b | $init_{00}$ |
| 0000000000001869 | 0000000000001869 | 00000000000008a4 | 000000000000001d | $init_{01}$ |

Figura 4.19 – Geração dos valores utilizados para inicialização das BRAMs.

| 63 — 47 — 31 — 15 — 0 | |
|---|---|
| 000008940000000fffffa57ffffffff000004fe00000000fffff05bffffffff | $init_{00}$ |
| 0000186900000000000112a00000000000008a4000000000000001d00000000 | $init_{01}$ |

Figura 4.20 – Valores utilizados para inicialização das BRAMs, com largura de 255 bits (64 valores hexadecimais).

O Algoritmo 4.5 apresenta o procedimento para o agrupamento das listas de dados *init_xx* para cada instância de BRAM. Os dados são organizados em sublistas com tamanho

definido por *values_bram*, que correspondem a 64 valores *init_xx* para RAMB18E1 ($64 \times$ 256 bits = 16384 bits) , ou 128 *init_xx* para RAMB36E1 ($128 \times 256$ bits = 32768 bits).

1: **start** an empty list called *list_bram_init*
2: **for** $i \leftarrow 0$ **to** length of *list_init_xx* **step** *values_bram* **do**
3: **start** an empty list called *bram_init_xx*
4: **for** $j \leftarrow 0$ **to** *values_bram* **do**
5: **get** *init_xx* from $list_init_xx_{i+j}$
6: **add** *init_xx* to *bram_init_xx*
7: **end for**
8: **add** *bram_init_xx* to *list_bram_init*
9: **end for**
10: **return** *list_bram_init*

Algoritmo 4.5 – Agrupamento da lista de *init_xx* por instância de BRAM

O Algoritmo 4.6 descreve o processo de geração de instanciação de cada BRAM. Inicialmente, o algoritmo realiza a soma da largura dos dados da BRAM, $bram_{width}$, e da largura da paridade $bram_{par}$, resultando em $bram_{width+par}$ (linha 2). Para cada elemento da lista *list_bram_init*, o algoritmo concatena o nome (tipo e número da camada) com a instância da BRAM, o índice *i* do loop. O resultado desta operação é atribuído à variável *name*. O algoritmo então recupera os dados *bram_init_xx* da lista *list_bram_init* no índice *i*. Se houver espaços vazios nos dados, estes são preenchidos com zeros.

Após a recuperação e o preenchimento dos dados, o algoritmo formata a instanciação da BRAM, *bram_instantiation*, com o nome, os dados *bram_init_xx* e o parâmetro $bram_{width+par}$. O algoritmo retorna a lista contendo todas as instanciações formatadas das BRAMs.

1: **start** an empty list called *list_bram_instantiation*
2: **sum** $bram_{width}$ and $bram_{par}$ to $bram_{width+par}$
3: **for** $i \leftarrow 0$ **to** length of *list_bram_init* **do**
4: **concatenate** *name* of bram and *i*.
5: **get** *bram_init_xx* from $list_bram_init_i$
6: **fill** the remaining empty data in *bram_init_xx* with zeros
7: **format** *bram_instantiation* with *name*, *bram_init_xx* and $bram_{width,par}$
8: **add** *bram_instantiation* to *list_bram_instantiation*
9: **end for**
10: **return** *list_bram_instantiation*

Algoritmo 4.6 – Geração de instanciação de cada BRAM

O Algoritmo 4.7 descreve o processo de escrita dos dados das BRAMs em um arquivo. Inicialmente, o algoritmo realiza a soma do número de bits, $bram_{width}$, e do parâmetro $bram_{par}$, resultando em $bram_{width+par}$. Em seguida, o algoritmo repete o caractere '0' um número de vezes igual ao valor do parâmetro $bram_{par}$, produzindo a constante da BRAM,

denotada como $bram_{const}$. Com isso os dados que serão escritos terão a mesma largura da porta de entrada da BRAM.

1: **sum** $bram_{width}$ + $bram_{par}$ to $bram_{width+par}$
2: **repeat** '0' times $bram_{par}$ to $bram_{const}$
3: **format** *bram_file* with *list_bram_instantiation* , $bram_{addr}$, $bram_{we}$, $bram_{const}$, $bram_{width+par}$, $bram_{width}$ and $bram_{par}$
4: **write** *bram_file*

Algoritmo 4.7 – Escrita de Dados de BRAM em um Arquivo

O próximo passo do algoritmo é formatar o arquivo da BRAM, utilizando a lista de instanciações da BRAM, e os parâmetros *n_bits*, $bram_{addr}$, $bram_{we}$, $bram_{const}$, $bram_{width}$ e $bram_{par}$. Finalmente, o algoritmo escreve o arquivo formatado da BRAM, *bram_file*, conforme apresentado na Descrição 4.3.

Código 4.3 – Exemplo de instanciação de BRAM.

```
MEM_IWGHT_LAYER0_INSTANCE0 : if BRAM_NAME = "iwght_layer0_instance0" generate
    BRAM_SINGLE_MACRO_inst : BRAM_SINGLE_MACRO
    generic map (
        BRAM_SIZE => "36Kb",
        DEVICE => "7SERIES",
        DO_REG => 0,
        INIT => X"000000000000000000",
        INIT_FILE => "NONE",
        WRITE_WIDTH => 44,
        READ_WIDTH => 44,
        SRVAL => X"000000000000000000",
        WRITE_MODE => "WRITE_FIRST",
        INIT_00 => X"0000089400000000fffffa57ffffffff000004fe00000000ffff05bfffffffff",
        INIT_01 => X"00001869000000000000112a00000000000008a4000000000000001d00000000",
        ...
   )
```

O algoritmo de escrita final, como indicado anteriormente, opera sob uma lista, *list_bram_init*, que contém pesos, características e parâmetros 'gold' de todas as camadas, bem como uma instância vazia de BRAM. Este procedimento resulta na geração de um único arquivo. Esse método simplifica a instanciação das BRAMs, eliminando a necessidade de *generics* específicos para cada camada ou tipo de BRAM, seja ela de pesos e viés, características ou 'gold'. Portanto, proporciona uma abordagem simplificada para o gerenciamento e a manipulação dos dados das BRAMs.

## 4.6 Camadas Max Pooling e Fully Connected

Para executar uma CNN completa faltam pelo menos duas camadas que normalmente estão presentes: *max polling* e *fully connected*. Ambos os componentes foram desenvolvidos usando as mesmas portas da CONVWS (Seção 4.1) facilitando a integração dentro dos *Cores* desenvolvidos, e permitindo que utilizem o mesmo *test bench*, apresentado na Figura 4.4. Dessa forma o *Core* tem suporte para três operações: convolução, *max polling* e *fully connected*.

### 4.6.1 *Max pooling 2D* — MP2D

A arquitetura MP2D[17] implementa uma camada de *max-pooling* em duas dimensões, que é uma operação essencial em CNNs. Este componente está projetado para reduzir a quantidade das características de entrada (IFMAP) selecionando os dados mais importantes. Essa redução é realizada ao aplicar uma janela deslizante que percorre o IFMAP, produzindo um mapa de características de saída (OFMAP) com dimensões reduzidas.

A arquitetura é simples. A altura e a largura da janela de *pooling* ($kernel_{size}$) tem sempre o mesmo valor — para a janela ser quadrada. *Stride* é sempre igual ao tamanho do *polling* ($kernel_{size}$). *Padding*, o preenchimento a ser adicionado, em ambos os lados é sempre zero ([10]). A dilatação, que controla o *stride* dos elementos na janela ([10]), é sempre um.

Foram removidas as portas referentes a leitura de *weights* por não serem realizadas operações com pesos ou vieses. Por dessas decisões de projeto e características, a camada é composta apenas de uma FSM, Figura 4.21, que controla a ordem das operações, com quatro os estados:

- WAITSTART: Este é o estado inicial da máquina de estados. Neste estado, a máquina de estados aguarda o sinal de partida da operação de *max-pooling* ($start_{op}$). Uma vez recebido o sinal de partida, a máquina de estados passa para o estado UPDATEADD.

- UPDATEADD: Neste estado, a máquina de estados calcula os endereços de memória dos elementos na janela de *pooling* e os armazena para o próximo estado. São calculados $kernel_{size}^2$ endereços de *pooling*. Após definir todos os endereços necessários, a máquina de estados transita para o estado READFEATURES.

- READFEATURES: Neste estado, a máquina de estados lê os elementos na janela de *pooling* a partir dos endereços calculados no estado UPDATEADD. Se o elemento lido é maior que o valor máximo atual na janela de *pooling*, então o valor máximo é atualizado. Isso é feito para todos os elementos ($kernel_{size}^2$) na janela de *pooling*, após isso a máquina de estados passa para o estado WRITEFEATURES.

- WRITEFEATURES: Neste estado, o valor máximo encontrado na janela de *pooling* é escrito na memória de saída. A máquina de estados então verifica se se toda a imagem ou todos os mapas de características foram processados ($image_{size}^2 * image_{channel}$). Se sim, ela retorna ao estado WAITSTART para aguardar o próximo sinal de partida. Caso contrário, ela volta para o estado UPDATEADD para processar a próxima janela de *pooling*.

---

[17] Descrição VHDL:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/rtl/pool/maxpool2d.vhd

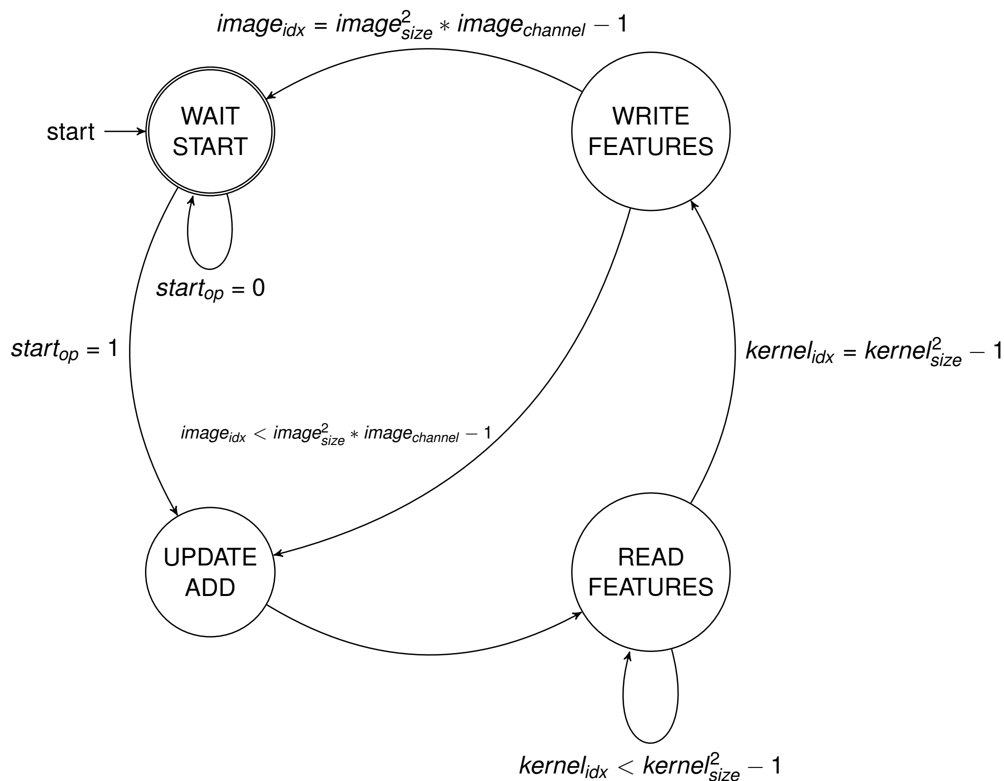



Figura 4.21 – Máquina de estados da *maxpooling2d*.

A Tabela 4.4 ilustra a ordem de leitura dos endereços durante uma operação de *max-pooling*. Especificamente, a tabela está mapeada para uma imagem de tamanho $6 \times 6$ (representada por $image_{size}$), com uma janela de *pooling* de tamanho 2 ($kernel_{size}$).

As células na tabela representam os endereços dos pixels na imagem de entrada. A ordem de leitura destes endereços é estruturada para facilitar a operação de *max-pooling*. Nesse caso, a operação de *max-pooling* é realizada em uma janela deslizante de tamanho $2 \times 2$ que percorre a imagem de entrada. Cada grupo de quatro números consecutivos na tabela representa uma janela de *pooling*.

Por exemplo, os endereços 0 e 1, seguidos por 2 e 3, formam a primeira janela de *pooling*. A operação de *max-pooling* é então aplicada a esses quatro pixels, resultando no valor máximo sendo retido. A janela de *pooling* então se move para o próximo grupo de quatro endereços, ou seja, 4 e 5, seguidos por 6 e 7, e o processo se repete. Isso continua até que a janela de *pooling* tenha percorrido todos os pixels da imagem.

| | | Columns | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Lines | 0 | 0 | 1 | 4 | 5 | 8 | 9 | 12 | 13 |
| | 1 | 2 | 3 | 6 | 7 | 10 | 11 | 14 | 15 |
| | 2 | 16 | 17 | 20 | 21 | 24 | 25 | 28 | 29 |
| | 3 | 18 | 19 | 22 | 23 | 26 | 27 | 30 | 31 |
| | 4 | 32 | 33 | 36 | 37 | 40 | 41 | 44 | 45 |
| | 5 | 34 | 35 | 38 | 39 | 42 | 43 | 46 | 47 |

Tabela 4.4 – Ordem de leitura dos endereços da *maxpooling2d* para uma imagem com tamanho ($image_{size}$)de $6 \times 6$ e pooling com tamanho ($kernel_{size}$) igual a 2

### 4.6.2 *Fully connected* — FC

A arquitetura FC[18] é projetada para implementar uma camada totalmente conectada (Fully Connected Layer) de uma rede neural profunda. Essa camada realiza a multiplicação dos valores das características de entrada (IFMAPS) pelos pesos (WEIGHTS), seguida pela adição dos resultados para produzir os mapas de características de saída (OFMAPS), conforme Equação 4.5.

$$feature_{output} = feature_{input} * weight^{T} + bias \tag{4.5}$$

FC é composto por uma máquina de estados, um MAC e um banco de registradores, um demultiplexador e um multiplexador, conforme apresentado na Figura 4.22. Há um registrador para cada OFMAP onde são escritos os resultados parciais, diferentemente da CONVWS. O demultiplexador $M_D$ seleciona qual registrador ($reg_n$) receberá em (Q) o resultado (porta `res`) do MAC. O multiplexador $M_Q$ seleciona a saída (Q) de um dos registrador ($reg_n$), este sinal será direcionado para a porta `sum` do MAC.

Para o cálculo do OFMAP, a arquitetura inclui uma instância de uma entidade denominada MAC (multiplicador-acumulador). A entidade MAC recebe os WEIGHTS e os IFMAPS e realiza operações de multiplicação entre ambos e a soma com o resultado do atual do registrador. As instâncias da entidade registrador são usadas para armazenar temporariamente os resultados da operação do MAC.

A operação do módulo é gerenciada por uma máquina de estados finitos, Figura 4.23. Cada estado é associado a um estágio diferente no processamento de uma camada totalmente conectada:

[18] Descrição VHDL:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/rtl/fully_connected/simple.vhd

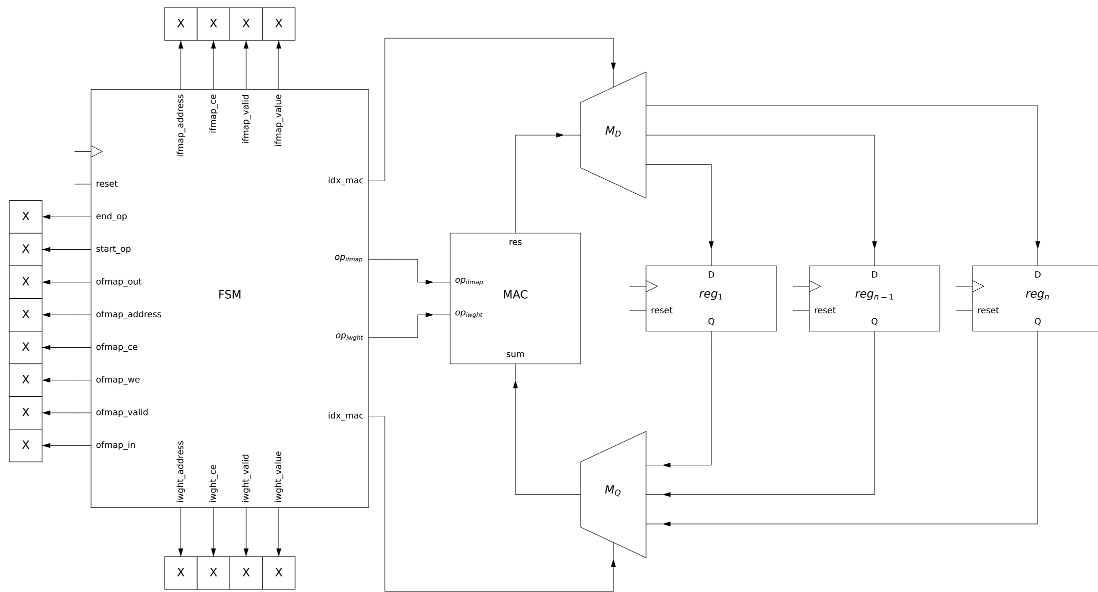



Figura 4.22 – Fully connected.

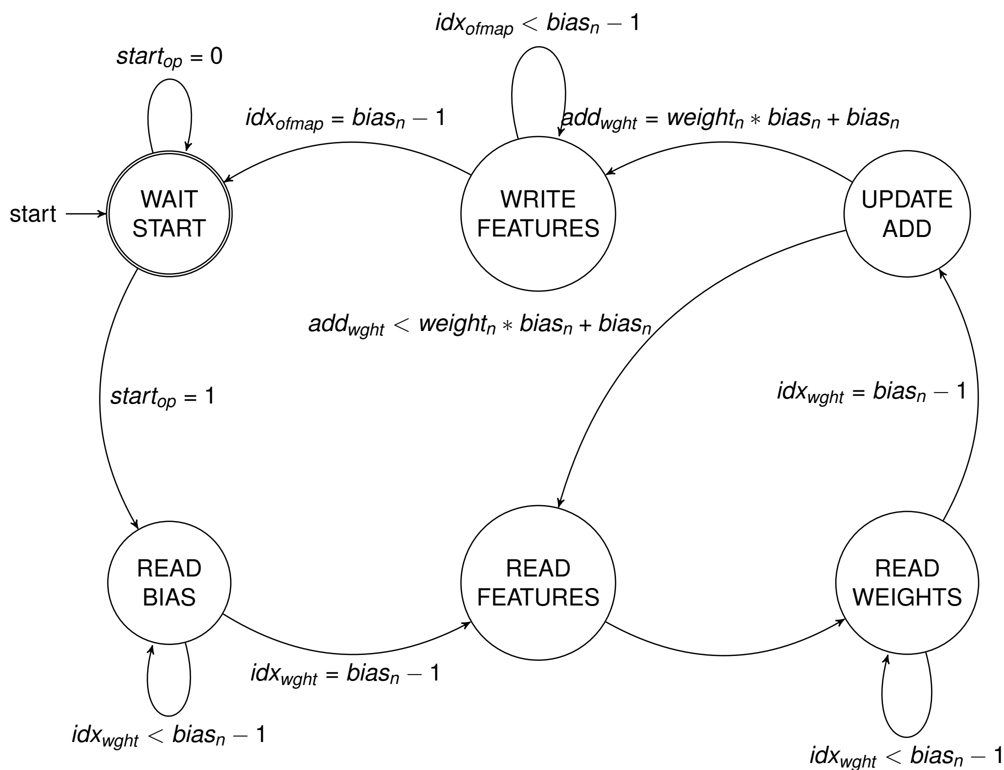



Figura 4.23 – Máquina de estados da arquitetura *fully_connected*

- WAITSTART: É o estado inicial no qual a máquina de estados está aguardando o sinal de início da operação ($start_{op}$ = 1). Se a operação é iniciada, a máquina de estados muda para o estado READBIAS.
- READBIAS: Este estado é responsável pela leitura dos pesos (ou viés). Durante este estado, se os pesos forem válidos, as operações de multiplicação e adição são rea-

lizadas nos pesos e as características de entrada, e os resultados são armazenados em registradores temporários. Após ler todos os pesos ($idx_{wght} = bias_n$), a máquina de estados passa para o estado READFEATURES.

- READFEATURES: Neste estado, a máquina de estados lê as características de entrada. Quando as características de entrada são válidas, a máquina de estados muda para o estado READWEIGHTS.
- READWEIGHTS: Este estado é responsável pela leitura dos pesos. Aqui, as operações de multiplicação e adição são realizadas nos pesos e as características de entrada, e os resultados são armazenados em registradores temporários. Após ler todos os pesos ($bias_n$), a máquina de estados muda para o estado WAITVALID.
- UPDATEADD: Esse estado atualiza os endereços dos pesos ($add_{iwght}$). Caso todos os pesos tenham sido processados ($add_{wght} = weight_n * bias_n + bias_n$) o estado é alterado para WRITEFEATURES.
- WRITEFEATURES: Neste estado, a máquina de estados escreve os mapas de características de saída para o registrador de saída. Após escrever todas as características de saída ($idx_{ofmap} = bias_n$), a máquina de estados retorna para o estado WAITSTART para aguardar a próxima operação.

## 4.7 Acelerador em FPGA

A descrição VHDL[19] proposto tem por finalidade criar um ambiente de hardware sintetizável para a implementação de uma CNNSM em um dispositivo FPGA. Sua estrutura é projetada de maneira a conter todas as operações e módulos necessários ao funcionamento da CNN, que são descritos em um *test bench*, dentro de uma arquitetura inteiramente sintetizável.

A motivação para tal abordagem advém da própria natureza dos FPGAs, que, por serem circuitos reprogramáveis, não conseguem processar a descrição não sintetizável, como comumente se encontra em testbenches. Portanto, o objetivo primordial da descrição VHDL em questão é converter os elementos descritivos e funcionais do *test bench* da CNN em um formato que possa ser sintetizado e, assim, implantado em um FPGA.

A arquitetura *Accelerator* (ACCEL) foi desenvolvida para essa finalidade, ela inclui três componentes principais: uma MEMFMAP, uma instância de CNNSM e uma máquina de estados. A MEMFMAP tem uma amostra de imagem nessa versão inicial. A CNNSM instanciada foi desenvolvida nesse trabalho.

[19] Descrição VHDL:
https://github.com/tarsioonofrio/acc_dse_env/blob/484fa709cfe5597e464643d2bc0e20e2b14412e6/rtl/accelerator/simple.vhd

A máquina de estados na arquitetura do ACCEL, Figura 4.24, controla a sequência de operações no acelerador, ela tem seis estados distintos na máquina de estados:

- WAITSTART: a máquina de estados espera por um sinal de início para começar o processo na CNNSM. Durante este estado, todos os sinais de controle e endereçamento são inicializados para seus estados padrão.
- WRITEFEATURES: a máquina de estados começa a escrever os valores de entrada na memória para uso posterior. O endereço na memória é incrementado após cada ciclo de escrita. Uma vez que todos os valores de entrada necessários tenham sido escritos na memória, a máquina de estados avança para o próximo estado.
- STARTCNN: a máquina de estados envia um sinal para iniciar a operação da CNNSM. O controle da memória é desativado durante este estado para evitar interferência com a operação da CNN.
- STOPCNN: a máquina de estados aguarda o término da operação da CNNSM. Uma vez que a CNNSM tenha concluído a operação na última CONVWS, a máquina de estados avança para o próximo estado.
- VALIDATE: a máquina de estados começa a validar a saída da CNNSM comparando-a com os "dados de ouro". O endereço na memória é incrementado após cada ciclo de validação. Se a saída da CNNSM for validada com sucesso, a máquina de estados retorna ao estado WAITSTART, caso contrário, ela gera um relatório de falha.

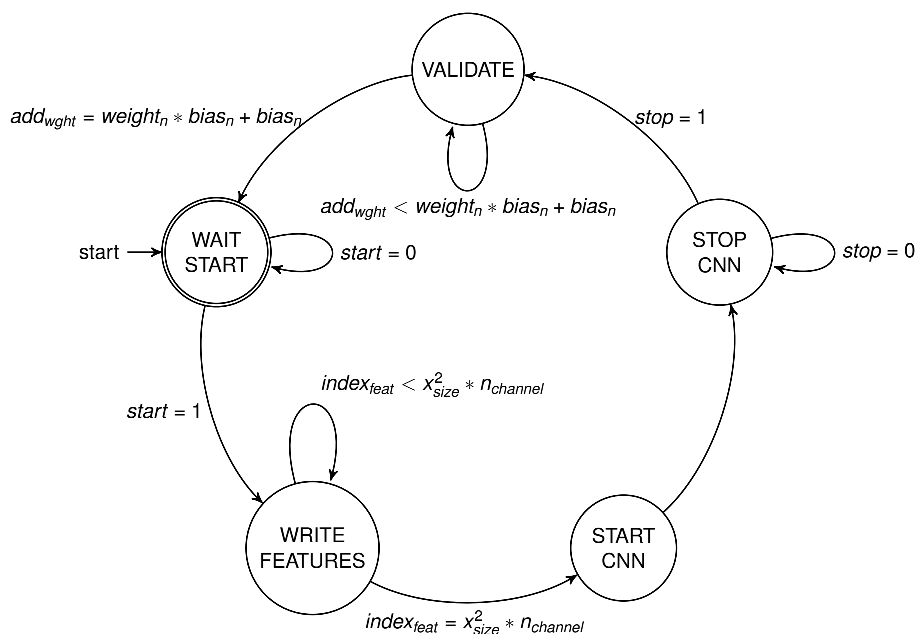



Figura 4.24 – Máquina de estados do ACCEL

# 5. RESULTADOS

Este Capítulo está organizado em 5 Seções. A Seção 5.1 apresenta o *dataset* utilizado para validar as CNNs. A Seção 5.2 apresenta um conjunto de 7 modelos de CNNs que foram validadas com o framework proposto. A Seção 5.3 compara o desempenho do acelerador em relação a GPU e CPU. A Seção 5.4 avalia a utilização de recursos no FPGA Artix A7. Finalmente, a 5.5 realiza uma avaliação preliminar com trabalhos que implementam CNNs em FPGAs.

## 5.1 Conjunto de dados

O CIFAR-10 [8] é um conjunto de dados (*dataset*) comumente utilizado em aprendizado de máquina e visão computacional. Este dataset, disponibilizado pelo Canadian Institute for Advanced Research (CIFAR), é composto por 60.000 imagens coloridas de 32x32 pixels, divididas em 10 classes distintas. A base de dados é subdividida em 50.000 imagens para treinamento e 10.000 para teste. A Figura 5.1 ilustra uma CNN para executar o *dataset* CIFAR, composta por 3 camadas convolucionais (*layers* 0,1,2) e uma camada totalmente conectada (FC).

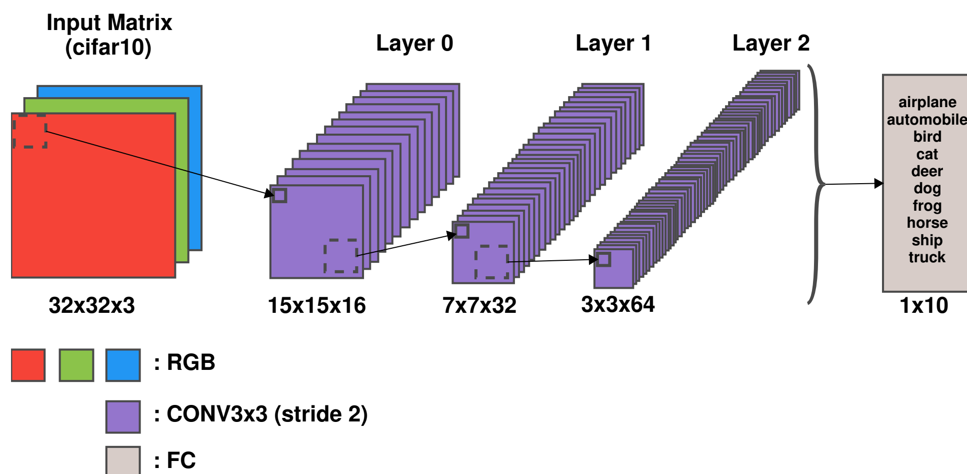



Figura 5.1 – Exemplo de uma CNN para executar o *dataset* CIFAR [4].

## 5.2 Modelos

Neste estudo, foram elaborados e treinados sete diferentes modelos de CNNs, com o objetivo de explorar a influência da estrutura da rede e da configuração dos filtros de convolução no desempenho da implementação em hardware:

- Quatro modelos designados por $D$, cada um composto por 4 camadas, sendo 3 camadas de convolução e 1 camada totalmente conectada (FC). A configuração de filtros para as camadas de convolução de cada modelo é:
  - $D_1$: Possui 16, 32 e 64 filtros nas camadas de convolução, respectivamente, sendo a configuração apresentada na Figura 5.1.
  - $D_2$: Todos as camadas de convolução contêm 16 filtros.
  - $D_3$: Cada camada de convolução contêm 32 filtros.
  - $D_4$: Todas as camadas de convolução possuem 64 filtros.
- Dois modelos identificados como $S$, cada um contendo 3 camadas, sendo 2 camadas de convolução e 1 camada FC. A configuração dos filtros para as camadas de convolução é como segue:
  - $S_1$: Apresenta 16 e 32 filtros nas camadas de convolução.
  - $S_2$: Ambas as camadas de convolução contêm 32 e 64 filtros, respectivamente.
- Um modelo denominado $M$, composto por 5 camadas, das quais 3 são camadas de convolução, 1 camada de *pooling* máximo (*max pool*) e 1 camada FC. A configuração de filtros para as camadas de convolução é semelhante à de $D_1$, com 16, 32 e 64 filtros, diferenciando-se pela adição de uma camada de *max pool* (MP2D).

Todos os modelos foram submetidos a testes com a implementação de BRAMs desenvolvida no âmbito deste trabalho. No entanto, é importante ressaltar algumas limitações decorrentes do tempo disponível, o que impossibilitou a integração completa das novas camadas, FC e MP2D, no framework. Como resultado, foi possível simular e sintetizar a CNNSM somente com as camadas convolucionais, sendo inviável a síntese do modelo $M$ completo. As duas novas camadas foram validadas individualmente por meio de simulações.

A Tabela 5.1 apresenta os dados relacionados à quantidade de BRAMs e parâmetros para os modelos $D$ e $S$ de CNNs. Cada modelo é caracterizado pela quantidade de BRAMs utilizadas para os pesos e para as características extraídas, bem como pelo número de parâmetros totais.

Vale ressaltar que existem mais 6 BRAMs e 3072 parâmetros adicionais correspondentes aos dados de entrada de dados presente na MEMFMAP do ACCEL. Esses recursos adicionais são necessários para a manipulação e processamento dos dados de entrada na CNNSM.

Tabela 5.1 – Quantidade de BRAMs e parâmetros gerados para as DNN.

| | Nb. BRAMs | | Nb. Parameters | |
|---|---|---|---|---|
| | Weights | Features | Weights | Features |
| $D_1$ | 60 | 20 | 29354 | 8816 |
| $D_2$ | 14 | 17 | 6538 | 7600 |
| $D_3$ | 46 | 26 | 22282 | 12128 |
| $D_4$ | 162 | 44 | 81418 | 21184 |
| $S_1$ | 42 | 18 | 20778 | 8240 |
| $S_2$ | 101 | 28 | 50762 | 13408 |

## 5.3 Benchmark

Esta seção apresenta um estudo comparativo de desempenho entre o acelerador proposto com frequência 100 MHz, um dispositivo GPU (Nvidia GeForce GTX 980 Ti e Intel® Core™ i5-2320 CPU @ 3.00GHz × 4) e uma CPU (Intel(R) Core(TM) i9-7940X CPU @ 3.10GHz). O desempenho é avaliado em termos de tempo de execução em milissegundos para processar várias tarefas. Serão analisados diferentes modelos de CNNs, variando a quantidade de filtros em suas camadas de convolução e o número de lotes de processamento. Através desta análise, pretendemos ilustrar o potencial do acelerador para superar o desempenho de hardware tradicionais em tarefas que exigem alta capacidade de processamento.

A Tabela 5.2 mostra que o acelerador proposto apresenta um desempenho superior ao da GPU e da CPU, indicado por um menor tempo de execução em milissegundos para processar as tarefas. No caso do acelerador, o tempo de processamento para uma única amostra ($batch_1$) varia de aproximadamente 6,21 ms ($D_2$) a 72,41 ms ($D_4$).

Tabela 5.2 – Benchmark da simulação de cada CNN no Modelsim em milisegundo (ms) pressupondo ACC com clock de 100 MHz em relação a GPU (Nvidia GeForce GTX 980 Ti e Intel® Core™ i5-2320 CPU @ 3.00GHz × 4) e uma CPU (Intel(R) Core(TM) i9-7940X CPU @ 3.10GHz). O resultado do $batch_{10}$ o tempo médio das 10 amostras.

| | Accelerator | GPU | | CPU | |
|---|---|---|---|---|---|
| | $batch_1$ | $batch_1$ | $batch_{10}$ | $batch_1$ | $batch_{10}$ |
| $D_1$ | 15,65 | 33,38 | 3,53 | 46,93 | 4,40 |
| $D_2$ | 6,21 | 34,65 | 17,06 | 44,92 | 4,40 |
| $D_3$ | 20,36 | 33,43 | 17,42 | 48,22 | 4,68 |
| $D_4$ | 72,41 | 33,72 | 17,80 | 46,54 | 4,98 |
| $S_1$ | 8,35 | 34,35 | 4,65 | 44,72 | 4,99 |
| $S_2$ | 28,92 | 34,61 | 8,25 | 45,72 | 5,07 |

Em contraste, os tempos de processamento da GPU e da CPU são significativamente mais altos. Nota-se que a GPU tem um desempenho consistente, com tempo de processamento variando entre 33,38 ms ($D_1$) e 34,65 ms ($D_2$) para $batch_1$, enquanto a CPU demonstra um desempenho mais variável, com tempos que vão de 44,72 ms ($S_1$) a 48,22 ms ($D_3$) para o mesmo parâmetro.

É interessante observar que a eficácia do acelerador é notavelmente superior quando se trabalha com um único lote ($batch_1$) em comparação com 10 lotes ($batch_1 0$).

Nota-se que o tempo de execução para $batch_{10}$ é inferior ao de $batch_1$ tanto na GPU quanto na CPU. Este fenômeno pode ser explicado considerando-se duas características principais das modernas arquiteturas de GPU e CPU: a execução paralela e o overhead na transferência de dados.

Com a execução paralela, ao aumentar o tamanho do lote de dados (neste caso, para 10), as unidades de processamento da GPU ou da CPU conseguem distribuir as operações de maneira mais eficiente entre os múltiplos núcleos de processamento, levando a um menor tempo de execução por amostra.

Em relação à transferência de dados, cada operação de transferência de dados entre a memória principal e a GPU ou CPU incide em um overhead de tempo. Ao processar um lote maior de uma só vez, ocorre uma redução no número de operações de transferência necessárias, resultando em um menor tempo total de transferência de dados.

A Tabela 5.3 compara o desempenho, em milissegundos (ms), de diferentes camadas de diferentes modelos de redes neurais quando simuladas no acelerador. Ao examinar os dados, é possível observar que o tempo necessário para a execução da operação de convolução aumenta conforme a quantidade de filtros nas camadas de convolução dos modelos aumenta. Para as redes D, o modelo $D_1$, que tem 16, 32 e 64 filtros nas camadas de convolução, respectivamente, apresenta os menores tempos para as operações na camada 1 (2,25 ms), camada 2 (6,10 ms) e camada 3 (7,29 ms). Por outro lado, o modelo $D_4$, que tem 64 filtros em todas as três camadas de convolução, apresenta os maiores tempos para as operações na camada 1 (9,00 ms), camada 2 (48,83 ms) e camada 3 (14,58 ms).

Similarmente, para as redes S, o modelo $S_1$, que tem 16 e 32 filtros nas duas camadas de convolução, apresenta tempos menores para as operações na camada 1 (2,25 ms) e camada 2 (6,10 ms) quando comparado com o modelo $S_2$, que tem 32 e 64 filtros nas camadas de convolução e apresenta tempos de 4,50 ms e 24,41 ms para as operações na camada 1 e camada 2, respectivamente.

Tabela 5.3 – Benchmark da simulação no Modelsim em milisegundo (ms) pressupondo clock de 100 MHz para ACC na execução das operações de convolução.

| | $Layer_1$ | $Layer_2$ | $Layer_3$ |
|---|---|---|---|
| $D_1$ | 2,25 | 6,10 | 7,29 |
| $D_2$ | 2,25 | 3,05 | 0,91 |
| $D_3$ | 4,50 | 12,21 | 3,65 |
| $D_4$ | 9,00 | 48,83 | 14,58 |
| $S_1$ | 2,25 | 6,10 | |
| $S_2$ | 4,50 | 24,41 | |

## 5.4 Uso de recursos da FPGA

Esta seção discute a utilização de recursos de hardware do acelerador por diferentes Redes CNNs. Abordaremos em detalhes a alocação de *Look-Up Tables* (LUTs), *Flip-Flops* (FFs), blocos de processamento digital de sinais (DSP) e blocos de memória RAMB36 para cada módulo, sendo eles: ACCEL, CNNSM, CORESR, CONVWS e *memory* incluindo MEMFMAP (BRAMs com mapa de características de entrada e saída), MEMWGHT (BRAMS com pesos e viéses dos filtros da convolução). Os resultados não serão apresentados de forma hierárquica, mas serão apresentados o consumo de recursos de cada módulo excluindo o consumo dos seus submódulos.

A prototipação está em andamento, com o projeto já implementado na FPGA. Entretanto, o processo de validação ainda não foi concluído. Não foi possível validar o resultado da mesma forma como foi realizado nas simulações. Também não foi possível medir o tempo de execução do modelo no hardware descrito.

A Figura 5.2 apresenta um diagrama de conjuntos que ilustra o uso médio de LUTs (Look-Up Tables) e FFs (Flip-Flops) por módulo, sendo que o nome de cada módulo é destacado em negrito. O diagrama é dividido em vários círculos, cada um representando um módulo específico e o seu respectivo uso de LUTs e FFs, proporcionando uma visão da distribuição dos recursos de LUTs e FFs por diferentes módulos na arquitetura.

Os círculos internos representam os módulos individuais: CONVWS e MEM (MEMFMAP e MEMWGHT), que estão contidos em três núcleos, denominados $CORESR_1$, $CORESR_2$ e $CORESR_3$. Cada CONVWS consome uma média de 1580 FFs e aproximadamente 970 LUTs, enquanto o módulo MEM consome 70 FFs. Adicionalmente, cada núcleo tem um consumo de 68.5 LUTs. Estes três núcleos estão contidos na entidade CNNSM, que consome uma quantidade adicional de LUTs, especificamente 126 para o modelo D e 87.5 para o modelo S. Finalmente, a entidade ACCEL, que contém a CNNSM e todos os seus núcleos, consome uma quantidade adicional de 56 FFs e 53 LUTs.

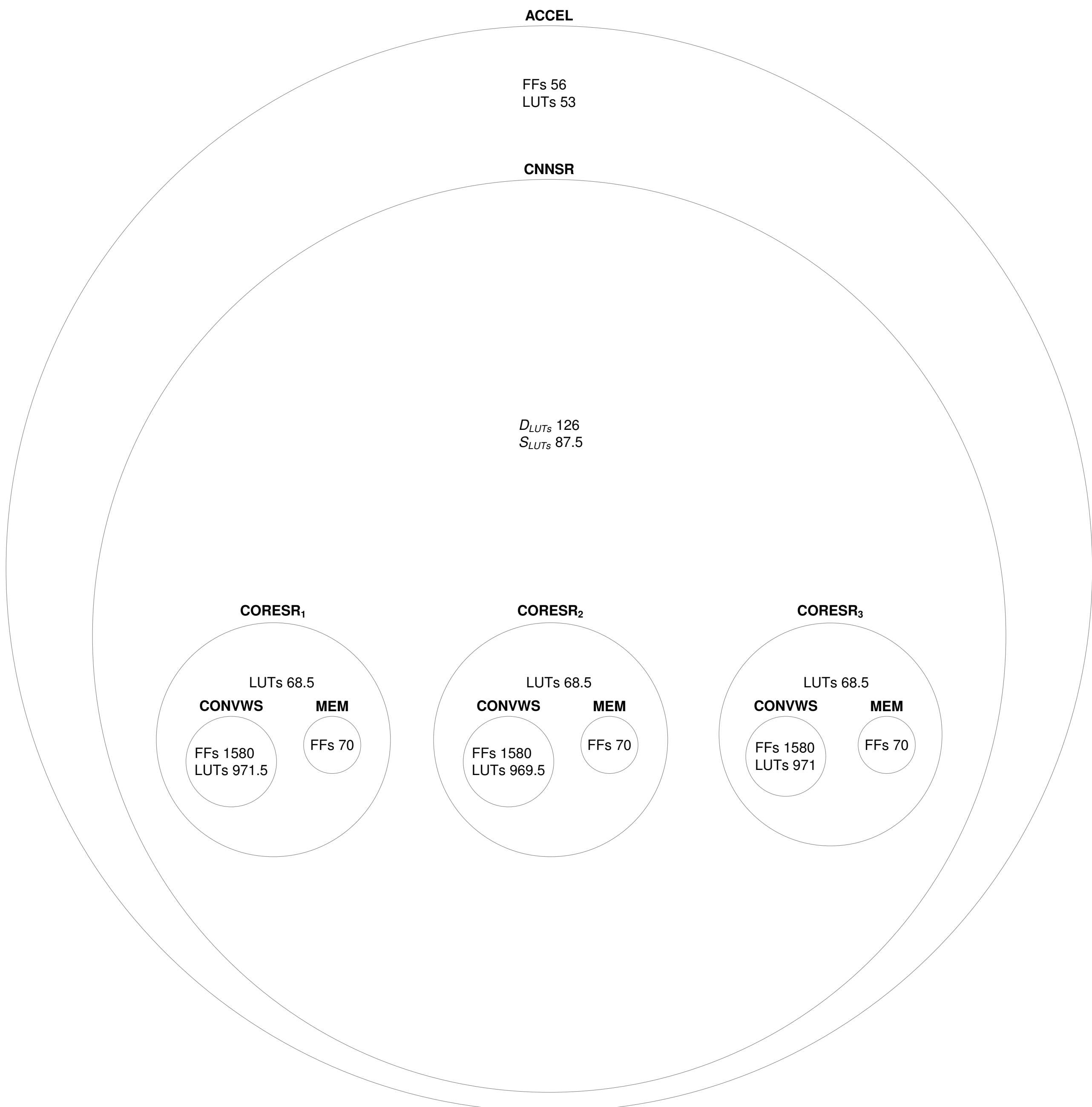



Figura 5.2 – Diagrama de conjuntos com a média de uso de LUTS e FFs por módulo. Nome de cada módulo em negrito.

Nos parágrafos a seguir explicaremos em detalhes o consumo de recursos de cada módulo para cada modelo de CNNSM testado.

A Tabela 5.4 mostra o uso dos recursos da FPGA para a prototipação de diferentes modelos de redes neurais. Os recursos computacionais da FPGA, representados pelos LUTs, FFs, RAMB36 e DSP (blocos de processamento digital de sinal), são distintamente utilizados pelos diferentes modelos de redes neurais. Para todos os modelos, a quantidade de LUTs, FFs e DSP Blocks permanece relativamente estável. Observar que o CONVWS possui uma matrix 3x3 de convolução, justificando o número de 9 blocos DSP inferidos por camada convolucional.

No entanto, a utilização de RAMB36 mostra uma variação significativa dependendo do modelo. Os modelos D, que possuem mais camadas e filtros do que os modelos S, requerem mais blocos de memória, em média. Em particular, o modelo $D_1$ requer 74 blocos de memória, enquanto o modelo D2 requer apenas 34. Esta diferença deve-se à diferença na configuração dos filtros das camadas de convolução desses modelos.

Por outro lado, os modelos S requerem menos FF e LUTs que os modelos D, conforme seria de esperar devido à sua menor complexidade. O modelo $S_1$ requer 35 RAMB36, entretanto o modelo $S_2$, que possui mais filtros nas camadas de convolução, requer 73 blocos de memória.

Tabela 5.4 – Utilização de recursos da FPGA para cada CNN.

| | LUTs | FFs | RAMB36 | DSP Blocks |
|---|---|---|---|---|
| $D_1$ | 4428 | 5360 | 74 | 27 |
| $D_2$ | 3923 | 5307 | 34 | 27 |
| $D_3$ | 4343 | 5333 | 72 | 27 |
| $D_4$ | 4358 | 5351 | 72 | 27 |
| $S_1$ | 2821 | 3618 | 35 | 18 |
| $S_2$ | 3171 | 3640 | 73 | 18 |

O módulo do ACCEL (Seção 4.7) apresenta uma utilização constante de 56 FFs e 53 LUTs, com a única exceção sendo o modelo D2, que utilizou 55 LUTs. Como o ACCEL não possui parâmetros, sua utilização de recursos permanece constante. Este módulo realizada o controle da CNNSM.

A Tabela 5.5 descreve a utilização de LUTs pelo módulo que controla a instanciação dos módulos CNNSM (Seção 4.4.3. Os resultados mostram uma utilização constante de LUTs em todos os modelos D ($D_1$, $D_2$, $D_3$ e $D_4$), com cada um utilizando 126 LUTs. As redes S ($S_1$ e $S_2$) usaram menos LUTs, com 90 e 85, respectivamente. Isso pode ser atribuído ao fato de que os modelos S têm menos camadas do que os modelos D, resultando em menor complexidade e, consequentemente, menor utilização de recursos. Vale ressaltar este módulo não possui registradores, uma vez que sua função é o roteamento de sinais entre os CORESR e a MEMFMAP.

Tabela 5.5 – Utilização de recursos da FPGA pelo módulo CNNSR.

| | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $S_1$ | $S_2$ |
|---|---|---|---|---|---|---|
| LUTs | 126 | 126 | 126 | 126 | 90 | 85 |

A Tabela 5.6 mostra a utilização de LUTs por cada instância do CORESR (Seção 4.4.2)em diferentes CNNs. Observa-se que, para as CNNs tipo D ($D_1$, $D_2$, $D_3$, e $D_4$), o uso de LUTs é uniforme. Especificamente, a maior parte das instâncias requer 69 LUTs, com algumas exceções em $D_3$ e $D_4$ que requerem um pouco menos, entre 65 e 68 LUTs.

Para as CNNs tipo S ($S_1$ e $S_2$), que são menos complexas, o uso de LUTs por instância do CORESR é menor, situando-se entre 60 e 65 LUTs.

Como visto anteriormente, o módulo CORESR, semelhante ao módulo da CNNSM, não possui registradores, pois tem como função primária o roteamento de sinais entre um CONVWS e uma MEMFMAP.

Tabela 5.6 – Utilização de recursos da FPGA por instância de CORESR em cada CNN

| | $D_1$ | | | $D_2$ | | | $D_3$ | | | $D_4$ | | | $S_1$ | | $S_2$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $Core_n$ | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 1 | 2 |
| LUTs | 69 | 69 | 69 | 69 | 69 | 69 | 66 | 69 | 65 | 68 | 68 | 65 | 65 | 69 | 61 | 60 |

O CONVWS na Figura 5.7 tem pouca variação de utilização de recursos seja LUTsn ou FFs mesmo diante das variações da quantidade de canais de entrada ou saída. Cada CONVWS tem 9 blocos de DSP, tem entre 962 e 973 LUTS e entre 1570 e 1585 FFs. Ao analisar a tabela 5.7, é possível observar que a utilização de LUTs, FFs e blocos DSPs em cada instância de CONVWS para diferentes CNNs mantém-se consistente.

Para as LUTs, todos os modelos D ($D_1$, $D_2$, $D_3$ e $D_4$) e S ($S_1$ e $S_2$) apresentam pequena variação na utilização de LUTs entre as instâncias de CONVWS, aproximadamente 970. No caso dos FFs, a consistência é similar, com todos os núcleos e CNNs utilizando por volta de 1580 FFs. Nota-se uma pequena diferença no modelo D2, que emprega em torno de 1570 FFs. Além disso, cada instância de CONVWS utiliza 9 blocos de DSP referentes aos 9 MACs do módulo.

Estas constatações demonstram que a implementação do acelerador para redes de diferentes complexidades apresenta um padrão de utilização de LUTs, FFs e blocos DSPs. A consistência na utilização de recursos, independentemente do número de canais de entrada ou saída, indica que o acelerador pode acomodar várias CNNs sem aumento significativo na utilização de recursos da FPGA.

Tabela 5.7 – Utilização de recursos da FPGA por instância de CONVWS por CORESR e CNN.

| | D1 | | | D2 | | | D3 | | | D4 | | | S1 | | S2 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $Core_n$ | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 1 | 2 |
| LUTs | 971 | 972 | 972 | 970 | 962 | 970 | 964 | 972 | 973 | 972 | 970 | 969 | 973 | 964 | 973 | 969 |
| FFs | 1585 | 1584 | 1585 | 1570 | 1571 | 1570 | 1577 | 1577 | 1577 | 1583 | 1583 | 1583 | 1577 | 1577 | 1584 | 1585 |

O MEMFMAP do ACCEL, que armazena os dados de entrada da imagem para a CNNSM, mantém uma uniformidade no uso de recursos de hardware. Isso ocorre porque a camada de entrada é idêntica em todas as implementações e imagens com as mesmas dimensões são utilizadas em todos os modelos. Nesse contexto, é observado que a MEMFMAP constantemente aloca 6 blocos de memória RAMB36 e 68 FFs, independentemente do modelo de CNN implementado, sejam eles $D_1$, $D_2$, $D_3$, $D_4$, $S_1$ ou $S_2$.

Em relação ao uso de LUTs, a MEMFMAP apresenta ligeiras diferenças dependendo da CNN. Em particular, as redes $D_1$, $D_3$, $D_4$ e $S_2$ necessitam de 85 LUTs, a rede $D_2$ precisa de 86 LUTs e a rede $S_1$ requer 92 LUTs.

As Tabelas 5.8, 5.9 e 5.10 apresentam o uso de recursos para os módulos de memória MEMFMAP e MEMWGHT para cada modelo de rede. Os FFs variam pouco, entre 67 e 75. Isso ocorre devido ao fato que apenas um dos registradores precisa ser parametrizado, o qual armazena a informação da BRAM a ser que está sendo lida, e é parametrizado com a quantidade de BRAMs no módulo.

A Tabela 5.8 indica o uso de recursos pela MEMFMAP do módulo CNNSM. Observa-se que a alocação de LUTs e FFs varia levemente entre os diferentes modelos, enquanto o uso de RAMB36 é mais diversificado, destacando-se os modelos S1 e S2.

Tabela 5.8 – Utilização de recursos da FPGA pela MEMFMAP do módulo CNNSR para cada modelo de CNN.

| | LUTs | FFs | RAMB36 |
|---|---|---|---|
| $D_1$ | 46 | 67 | 2 |
| $D_2$ | 45 | 66 | 1 |
| $D_3$ | 37 | 66 | 1 |
| $D_4$ | 44 | 66 | 1 |
| $S_1$ | 87 | 68 | 4 |
| $S_2$ | 94 | 68 | 7 |

A Tabela 5.9 exibe o uso de recursos pela MEMFMAP diferenciando por CORESR e por CNN. Há uma variação mais acentuada no uso de RAMB36 e LUTs entre os diferentes modelos de CNN, enquanto a utilização de FFs mantém-se mais estável.

Tabela 5.9 – Utilização de recursos da FPGA por instância de MEMFMAP de entrada por CORESR e CNN.

| | D1 | | | D2 | | | D3 | | | D4 | | | S1 | | S2 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $Core_n$ | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 1 | 2 |
| RAMB36 | 6 | 8 | 4 | 6 | 8 | 2 | 6 | 15 | 4 | 6 | 15 | 4 | 6 | 8 | 6 | 15 |
| LUTs | 91 | 131 | 87 | 92 | 132 | 47 | 91 | 183 | 86 | 91 | 183 | 87 | 92 | 131 | 91 | 183 |
| FFs | 68 | 69 | 68 | 68 | 69 | 67 | 68 | 69 | 68 | 68 | 69 | 68 | 68 | 69 | 68 | 69 |

Por fim, a Tabela 5.10 apresenta a utilização de recursos da FPGA pelo módulo que aloca os pesos dos filtros de convolução (MEMWGHT) diferenciando por CORESR e por CNN. Nessa tabela, percebe-se que a utilização de RAMB36 e LUTs lógicos varia de maneira mais acentuada em comparação com a tabela anterior, especialmente em modelos com mais filtros de convolução. Entretanto, a alocação de FFs mostra uma variação relativamente pequena.

Tabela 5.10 – Utilização de recursos da FPGA por MEMWGHT por CORESR e CNN.

| | D1 | | | D2 | | | D3 | | | D4 | | | S1 | | S2 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $Core_n$ | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 1 | 2 |
| RAMB36 | 1 | 10 | 37 | 1 | 5 | 5 | 2 | 19 | 19 | 2 | 19 | 19 | 1 | 10 | 2 | 37 |
| LUTs | 40 | 174 | 473 | 43 | 91 | 91 | 47 | 266 | 265 | 47 | 266 | 266 | 37 | 178 | 47 | 471 |
| FFs | 66 | 69 | 75 | 66 | 68 | 68 | 67 | 70 | 70 | 67 | 70 | 70 | 66 | 69 | 67 | 75 |

A Figura 5.3 mostra que há uma correlação linear entre o número de blocos de memória RAMB36 e as LUTs utilizadas por cada módulo de Controle e Acesso as BRAMs (Seção 4.5.2). Essa relação linear sugere que o aumento no uso de blocos de memória RAMB36 é acompanhado proporcionalmente por um crescimento na utilização de LUTs. À medida que mais memória é requerida (RAMB36), mais lógica é necessária (LUTs) para o processamento de tais dados.

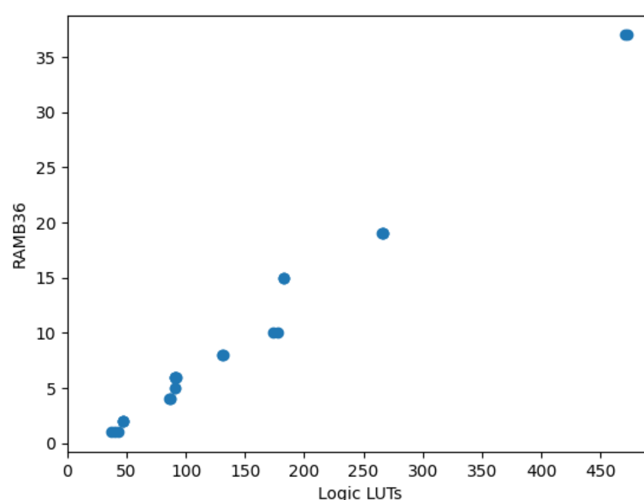
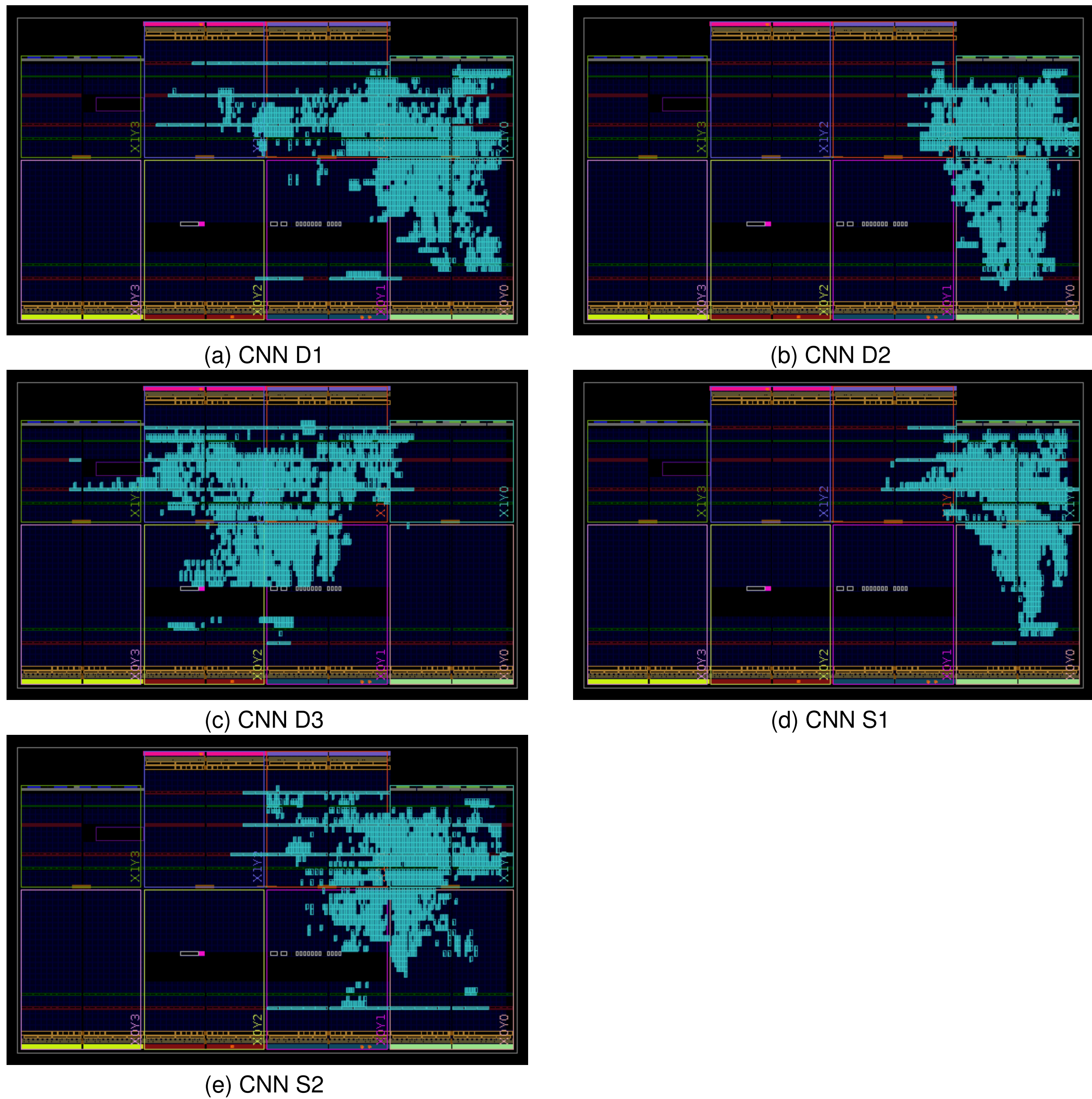



Figura 5.3 – Gráfico da relação entre RAMB36 e LUTs para cada instância do módulo de Controle e Acesso as BRAM.

A Figura 5.4 apresenta as plantas baixas (implementação física) no FPGA Artix7 para as CNNs $D_1$, $D_2$, $D_3$, $S_1$ e $S_2$. Uma inspeção visual dessas figuras revela variações distintas na alocação de recursos, com modelos maiores demandando uma quantidade maior de recursos comparados aos modelos menores. O modelo $D_4$ não pode ser implementado, uma vez que requer uma quantidade de BRAMB36 que excede a capacidade da FPGA Artix-7 utilizada neste estudo, que disponibiliza um total de 135 RAMB36.



(a) CNN D1

(b) CNN D2

(c) CNN D3

(d) CNN S1

(e) CNN S2

Figura 5.4 – Planta baixa no FPGA Artix 7 para os modelos de CNN explorados.

## 5.5 Comparação com Trabalhos Relacionados

A Tabela 5.11 compara a implementação *D*1 aos trabalhos relacionados [11]. Em comum os trabalhos utilizam representação inteira de ponto fixo (FP), como precisão semelhante ao trabalho proposto (16 bits). A ocupação de área é muito inferior aos demais trabalhos, pois utilizamos apenas um acelerador CONVWS por camada, enquanto as demais propostas utilizam paralelismo espacial para acelerar a CNN.

Tabela 5.11 – Comparação entre CNNs implementadas em FPGAs [11].

| Technique | Year | Key Features | Conv Layers | Image Operations (GOP) | Platform | Precision | Frequency (MHz) | LUT Type | Design Entry | Resources | | | | Resources Utilization | | | | Performance | Speedup | Baseline | Power Efficiency |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | BRAMs | LUTs | FFs | DSPs | BRAMs | LUTs | FFs | DSPs | | | | |
| CONV Coprocessor Accelerator | 2009 | Parallel clusters of 2D convolver units, data quantization,off-chip memory banks | 4 | 1,06 | Virtex5 LX330T | 16-bitFP | 115 | 6-input LUTs | C | 324 | 207360 | 207360 | 192 | 0,93% | 17,00% | 19,05% | 55,73% | 6,74 | 6x | 2.2 GHz AMD Opteron | 0,61 |
| MAPLE | 2010 | In-memory processing, banked off-chip memories,2D array of VPEs | 4 | 1,06 | Virtex5 SX240T | FP | 125 | 6-input LUTs | C++ | 516 | 149760 | 149760 | 1056 | N/A | | | | 7 | 0.5x | 1.3 GHz C870 GPU | N/A |
| DC-CNN | 2010 | Integer factorization to determine the best config for each layer,input and output switches | 3 | 0,52 | Virtex5 SX240T | 48-bitFP | 120 | 6-input LUTs | RTL | 516 | 149760 | 149760 | 1056 | N/A | | | | 16 | 4.0x-6.5x | 1.35 GHz C870 GPU | 1,14 |
| NeuFlow | 2011 | Multiple full-custom processing tiles (PTs), Pipelining, Fast streamingmemory interface | 4 | N/A | Virtex6 VLX240T | 16-bitFP | 200 | 6-input LUTs | HDL | 416 | 150720 | 301440 | 768 | N/A | | | | 147 | 133.6x | 2.66 GHz-Core 2 Duo CPU | 14,7 |
| Memory Centric Accelerator | 2013 | Flexible off-chip memory hierarchy, Data reuse,Loop transformation | 4 | 5,48 | Virtex6 VLX240T | FP | 150 | 6-input LUTs | C | 416 | 150720 | 301440 | 768 | 45,50% | 11,00% | N/A | 60,00% | 17 | 11x | Standard Virtex6 Implementation | N/A |
| nn-X | 2014 | Cascaded pipelining, Multiple streamprocessing | 2 | 0,55 | Zynq XC7Z045 | 16-bitFP | 142 | 4-input LUTs | Lua | 545 | 218600 | 437200 | 900 | N/A | | | | 23,18 | 115x | 800 MHzEmbedded ARM Cortex-A9 Processors | 2,9 |
| Este TCC (D1) | 2023 | | 3 | N/A | Artix-7 | 16-bit integer | N/A | 6-input LUTs | RTL | 80 | 4428 | 5360 | 27 | 54,81% | 6,98% | 42,00% | 11,25% | | N/A | GTX 980Ti | |

# 6. CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho atendeu o que foi proposto no título deste trabalho “Ambiente para Exploração de CNNs em nível RTL”. O trabalho de referência correspondeu a apenas um acelerador, sem integração ao outros aceleradores ou blocos de memória. Este trabalho, através da ferramenta de inicialização de BRAMs, permite gerar os arquivos de inicialização das memórias e parametrização da rede. O código VHDL permite instanciar diversos módulos de aceleração em série, permitindo a implementação de uma CNN completa. Além disto, o trabalho desenvolveu os módulos *max polling* e *fully connected*, adicionando flexibilidade na modelagem das redes CNN a partir do *framework* TensorFlow. Assim, temos todo um ambiente que nos permite agora explorar arquiteturas de CNN no nível RTL.

Em conclusão, este trabalho representa um avanço significativo no desenvolvimento de um ambiente de exploração para redes neurais convolucionais (CNNs) em nível RTL. A integração do *framework* TensorFlow permite parametrizar o código VHDL, gerando-se a rede conforme os parâmetros deste ambiente, e assim validar a CNN gerada em um simulador ou em dispositivos FPGAs.

Este TCC proporcionou ao autor um aprofundamento significativo em diversos conhecimentos relacionados à área de Engenharia de Computação. Durante o desenvolvimento do projeto, foram adquiridos conhecimentos sobre o funcionamento e a arquitetura de redes neurais convolucionais, bem como sobre técnicas de aceleração de hardware para processamento eficiente dessas redes. Além disso, o autor teve a oportunidade de explorar conceitos de descrição de hardware em nível RTL, compreendendo a importância da otimização e parametrização de módulos para a implementação de aceleradores de hardware. O trabalho também envolveu a aplicação de técnicas de projeto e desenvolvimento de sistemas digitais, incluindo a interconexão de interfaces de memória e a criação de *test bench* para validação do acelerador. Esses conhecimentos adquiridos são de grande relevância para a formação do autor como Engenheiro de Computação, capacitando-o a enfrentar desafios na área de processamento de dados e aprendizado de máquina.

Trabalhos futuros incluem:

1. **Implementar** a função de ativação da camada FC (Fully Connected), que não foi implementada por falta de tempo no TCC. A FC é uma camada importante em redes neurais convolucionais, e sua ativação é necessária para que o acelerador funcione corretamente.

2. Finalizar a integração da camada FC e MP2P com o framework.

3. Refatorar/**organizar** o framework desenvolvido. Refatorar o framework é importante para melhorar a organização e manutenção do código. Isso pode facilitar o desenvol-

vimento de novas funcionalidades e correção de bugs, além de tornar o código mais legível e compreensível .

4. Integrar o este trabalho com o fluxo de DSE do proposto na Tese de Loeonardo Juracy [4].
5. Aumentar a integração com o TensorFlow, utilizando-o diretamente para gerar as inicializações de memória. A sua integração com o acelerador pode trazer benefícios em termos de desempenho e facilidade de uso. Utilizar o TensorFlow diretamente para gerar as *features* pode ser uma forma de melhorar a integração.
6. Integração com o PyTorch, por ser mais flexível que Keras (TensorFlow).
7. Avaliar a utilização de memória unificada. Uma memória unificada pode trazer benefícios em termos de desempenho e eficiência. Testá-la pode ajudar a avaliar sua viabilidade e identificar possíveis melhorias ou limitações para trabalhos futuros.
8. Traduzir o código para Verilog. O uso de Verilog simplifica a integração com IPs gerados ou disponibilizados pela XILINX, facilitar sua integração em projetos de maior complexidade.
9. Tornar a **ferramenta de inicialização das BRAMs genérica**, para possa ser utilizada em outros projetos.
10. Alterar o CONVWS ou desenvolver um novo módulo com suporte a *stride* 1 e tamanho de filtros parametrizáveis (hoje há suporte apenas para filtro $3 \times 3$). Estas modificações permitirão a avaliação de uma maior número de arquiteturas de CNN.
11. Avaliar a arquitetura de convolução 1D. A arquitetura de convolução 1D é utilizada em aplicações que envolvem processamento de sinais unidimensionais, como processamento de áudio ou séries temporais. Testar essa arquitetura pode ajudar a avaliar sua eficácia e identificar possíveis melhorias para aplicações futuras.
12. Paralelizar os “Cores”. Essa abordagem pode melhorar o desempenho do acelerador, permitindo paralelismo temporal (na forma de um *pipeline*) ou espacial (vários CONVWS operando sobre a mesma camada, mas em diferentes canais).
13. Melhorar o *pipeline* entre os cores para que a vazão aumente, ou seja, desenvolver uma **arquitetura multiciclo.**
14. **Prototipar** a rede completa, ou seja, incluir as camadas adicionais: *Max polling 2D* e *Fully Connected*.
15. Remover *clock* com deslocamento de 180 graus do módulo *memory* das BRAMs (i.e., borda de descida), e alterar todos os módulos para que façam a leitura em borda de subida.

# REFERÊNCIAS BIBLIOGRÁFICAS


[1] Caffe. “Caffe”. Capturado em: https://caffe.berkeleyvision.org/, 2022.

[2] Goodfellow, I.; Bengio, Y.; Courville, A. “Deep Learning”. MIT Press, 2016, 800p, http://www.deeplearningbook.org.

[3] Haykin, S. S. “Neural networks and learning machines”. Pearson Education, 2009, third ed., 906p.

[4] Juracy, L. R. “A Framework for Fast Architecture Exploration of Convolutional Neural Network Accelerators”, Tese de Doutorado, PPGCC-PUCRS, 2022, 137pp.

[5] Juracy, L. R.; Amory, A. M.; Moraes, F. “A Comprehensive Evaluation of Convolutional Hardware Accelerators”, *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 70–3, 2023, pp. 1149–1153.

[6] Juracy, L. R.; Garibotti, R.; Moraes, F. G. “From CNN to DNN Hardware Accelerators: A Survey on Design, Exploration, Simulation, and Frameworks”, *Foundations and Trends® in Electronic Design Automation*, vol. 13–4, 2023, pp. 270–344.

[7] Keras. “Layer activation functions”. Capturado em: https://keras.io/api/layers/activations/, 2022.

[8] Krizhevsky, A.; Nair, V.; Hinton., G. “The CIFAR-10 dataset”. Capturado em: https://www.cs.toronto.edu/~kriz/cifar.html, 2009.

[9] Moolchandani, D.; Kumar, A.; Sarangi, S. R. “Accelerating CNN inference on ASICs: A survey”, *Journal of Systems Architecture*, vol. 113–1, 2021, pp. 1–26.

[10] PyTorch. “PyTorch”. Capturado em: https://pytorch.org/, 2022.

[11] Shawahna, A.; Sait, S. M.; El-Maleh, A. “FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review”, *IEEE Access*, vol. 7, 2019, pp. 7823–7859.

[12] TensorFlow. “TensorFlow”. Capturado em: https://www.tensorflow.org/, 2022.

[13] Xiang, T.; Feng, Y.; Ye, X.; Tan, X.; Li, W.; Zhu, Y.; Wu, M.; Zhang, H.; Fan, D. “Accelerating CNN algorithm with fine-grained dataflow architectures”. In: SmartCity, 2018, pp. 243–251.

[14] Xilinx. “Vivado Design Suite 7 Series FPGA Libraries Guide”. Capturado em: https://docs.xilinx.com/v/u/2012.2-English/ug953-vivado-7series-libraries, 2012.

[15] Xilinx. “7 Series FPGAs Memory Resources User Guide (UG473)”. Capturado em: https://docs.xilinx.com/v/u/en-US/ug473_7Series_Memory_Resources, 2019.

[16] Xilinx. “Block Memory Generator v8.4”. Capturado em: https://docs.xilinx.com/v/u/en-US/pg058-blk-mem-gen, 2021.