Universidade de Brasília

Faculdade do Gama

Desenvolvimento de um Co-Projeto Hardware-Software para uma Rede Convolucional para Estimativa da Frequência Cardíaca Fetal

Gustavo Raspante Faria

TRABALHO DE CONCLUSÃO DE CURSO

ENGENHARIA ELETRÔNICA

Brasília

2024

Universidade de Brasília

Faculdade do Gama

Desenvolvimento de um Co-Projeto Hardware-Software para uma Rede Convolucional para Estimativa da Frequência Cardíaca Fetal

Gustavo Raspante Faria

Monografia submetida ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Orientador: Prof. Dr. Gilmar Silva Beserra

Brasília

2024

FICHA CATALOGRÁFICA

Faria, Gustavo Raspante.

Desenvolvimento de um Co-Projeto Hardware-Software para uma Rede Convolucional para Estimativa da Frequência Cardíaca Fetal / Gustavo Raspante Faria; orientador Gilmar Silva Beserra. -- Brasília, 2024.

80 p.

Trabalho de Conclusão de Curso (Engenharia Eletrônica) -- Universidade de Brasília, 2024.

1. fECG. 2. CNN. 3. VHDL. 4. FPGA. 5. SoC. IP. AXI.I. Beserra, Gilmar Silva, orient. II. Título.

Universidade de Brasília
Faculdade do Gama

Desenvolvimento de um Co-Projeto Hardware-Software para uma Rede Convolucional para Estimativa da Frequência Cardíaca Fetal

Gustavo Raspante Faria

Monografia submetida ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Trabalho aprovado. Brasília, 12 de julho de 2024:

Prof. Dr. Gilmar Silva Beserra
Orientador

Prof. Dr. Daniel Mauricio Muñoz Arboleda
Convidado 1

Prof. Dr. Daniel Chaves Café
Convidado 2

Este trabalho é dedicado aos meus avós, que se foram
e sempre serão lembrados, e a todos que amo.

Agradecimentos

Agradeço a Deus, minha família, minha namorada e meus amigos pelo constante suporte e por me apoiar durante toda a minha jornada. Sou grato a todos os meus professores, pois, ajudaram-me a ser quem eu sou, agradeço principalmente ao prof. Gilmar pela oportunidade.

“There is nothing to be afraid of. Take a gamble that love exists, and do a loving act.”
(Sister Calderón)

Resumo

Atualmente, uma das aplicações das redes neurais artificiais é o seu uso para obter a frequência cardíaca fetal a partir do aECG materno. Dessa forma, é possível realizar a aceleração em hardware da RNA, visando uma melhor eficiência de tempo de execução e de consumo energético. Com isso, o presente projeto consiste em utilizar a abordagem co-projeto *Hardware-Software* na implementação em *FPGA* de uma Rede Neural Convolucional para estimar a frequência cardíaca fetal. Inicialmente, a rede neural completa foi descrita em C e executada como software embarcado no microprocessador ARM de um kit de desenvolvimento Zynq SoC para identificar as camadas de maior tempo de execução. Em seguida, as camadas convolucionais que promoveram maior atraso foram implementadas em VHDL e validadas em *hardware*, obtendo-se um ganho considerável de tempo em relação à implementação feita apenas em *software*. No caso, a segunda, a terceira e a quarta camadas convolucionais da CNN utilizaram uma representação em ponto flutuante de 27 *bits*, de modo que a segunda camada implementada em hardware acelerou em 3,25 vezes com um EQM de 0,0001895; já a terceira camada foi acelerada em 2 vezes com um EQM de 0,0003257 e a quarta foi acelerada em 1,05 vezes com um EQM igual a 2,860758e-12, sendo que a comparação do tempo é com um processador de frequência de *clock* 6,67 vezes maior. A quarta camada convolucional implementada foi escolhida para ser encapsulada em uma IP na interface AXI4-Stream, realizando a comunicação com o bloco de processamento do SoC Zedboard por meio do AXI DMA. Além disso, o bloco IP obtido após o encapsulamento da camada obteve uma performance de 455 MFLOPS/W, mostrando-se cerca de 3 vezes mais eficiente que o próprio ARM Cortex-A9 presente na Zedboard.

Palavras-chave: fECG. CNN. VHDL. FPGA. SoC. IP. AXI.

Abstract

Currently, one of the applications of artificial neural networks is using them to obtain the fetal heart rate from maternal aECG. This allows for hardware acceleration of the ANN, aiming for better execution time efficiency and energy consumption. This project focuses on using a Hardware-Software co-design approach to implement a Convolutional Neural Network on FPGA to estimate the fetal heart rate. Initially, the complete neural network was described in C and executed as embedded software on the ARM microprocessor of a Zynq SoC development kit to identify the layers with the highest execution time. Subsequently, the convolutional layers that caused the most delay were implemented in VHDL and validated in hardware, achieving a considerable time gain compared to the software-only implementation. Specifically, the second, third, and fourth convolutional layers of the CNN used a 27-bit floating-point representation, with the second layer implemented in hardware accelerating by 3.25 times with an MSE of 0.0001895, the third layer accelerating by 2 times with an MSE of 0.0003257, and the fourth layer accelerating by 1.05 times with an MSE of 2.860758e-12, compared to a processor with a clock frequency 6.67 times higher. The fourth convolutional layer was chosen to be encapsulated into an IP core on the AXI4-Stream interface, communicating with the processing block of the Zedboard SoC via AXI DMA. Additionally, the IP core obtained after encapsulating the layer achieved a performance of 455 MFLOPS/W, making it approximately 3 times more efficient than the ARM Cortex-A9 present on the Zedboard.

Keywords: fECG. CNN. VHDL. FPGA. SoC. IP. AXI.

Lista de figuras

Lista de tabelas

Lista de abreviaturas e siglas

aECG	Eletrocardiograma Abdominal
ASIC	*Application Specific Integrated Circuits*
AXI	*Advanced eXtensible Interface*
BPM	Batimento Por Minuto
BRAM	*Block Random Access Memory*
CLB	*Configurable Logic Blocks*
CNN	*Convolutional Neural Networks* (Redes Neurais Convolucionais)
CPU	*Central Processing Unit*
DMA	*Direct Memory Access*
ECG	Eletrocardiograma
EQM	Erro Quadrático Médio
fECG	Eletrocardiograma Fetal
FGA	Faculdade do Gama da Universidade de Brasília
FHR	*Fetal Heart Rate* (Frequência Cardíaca Fetal)
FPGA	*Field Programmable Gate Array*
FSM	*Finite State Machine*
GPU	*Graphics Processing Unit*
HDL	*Hardware Description Language*
HLS	*High-Level Synthesis*
IOB	*In/Out Block*
IP	*Intellectual Property*
LMS	*Least Mean Squares*
mECG	Eletrocardiograma Materno
MFLOPS	Milhões de Operações em Ponto Flutuante por Segundo
NI-fECG	*Non Invasive fetal ECG* (Eletrocardiografia Fetal Não Invasiva)
PL	*Programmable Logic*
PS	*Processing System*
RNA	Redes Neurais Artificiais
SB	*Switch Box*
SoC	*System on a Chip*
SVD	*Singular Value Decomposition*
UnB	Universidade de Brasília
VHDL	*VHSIC Hardware Description Language*

Sumário

1 Introdução

1.1 Contextualização

As redes neurais artificiais seguem sendo cada vez mais exploradas na área da saúde, permitindo ser um modelo para o monitoramento da frequência cardíaca fetal (FHR - *Fetal Heart Rate*) (Kahankova *et al.*, 2020). A FHR é a quantidade de batimentos cardíacos do feto por minuto, existem diversas maneiras de a observar, o primeiro método é conhecido há séculos, no caso, a auscultação, ou seja, a escuta dos batimentos (Solt; Divon, 2004). No século XX, houve o desenvolvimento de diversos métodos invasivos ou não-invasivos. Os invasivos são realizados internamente em contato com o útero, por exemplo o eletrodo de escalpe fetal (Hunter *et al.*, 1964). Enquanto que os não-invasivos são realizados externamente no abdômen.

Em 1906, pela primeira vez, foi observado o eletrocardiograma fetal (fECG) e com a análise do sinal se observa a FHR (Sameni; Clifford, 2010). Com os avanços tecnológicos, em 1953 foi feito a eletrocardiografia fetal não invasiva (NI-fECG), mostrando-se um método de monitoramento contínuo extremamente promissor. Logo, ocorreu um salto na qualidade do acompanhamento da saúde do feto e a detecção de doenças cardíacas (Rafie; Kashou; Noseworthy, 2021).

Atualmente, existem diversos métodos de extração do NI-fECG, podendo utilizar somente eletrodos abdominais (*Abdominal Electrodes Sourced - AES*) ou em conjunto com eletrodos no peito da mãe (*Combined Source - CS*), como mostra a Figura 1.1, permitindo que seja obtido o fECG com algoritmos baseados em filtro de Kalman, transformada *wavelet*, filtro por mínimos quadrados (LMS - *least mean squares*), decomposição em valores singulares (SVD - *singular value decomposition*) e redes neurais artificiais (Kahankova *et al.*, 2020).

O método das redes neurais artificiais (RNAs) se baseia em dados fornecidos pelos eletrodos abdominais em conjunto com eletrodos no peito entregando o eletrocardiograma abdominal (aECG) em conjunto com o eletrocardiograma materno (mECG). Desse modo a RNA processa esses sinais para que seja observado apenas o fECG, permitindo a análise da FHR (Kahankova *et al.*, 2020).

Na Faculdade do Gama da Universidade de Brasília (FGA-UnB), um protótipo para a obtenção da FHR a partir do NI-fECG está em desenvolvimento. Baseando-se em implementar a estrutura de extração, em um dispositivo reconfigurável, ou seja, uma FPGA (*Field Programmable Gate Array*) permitindo a aceleração do algoritmo. O protótipo é divido em três partes:

- aquisição - coleta de amostras reais (Tutida, 2016);
- processamento - extração do fECG podendo ser por filtro adaptativo (Barbosa, 2016) ou por RNAs (Junior, 2018);
- comunicação - envio de dados para um dispositivo móvel via *Bluetooth* (Rodrigues, 2016).

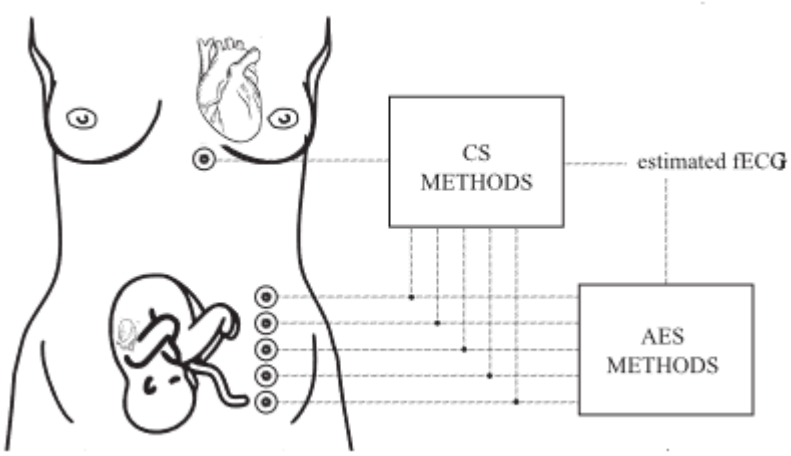


Figura 1.1 – Coleta do aECG e mECG. FONTE: (Kahankova *et al.*, 2020)

1.2 Objetivos

1.2.1 Objetivos Gerais

Acelerar o algoritmo da RNA proposta por (Junior, 2018) para diversas camadas convolucionais, realizando a execução do co-projeto *hardware-software* das camadas convolucionais implementadas neste trabalho e por (Andrade, 2022) com o restante da rede neural.

1.2.2 Objetivos Específicos

Em vista de atingir os objetivos gerais, devem ser realizados os seguintes objetivos específicos:

- Implementar o bloco IP de uma das camadas convolucionais da rede neural proposta.
- Analisar os melhores métodos de comunicação para a estrutura proposta.
- Realizar a comunicação de pelo menos uma camada convolucional implementada em *hardware* com a RNA.
- Descrever em VHDL as restantes camadas convolucionais e testá-las.

1.3 Estrutura do Trabalho

O presente trabalho tem em sua composição 5 capítulos. O Capítulo 1 é a Introdução, onde são apresentados a contextualização, objetivos do trabalho e metodologia. O Capítulo 2 é a fundamentação teórica, compreendendo os conceitos necessários para a realização dos objetivos, como: o desenvolvimento cardíaco fetal e o impacto na FHR, entendimento sobre Redes Neurais Artificiais, estrutura e funcionamento de uma FPGA, e sobre SoC aplicado em aceleração de algoritmos. No Capítulo 3, é demonstrado a descrição e a implementação da arquitetura, além da comunicação entre *hardware* e *software*. O Capítulo 4 apresenta os resultados do projeto. O Capítulo 5 apresenta uma conclusão, e dá sugestões para futuros trabalhos.

2 Referencial Teórico

2.1 Frequência Cardíaca Fetal

2.1.1 Desenvolvimento cardíaco inicial e sua atividade

Na terceira semana de gravidez, ocorre a diferenciação dos tecidos e órgãos, incluindo o coração junto ao sistema cardiovascular. Nessa fase inicial, os vasos sanguíneos são formados pelos processos de vasculogênese e angiogênese, o coração ainda é uma forma primitiva, chamada de tubo cardíaco primitivo ou coração tubular, formada por quatro partes (átrio primitivo, ventrículo, bulbo cardíaco e saco aórtico), pode-se observar, na Figura 2.1, o processo de formação do coração. Quando o tubo cardíaco primitivo se conecta aos vasos sanguíneos, o sistema cardiovascular primitivo se forma, sendo o primeiro sistema de órgãos que chega a um estado funcional, de maneira que já na quinta semana é possível detectar os batimentos cardíacos fetais (Moore; Persaud, 2008).

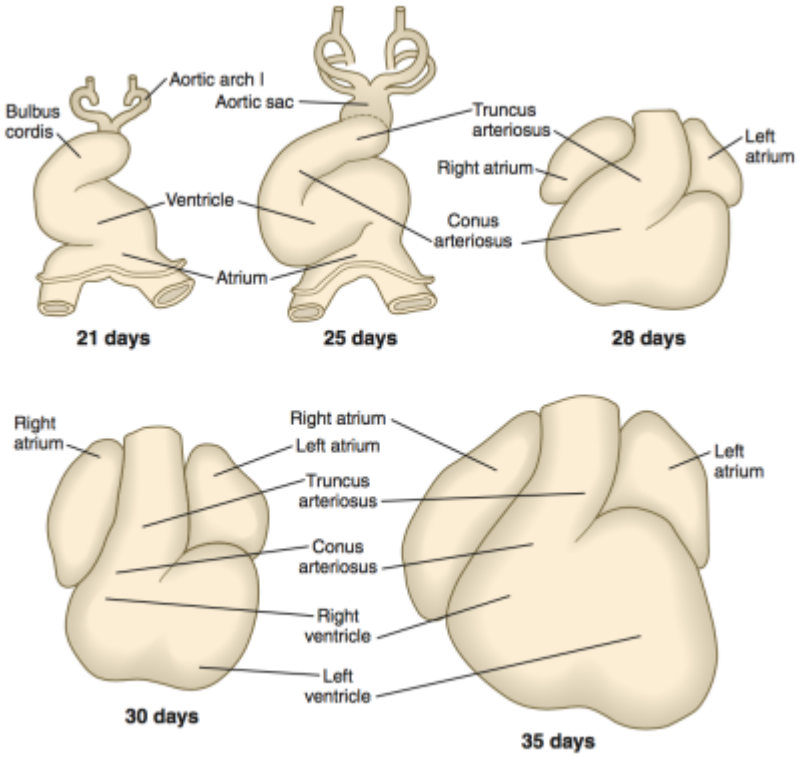


Figura 2.1 – Formação do coração fetal. FONTE: (Carlson, 2013)

Segundo (Sameni; Clifford, 2010), a partir da vigésima semana é possível escutar o batimento do coração fetal utilizando o método da auscultação, porém não se obtém uma quantidade de informação detalhada para um diagnóstico, por isso se faz necessário outros

métodos de obtenção da FHR. Além disso, a frequência de batimentos um feto saudável da vigésima semana até seu nascimento deve estar dentro da faixa de 120BPM até 160BPM (Steinburg *et al.*, 2013). Enfatizando a importância do acompanhamento durante a gravidez e durante o parto.

2.1.2 FHR

A FHR (*Fetal Heart Rate*) ou frequência cardíaca fetal indica a quantia de batimentos por minuto. Com isso, vem sendo utilizada como uma fonte de informação importante para o acompanhamento e diagnóstico da saúde do feto (Hasan; Ibrahimy; Reaz, 2009), pois doenças cardíacas causadas pela má formação do coração está entre as mais comuns de ocorrerem entre os fetos, um a cada 125 sofrem desse problema (Sameni; Clifford, 2010). Sua faixa de valores normais é de 120-160BPM, além disso sofre uma redução de 0,4BPM a cada semana durante o período de gestação (Abdulhay *et al.*, 2014).

Há diversas maneiras de observar a frequência cardíaca do feto, a mais simples, por exemplo é a auscultação. Os métodos para obtê-la são divididos em invasivos e não-invasivos, os invasivos consistem em coletar os dados da maneira interna, como cateter intrauterino e o eletrodo de escalpe fetal, mostrado na Figura 2.2. Os não-invasivos coletam os dados externamente, ou seja, no abdômen da mãe, por exemplo a cardiotocografia, que utiliza do efeito Doppler para obter os dados, e a utilização de eletrodos, como mostrado na Figura 1.1, para se extrair os sinais ECG (Kahankova *et al.*, 2020).

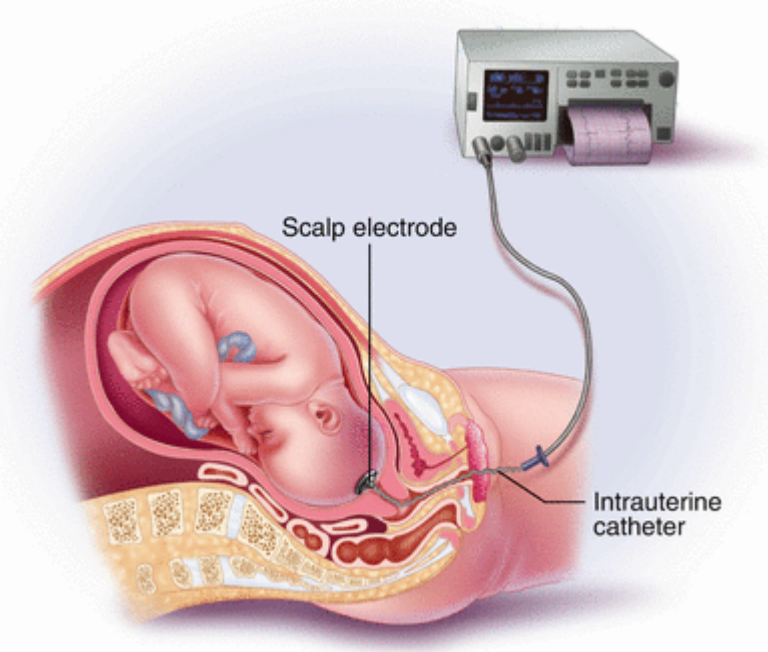


Figura 2.2 – Eletrodo de escalpe fetal. FONTE: (Lisenbee; Tyndall, 2016)

2.1.3 Eletrocardiograma Fetal (FECG)

O eletrocardiograma fetal corresponde a atividade elétrica do coração, de maneira que podemos observar seu ciclo. Na Figura 2.3, temos o ciclo cardíaco e suas fases, a onda P representa a despolarização atrial, indicando que o sangue está entrando nos ventrículos. Após isso, o complexo QRS demonstra a despolarização ventricular, junto disso ocorre a repolarização atrial, porém não é observada no gráfico por ser um processo de menor tensão. Por fim, temos o relaxamento ventricular, ou seja, a repolarização ventricular mostrada pela onda T (Sameni; Clifford, 2010). De maneira que, a FHR pode ser obtida analisando o intervalo entre dois picos R seguidos (Smith *et al.*, 2018).

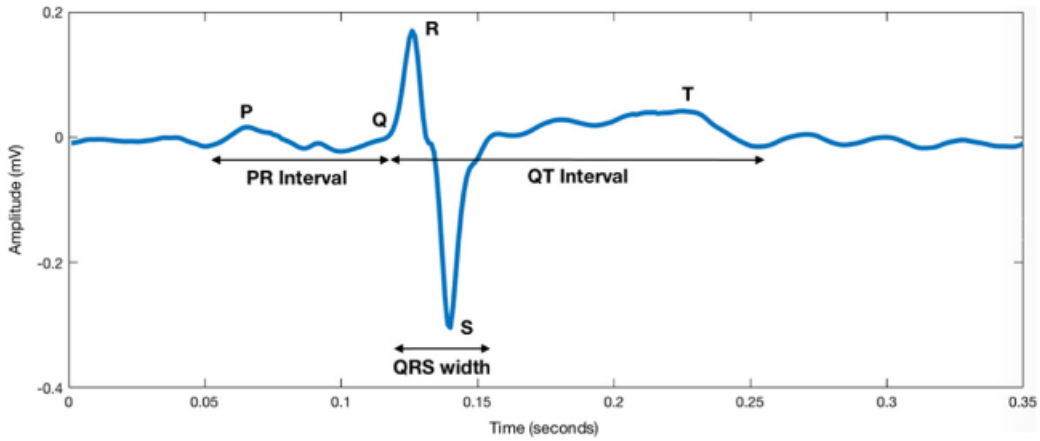


Figura 2.3 – Eletrocardiograma. FONTE: (Smith *et al.*, 2018)

Uma maneira de se obter o ECG fetal que se estabelece cada vez mais no meio da saúde é a não invasiva, a NI-fECG (eletrocardiografia fetal não invasiva) (Smith *et al.*, 2018). A sua obtenção se dá pelo posicionamento de eletrodos no abdômen e, em alguns casos, no tórax da gestante, como mostra a figura1.1. Com isso, podem ser obtidos o eletrocardiograma abdominal (aECG) em conjunto com o eletrocardiograma materno (mECG), o aECG é um sinal que se compõe por outros três sinais: o fECG, o mECG e o ruído, como mostrado na figura2.4. Portanto, a obtenção do fECG, consiste em filtrar o mECG e o ruído do aECG, para isso existem diversos métodos como: filtragem, transformada *wavelet*, decomposição em valores singulares (SVD - *singular value decomposition*) e redes neurais artificiais (Kahankova *et al.*, 2020).

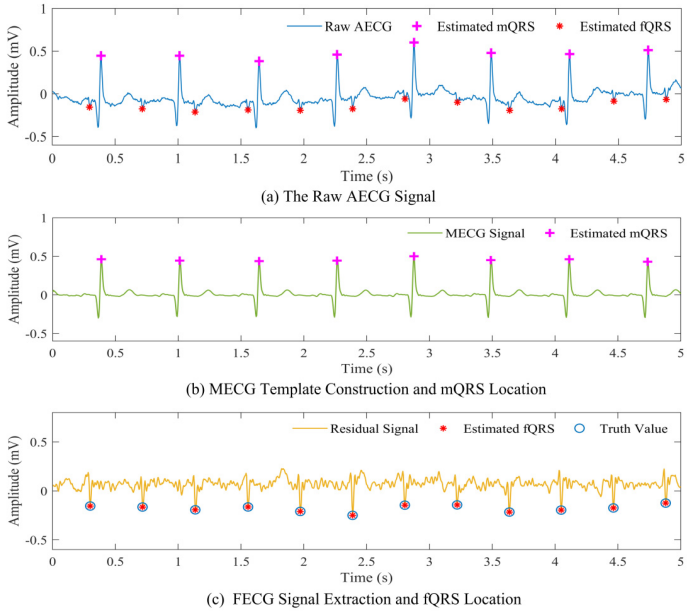


Figura 2.4 – Sinal aECG e sua composição. FONTE: (Zhang *et al.*, 2022)

2.2 Redes Neurais Artificias

2.2.1 Definição

Uma rede neural é um sistema feito para aproximar o modo como o cérebro realiza alguma tarefa (Haykin, 1999).As redes neurais artificiais surgiram a partir de estudos e modelagens matemáticas dos neurônios biológicos, principalmente por McCullock e Pitts em 1943. Os neurônios são compostos por três partes principais: o corpo celular (ou soma), os dendritos e o axônio, como mostrado na Figura 2.5. Os pulsos elétricos (impulsos nervosos) chegam pelo axônio, fornecendo informação ao neurônio (Kovacs, 1996).

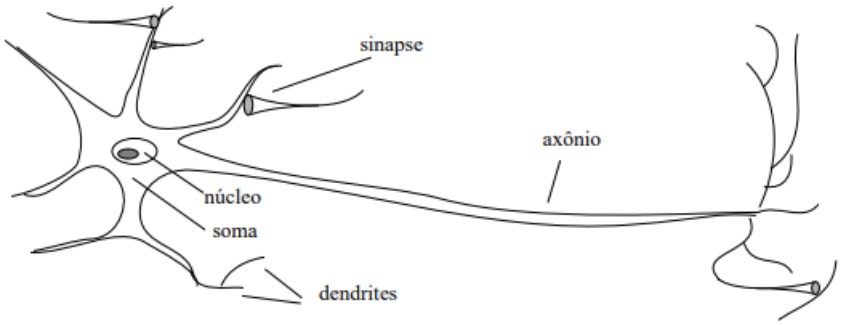


Figura 2.5 – Neurônio biológico. FONTE: (Rauber, 2005)

Segundo os estudos de McCullock e Pitts, a atividade do neurônio é tudo ou nada e a atividade de qualquer sinapse inibitória previne a excitação do neurônio naquele instante, com isso pode-se concluir que neurônio estará no estado ativado apenas se a sua saída

ultrapassar um valor limite, e cada entrada do neurônio terá um valor associado. Essas conclusões foram de suma importância para que ocorresse a implementação computacional do neurônio, portanto o modelo de McCullock e Pitts para um neurônio artificial seguem as equações 2.1 e 2.2, como mostrado na Figura 2.6 (Cardon; Müller, 1994).

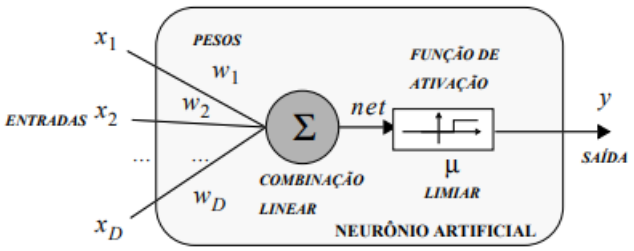


Figura 2.6 – Neurônio artificial. FONTE: (Rauber, 2005)

$$\mu_k = \sum_{m}^{j=1} w_{kj} x_j \tag{2.1}$$

$$y_k = \varphi(\mu_k + b_k) \tag{2.2}$$

Com isso, as redes neurais artificiais, segundo (Haykin, 1999), são compostas por vários neurônios, que são unidades de processamento simples, e funcionam como um processador distribuído de maneira paralela, tendo a propensão natural para armazenar conhecimento experimental e torná-lo disponível para o uso. Além disso, assemelha-se ao cérebro em dois aspectos: o conhecimento é adquirido pela rede a partir de seu ambiente através de um processo de aprendizagem; forças de conexão entre neurônios, conhecidas como pesos sinápticos, são utilizadas para armazenar o conhecimento adquirido.

2.2.2 Redes Neurais Artificiais Simples

As RNAs podem ser vista como um grafo orientado, pois contém arestas, simbolizando conexões, entre nodos (elementos de processamento) com um só sentido, como mostrado na Figura 2.7. Portanto a informação flui pelas arestas e é coletada pelos nodos. No caso, pesos e conexões positivos (possuem um valor numérico positivo) são excitatórios e aqueles que são negativos são inibitórios (Cardon; Müller, 1994).

As RNAs simples podem ser de uma única camada e são conhecidas como rede *perceptron*, em que as entradas recebem seus respectivos pesos, seguindo as equações 2.1 e 2.2. Além disso, a saída obedece uma função degrau mostrada na equação 2.3. Enquanto que as RNAs simples compostas por várias camadas são as redes *Multi Layer Perceptron*, que podem ter como função de ativação ou a função degrau ou a função sigmoidal, mostrada na equação 2.4, sendo a o parâmetro de inclinação da curva (Haykin, 1999).

$$f(\mu) = \begin{cases} 1, & \text{if } x + b \geq 0 \\ 0, & \text{else} \end{cases} \tag{2.3}$$

$$f(\mu) = \frac{1}{1 + e^{-a\mu}} \tag{2.4}$$

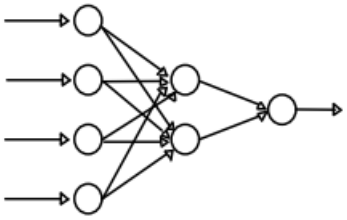


Figura 2.7 – Grafo orientado como RNA. FONTE: (Cardon; Müller, 1994)

2.2.3 Redes Neurais Convolucionais

As Redes Neurais Convolucionais ou CNN (*Convolutional Neural Network*) são arquiteturas inspiradas na biologia com capacidade de serem treinadas e que conseguem reconhecer características. Compostas por múltiplos estágios, suas entradas e saídas são matrizes de uma, duas ou três dimensões dependendo do tipo de informação usada, conhecidas como mapas de características (*feature maps*). Na saída, tem-se representação por matrizes com as características extraídas das entradas. Esse processo é composto por estágios com três camadas cada: uma de banco de filtros, que aplica a convolução e cada filtro detecta uma certa característica, uma camada de não linearidade, que consiste em aplicar uma função sigmoide na matriz, uma camada de agrupamento (*pooling*), que trata separadamente cada característica (*feature*). Um exemplo típico de uma CNN se encontra na Figura 2.8 (LeCun; Kavukcuoglu; Farabet, 2010).

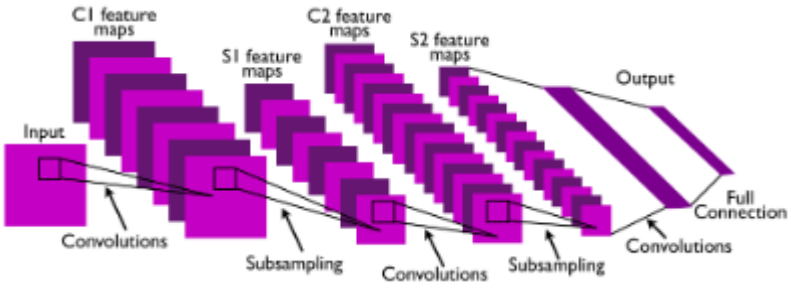


Figura 2.8 – Arquitetura de uma CNN de dois estágios. FONTE: (LeCun; Kavukcuoglu; Farabet, 2010)

2.2.3.1 *Backpropagation*

Como dito anteriormente, são arquiteturas treináveis e para realizar esse processo existem certos tipos de métodos, inclusive o de *backpropagtion*, ou seja, de retropropagação de erro. Esse método é separado em duas fases: propagação e retropopagação. Na propagação, é inserido o vetor de entrada, de modo que os pesos da rede são todos fixos, gerando um conjunto de saídas. Na retropopagação, os pesos são ajustados de acordo com uma regra de correção de erro, que gerada pela comparação entre a saída obtida com a saída desejada (Haykin, 1999).

Além disso, o algoritmo de retropropagação de erro é considerado como um método de aprendizado baseado em gradiente. Sendo que, pode ser representado com uma função de saída com as entradas e os pesos (2.5), uma função de custo (2.6), e a média das funções de custo (2.7), conforme a Figura 2.9.

$$M(Z^n, W), \tag{2.5}$$

em que, Z^n é a n-ésima amostra de entrada, W representa o conjunto de pesos ajustáveis e M é a saída da rede.

$$E^n = C(D^n, M(Z^n, W)), \tag{2.6}$$

E^n é a diferença entre o valor real (D^n) e o valor gerado pela rede neural ($M(Z^n, W)$).

$$E(W) = \frac{1}{N} \sum_{n=1} E^n, \tag{2.7}$$

de forma que, $E(W)$ representa a média das funções de custo e N e quantidade total de amostras de entradas.

Logo, a cada iteração o conjunto dos pesos W é ajustado a fim de reduzir o máximo possível da função E(W) (LeCun *et al.*, 2012).

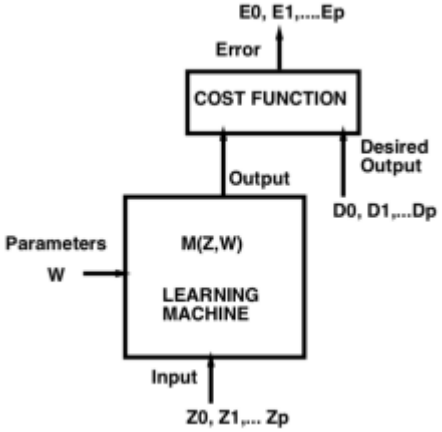


Figura 2.9 – Funcionamento do algoritmo *backpropagation*. FONTE: (LeCun *et al.*, 2012)

2.3 Aceleração de Hardware

A aceleração de hardware pode ser realizada através de diversas ferramentas como o CPUs (Central Processing Units), GPUs (Graphics Processing Units), FPGAs e ASICs (Application Specific Integrated Circuits). Implementando algoritmos voltados a serem aplicados em um hardware. O FPGA é fornece melhoras na performance, além de um bom consumo de energia por sua arquitetura flexível e paralela, permitindo ajustar os recursos para realizar as mudanças de requisitos como energia, performance e tolerância a falhas (Berbert; Bertini; Copetti, 2021).

No caso das redes neurais, o uso de FPGAs também é comum, pois são reconfiguráveis e reprogramáveis, facilitando a implementação de projetos. Além de permitirem uma alta customização de seus parâmetros, garantindo uma eficiência energética superior a das GPUs, tornando os FPGAs mais atrativas para pesquisas e projetos (Santana, 2020). Podendo serem utilizadas para acelerar operações matriciais das RNAs, como é mostrado em (TOMASI, 2020).

2.3.1 Definição sobre FPGAs

Field-Programmable Gate Array, comumente chamado por FPGA, é um circuito integrado, que formam blocos configuráveis ou reconfiguráveis com interconexões entre si também configuráveis. Ou seja, um componente composto por matrizes de blocos lógicos configuráveis (CLB - *Configurable Logic Blocks*) com suas interconexões sendo configuráveis, portanto são capazes de realizar diversas tarefas (Maxfield, 2004).

Na Figura 2.10, pode-se observar a arquitetura da família de FPGAs Artix-7, presente na placa Basys 3 e no SoC ZedBoard, composta pelos seguintes blocos fundamentais:

- CLB (*Configurable Logic Block*): matrizes de blocos lógicos configuráveis, compostos por *flip-flops*, que são registradores síncronos, e LUTs (*Look-up tables*), sendo esses feitos de multiplexadores capazes de implementar funções booleanas.
- SB (*Switch Box*): interconexões dos CLBs.
- BRAM (*Block Random Access Memory*): bloco de memória de acesso randômico composto por uma RAM de duas portas.
- IOB (*In/Out Block*): bloco de entrada e saída.
- CMT (*Clock Management Tile*): o bloco de gerenciamento de relógio, contém um MMCMs (*Mixed-Mode Clock Managers*) (MMCMs) e um PLL (*Phase Locked Loop*). Atua como um sintetizador de frequência, corrige o atraso de propagação do sinal de *clock*, e admite uma gama de diversas frequências como entrada.
- FIFO Logic (*First In First Out*): interface de memória que segue a lógica FIFO.
- BUFG (*Global Clock Buffer*): é um *clock buffer* com uma entrada e uma saída.

- DSP (*Digital Signal Processing*): blocos de processamento de sinal digital.
- BUFIO (*Clock Buffer for I/O*): um *buffer* local de *clock* que adentra o bloco de I/O.
- BUFR (*Regional Clock Buffer*): é um *buffer* de *clock* regional que direciona sinais de *clock* para uma rede de *clock* dedicada independente do *clock* global.
- MGT (*Multi-Gigabit Transceiver*): um serializador/desserializador capaz de operar em taxas de bits seriais acima de 1 Gigabit/segundo.

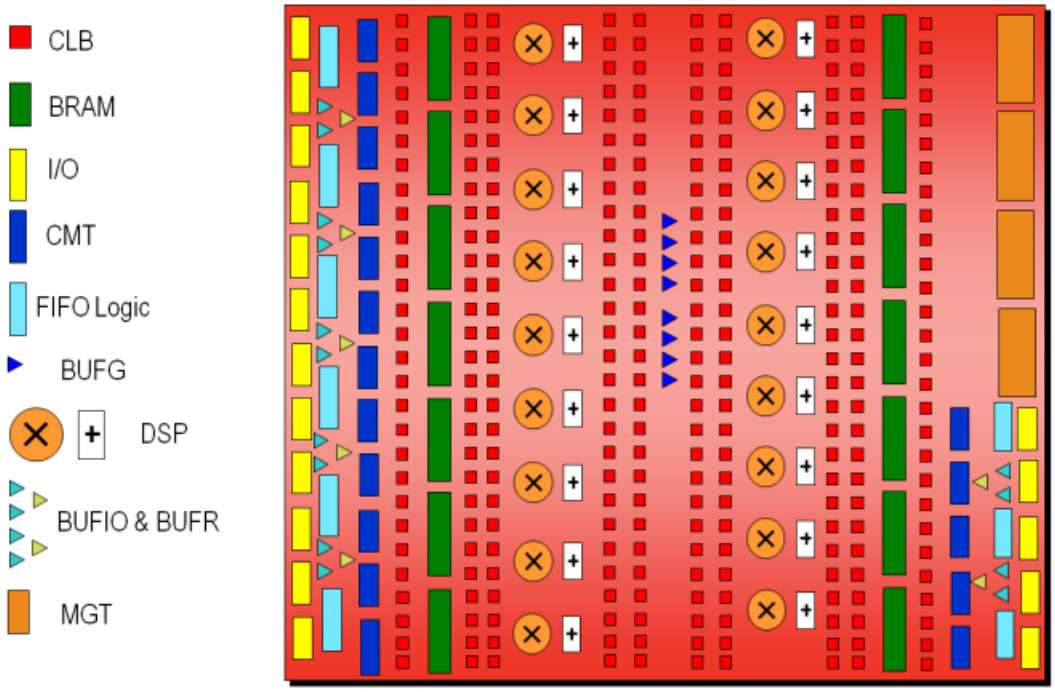


Figura 2.10 – Arquitetura da família de FPGA Artix-7. FONTE: (XILINX, 2013)

As matrizes de CLBs e suas conexões podem ser configuradas utilizando a HDL (*Hardware Description Language*) para implementar os circuitos lógicos desejados. De acordo com (Berbert; Bertini; Copetti, 2021), a linguagem HDL possui uma descrição textual composta de operadores, expressões, estruturas e operações de entrada e saída, gerando um mapa dessas portas, conhecido por *bitstream*.

2.3.2 *System On a Chip* (SoC)

Os SoCs são circuitos integrados com componentes de um computador, como CPU, memória, interfaces de I/O (USB, Ethernet, etc) e interfaces de comunicação (Wi-Fi, Blueto-oth, etc), além disso podem conter uma FPGA. No presente escopo, para realizar o co-projeto Hardware-Software foi utilizada uma placa de desenvolvimento ZedBoard, que contém uma Zynq-7000 SoC XC7Z020.

A ZedBoard pode ser dividida em *Processing System* (PS) e *Programmable Logic* (PL). O PS é composto por um CPU Dual-core ARM Cortex-A9, memória (RAM, ROM e cache),

interface externa de memória, controlador de Acesso Direto à Memória (DMA), interfaces e periféricos de entrada e saída, interconexão baseada em AXI entre os blocos PS e PL. Já o PL, no caso da XC7Z020, uma FPGA equivalente a família Artix-7, como mostra a Figura 2.11 (AMD, 2023).

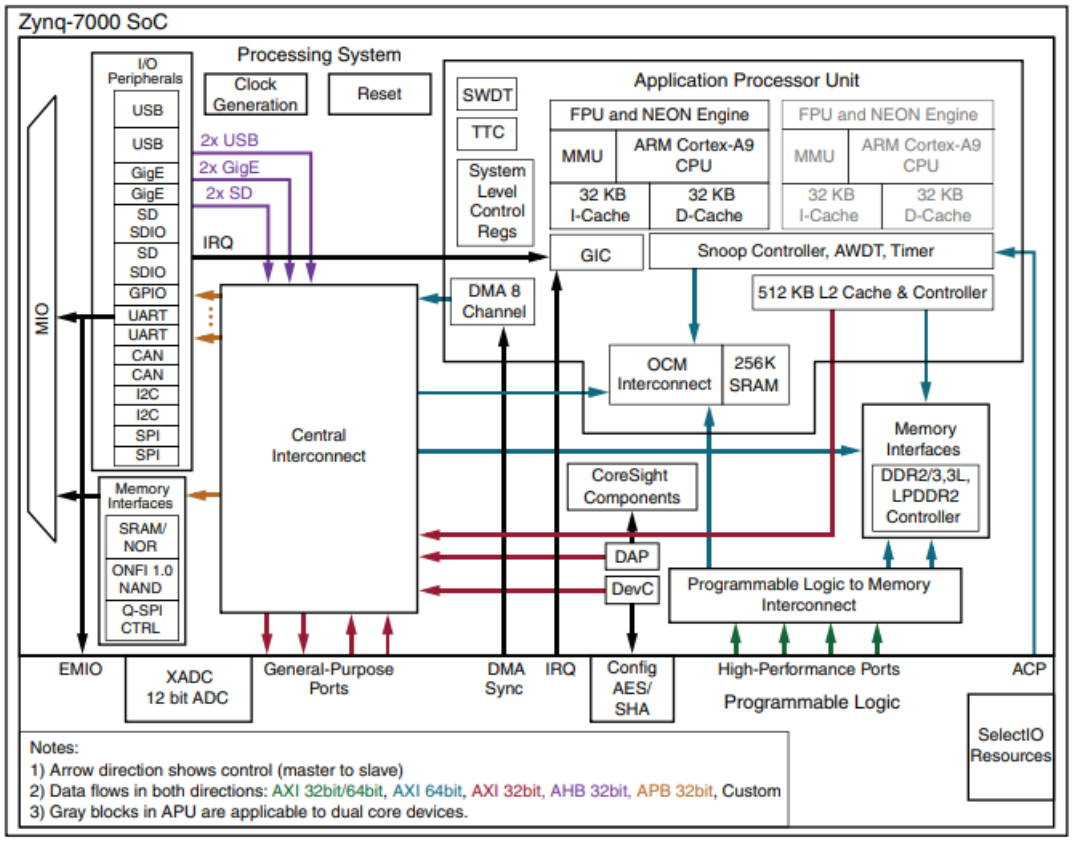


Figura 2.11 – Arquitetura da família de SoC Zynq-7000. FONTE: (AMD, 2023)

2.3.3 Protocolo *AXI*

O protocolo AXI faz parte de uma família de arquiteturas de barramento avançado de microcontrolador introduzida em 1996 utilizada na arquitetura ARM. Além disso, oferece benefícios como: padronizando para a interface *AXI* se faz necessário saber apenas um protocolo para a propriedade intelectual (*IP*); funcional em diferentes aplicações; acesso a uma comunidade global de parceiros da ARM (XILINX, 2017).

Segundo (ARM, 2011), as características mais mais marcantes do protocolo AXI são:

- adequado para designs de alta largura de banda e baixa latência;
- oferece operação em alta frequência sem a necessidade de usar pontes complexas;
- atende aos requisitos de interface de uma ampla gama de componentes;
- adequado para controladores de memória com alta latência de acesso inicial;
- proporciona flexibilidade na implementação de arquiteturas de interconexão;

- fases separadas de endereço/controle e dados;
- utiliza transações baseadas em rajadas (*bursts*), com apenas o endereço de início emitido;
- canais de dados de leitura e escrita separados, de modo que não seja custoso o uso do *Direct Memory Access*.

A arquitetura AXI contém canais de transação independentes baseados em operações em rajada, esses canais são: endereço de leitura, dado da leitura, endereço de escrita, dados de escrita e resposta de escrita. De modo que, um canal de endereço envia informações de controle que descrevem a natureza dos dados a serem transferidos, já os dados são transferidos entre mestre e escravo usando: um canal de dados de escrita para transmitir dados do mestre para o escravo, sendo que para uma operação de escrita, o escravo usa o canal de resposta de escrita para notificar a conclusão da transferência ao mestre, como mostra a Figura 2.11a; para a operação de leitura, a Figura 2.11b mostra que se utiliza um canal de dados de leitura para transferir dados do escravo para o mestre.

(a) Transação de escrita. (b) Transação de leitura.

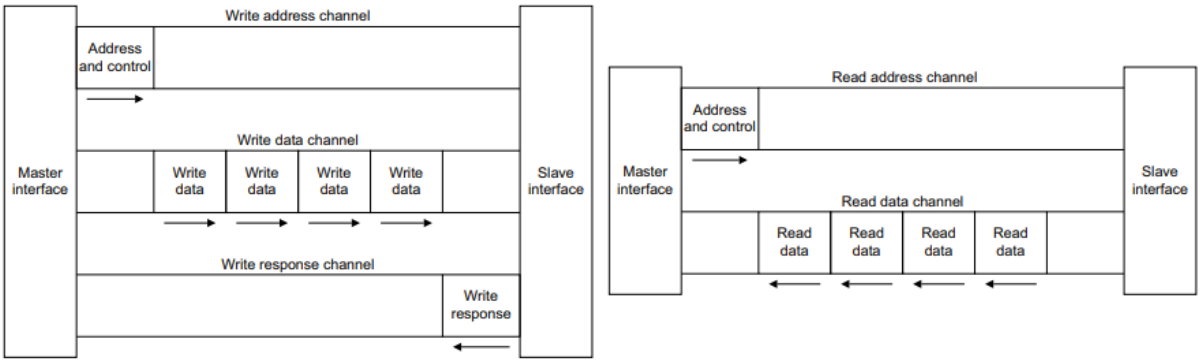


Figura 2.12 – Transação de leitura e escrita na arquitetura AXI. Fonte: (ARM, 2011).

2.3.3.1 *AXI4*

O *AXI*4 é uma interface de comunicação, permite alta performance em transações com mapeamento de memória, permite a transferência de até 256 unidades de dados (como palavras de 32 ou 64 bits) em uma única rajada. Além disso, suporta operações em *pipeline*, permitindo que múltiplas transações sejam iniciadas em paralelo, suporta uma largura de barramento de dados de 32 *bits* ou 64 *bits*, por fim, contém os sinais para controle de qualidade de serviço.

Os sinais do protocolo *AXI4* são mostrados no AnexoA.

2.3.3.2 *AXI4-Lite*

O *AXI4-Lite* é uma das interfaces do protocolo *AXI*, utilizada para comunicação simples e de baixa taxa de transferência mapeada em memória, sendo uma interface mapeada em memória leve e de única transação e simples de trabalhar tanto no design quanto na aplicação (XILINX, 2017). Como suas principais características temos: todas as transações têm comprimento de rajada 1; todos os acessos de dados utilizam a largura total do barramento de dados; o *AXI4-Lite* suporta uma largura de barramento de dados de 32 *bits* ou 64 *bits* (ARM, 2011).

Os sinais do protocolo *AXI4-Lite* são mostrados no AnexoB.

2.3.3.3 *AXI4-Stream*

O *AXI4-Stream* permite uma transferência de dados ainda mais rápida, pois ao contrário do AXI4 tradicional, onde cada transferência de dados precisa especificar um endereço, o *AXI4-Stream* simplesmente transfere os dados sem se preocupar com endereços. Também, permite realizar *burst* de tamanho ilimitado, possibilitando a transferência de grandes volumes de dados de forma contínua. De forma que, utiliza o método do *handshake* para indicar transações válidas, ou seja, quando os sinais *ready* e *valid* são verdadeiros.

Os principais sinais do protocolo *AXI4-Stream* são descritos da seguinte forma:

- AXIS_TVALID: indica que a transação é válida;
- AXIS_TREADY: indica que o bloco está pronto para receber (*slave*) os dados do mestre;
- AXIS_TDATA: quantidade de bits transferidos por *clock*;
- AXIS_TLAST: indica quando o último byte está sendo transferido (AMD, 2024).

2.3.4 *AXI Direct Memory Access* (DMA) *IP Core*

O *AXI DMA IP core* é um bloco de propriedade intelectual utilizado para fornecer acesso direto à memória de alta largura de banda entre interfaces mapeadas em memória *AXI4* e interfaces *AXI4-Stream*, sendo capaz mover grandes quantidades de dados rapidamente (AMD, 2022).

Além disso, permite utilizar de maneira opcional o modo *Scatter/Gather* (SG), que é capaz de realizar transferências de dados complexas e não contíguas e de maneira autônoma, retirando a necessidade de intervenção contínua da CPU. Sem utilizá-lo, o *AXI DMA* realiza transações com uma performance menor, entretanto consome menos recursos. Dessa forma, as transferências são controladas escolhendo um endereço de origem para transferências de memória para *stream* (*MM2S*) ou um endereço de destino para transferências de *stream* para memória (*S2MM*) e especificando a quantidade de bytes a ser transferida. Enquanto

que, os registros usados para inicializar, monitorar e gerenciar o *DMA* são acessados através de uma interface AXI4-Lite, como se observa na Figura 2.13 (AMD, 2022).

No presente trabalho, foi utilizado sem o modo SG, pois ainda permite:

- Suporte primário de largura de dados *AXI4* de 32, 64, 128, 256, 512 e 1.024 bits;
- Suporte primário de largura de dados *AXI4-Stream* de 8, 16, 32, 64, 128, 256, 512 e 1.024 bits;
- Motor opcional de realinhamento de dados para uma largura de dados de *stream* de até 512 bits;
- Permite o realinhamento de dados ao nível de byte (8 bits) nos caminhos de dados de mapa de memória primário e *stream*;
- Fluxos opcionais de Controle e Status *AXI* para interface com *AXI Ethernet IP*;
- Fornece um fluxo de controle opcional para o canal MM2S e um fluxo de status para o canal S2MM para descarregar controle e status de baixa largura de banda do caminho de dados de alta largura de banda;
- Modo Micro opcional, configurado para fornecer um IP de baixo consumo e baixo desempenho (AMD, 2022).

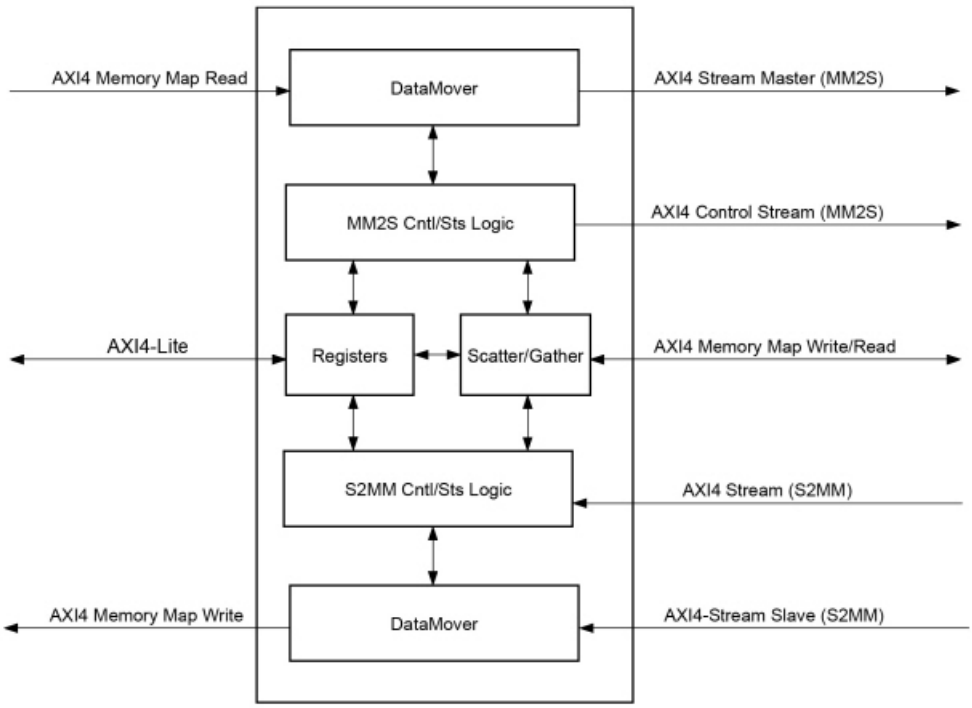


Figura 2.13 – Diagrama de Bloco do *AXI DMA*. FONTE: (AMD, 2022)

3 Desenvolvimento da Arquitetura

3.1 Metodologia

Primeiramente, foi feita uma pesquisa para aprofundamento teórico para compreender sobre a FHR, implicando em estudar o desenvolvimento do coração no feto e como a frequência de batimento é alterada ao longo do tempo, além de entender como analisar um sinal de eletrocardiograma. Ademais, uma pesquisa relacionada a RNAs e meios de implementá-las, seguindo o método proposto para a Rede Neural Convolucional por (Andrade, 2022). Portanto, pesquisando sobre FPGAs e SoCs, para realizar a aceleração em *hardware* da CNN.

A implementação da rede convolucional consistiu em replicar o trabalho feito por (Andrade, 2022) para melhor compreensão da arquitetura e sua implementação em VHDL. Posteriormente, instanciar IPs das camadas convolucionais e utilizar o protocolo AXI para a comunicação entre uma ou mais camadas e o restante da RNA implementada em *software*.

A realização dos testes iniciais se baseou em gerar sinais simulados de ECG no Octave e converter esses valores para binário e usá-los nos *testbenches* das camadas convolucionais realizados no Vivado. Após as simulações iniciais, foram realizadas validações em *hardware* dos blocos internos das camadas utilizando a ferramenta *ILA Core*.

Além disso, as camadas foram implementadas de maneira completa. Encapsulado em IP a quarta camada convolucional para realizar a comunicação entre o bloco PS e PL da *ZedBoard* com o protocolo *AXI4-Stream* junto do *AXI DMA*. A verificação o *design* de blocos proposto utilizando o ILA. Finalizando o co-projeto *Hardware-Software*, por meio da ferramenta SDK.

3.2 Análise de *profiling*

Para encontrar o tempo de execução da CNN, seguindo a abordagem do co-projeto *Hardware-Software*, foi realizado o tutorial de análise de profiling disponível em (Areibi; Saunders, 2020). Portanto, o código foi executado no microprocessador ARM presente no SoC Zynq-7000, desse modo foi utilizada a ferramenta *Block Design*, presente no software Vivado, integrando os blocos IP (*Intellectual Property*) do processador, barramentos AXI e blocos de memória, como mostra a Figura 3.1. Após a validação do projeto IP, foi realizada a síntese e implementação do design, finalizando ao gerar o *bitstream* e o exportando para o XSDK (*Xilinx Software Development Kit*).

Na ferramente XSDK, foram inseridos os arquivos necessários para o funcionamento da rede neural, ou seja, o arquivo em C e suas bibliotecas, além de ajustar as configura-

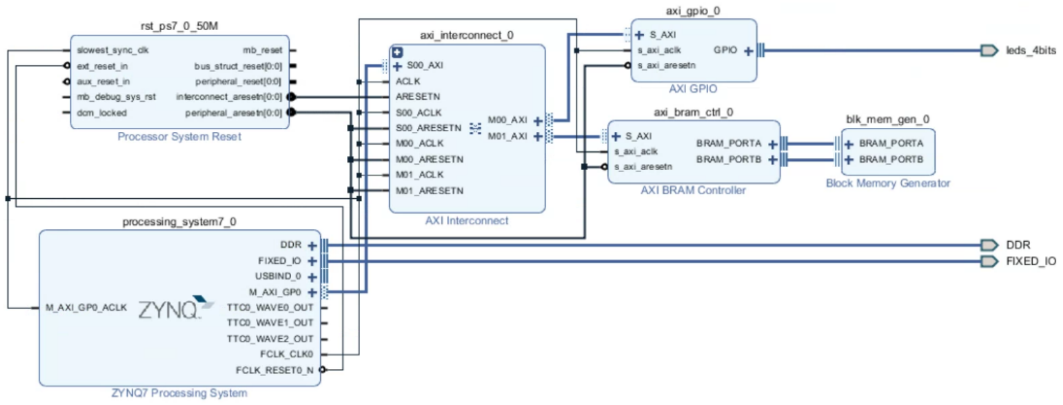


Figura 3.1 – Diagrama de blocos IP. Fonte: Autor.

ções de pacotes de suporte da placa e alterar os compiladores para que fosse permitida a análise através do gprof. Segundo os resultados obtidos da análise de *p*rofiling, as camadas convolucionais de maior gargalo foram a segunda e a terceira camada convolucionais.

3.3 Descrição das camadas

Para a implementação dessas camadas em VHDL foi proposta uma arquitetura RTL, variando apenas no tamanho do filtro. Utilizando representação aritmética em ponto flutuante de 27 bits e blocos multiplicadores e somadores baseados em máquinas de estados finitas disponibilizados em (Muñoz D. F. Sanchez, 2010) e (Muñoz *et al.*, 2010). Com isso, os *loops* das funções da CNN em C, por exemplo os da quarta camada convolucional exibidos no código 3.1, foram divididos como blocos na implementação da arquitetura.

```
1  void conv4()
2  {
3    for(int l=0; l<32; l++) // l - is the number of samples
4    {
5      for(int i=0; i<sampleLength/poolLength; i++) //i - size of the samples
6      {
7        for(int j=0; j<numberOfFilters; j++) //j - number of filters
8        {
9          for(int k=0; k<secondFilterLength; k++) //k - size of the filters
10         {
11           fourthConvOutput[l][i] +=
                 (thirdConvOutput[j][i+k]*fourthConvFilter[l][j][k]);
12         }
13       }
14       fourthConvOutput[l][i] += fourthConvBias[l];
15       fourthConvOutput[l][i] = RELU(fourthConvOutput[l][i]);
16     }
17   }
18 }
```

Código 3.1 – Quarta camada convolucional em C

Observando a Figura 3.2, tem-se o projeto RTL da quarta camada convolucional, de forma que: o bloco *conv1sample*, representando o *loop* mais interno, realiza a convolução entre uma amostra e um filtro; já o *cnn_conv4*, a partir de matrizes auxiliares, atualiza as entradas e acumula a saída do *conv1sample* até que toda a matriz auxiliar seja varrida pela matriz parcial de filtro, por fim soma o *BIAS* e finaliza ao aplicar a função não-linear *RELU*; enquanto isso, o bloco *cnn_conv4b* recebe toda a matriz de entrada e contém todos os valores do *kernel* e do *BIAS*, com isso atualiza as entradas do *cnn_conv4* e armazena as saídas obtidas na matriz de *output*. As outras camadas seguem o mesmo modelo, variando apenas no bloco convolucional e nas matrizes de entrada e filtro.

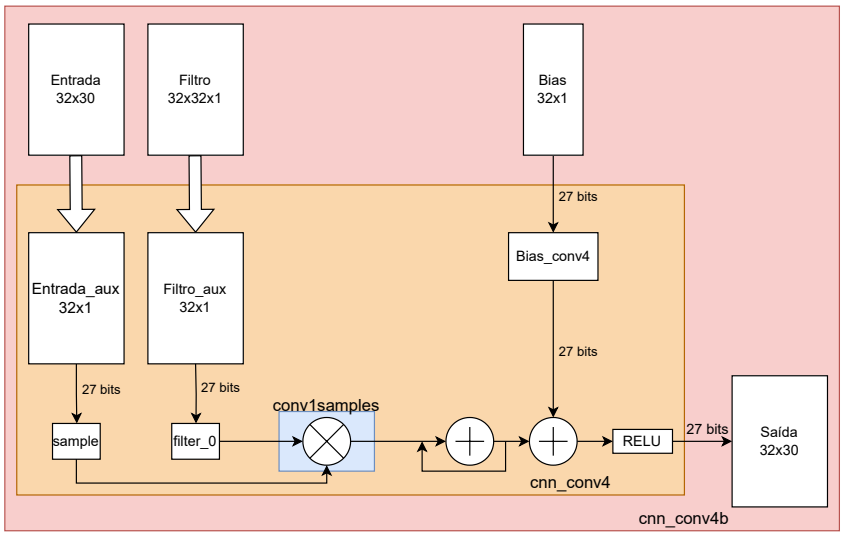
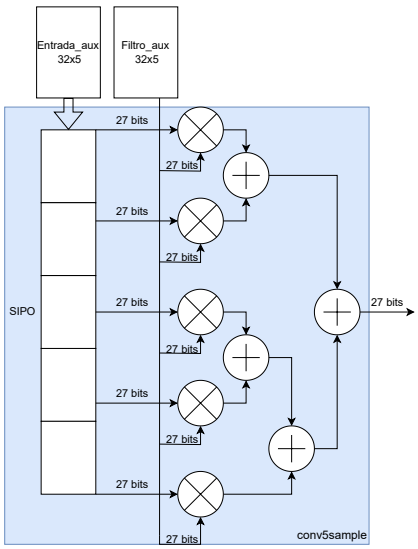


Figura 3.2 – Arquitetura proposta para a quarta camada convolucional em VHDL. Fonte: Autor.

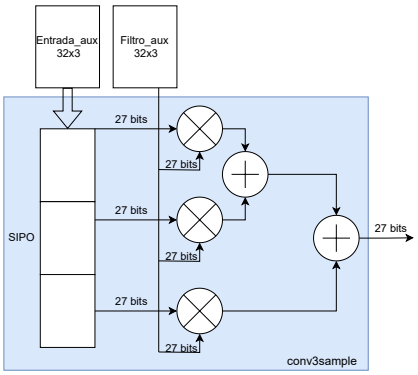
3.3.1 Bloco convolucional

Esse bloco funciona como o *loop* mais interno de uma camada convolucional da CNN em C, realizando a operação de convolução entre os valores de amostra e os filtros. Como se observa nas figuras 3.2a, 3.2b e 3.2c, a segunda, terceira e quarta camada tem, respectivamente, um filtro 5x1, um filtro 3x1 e um filtro 1x1. A descrição em VHDL dos mesmos estão no repositório (Raspante, 2024).

(a) Segunda camada.



(b) Terceira camada.



(c) Quarta camada.

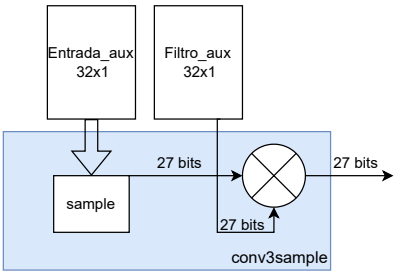


Figura 3.3 – Bloco convolucional das camadas implementadas em *hardware*. Fonte: Autor.

3.3.2 Bloco intermediário

O bloco intermediário equivale ao segundo e terceiro *loop* das camadas convolucionais como mostra o código 3.1. De forma que, gera uma saída por meio de matrizes auxiliares das entradas e dos filtros e de um *bias*, aplicando a função RELU no fim das operações. Sendo representado pelo bloco *cnn_conv4* na Figura 3.2. A descrição dos blocos intermediários está disponível em (Raspante, 2024).

Segue a máquina de estados finitos disposta na Figura 3.4, sendo que a mesma funciona da seguinta forma:

- inicio: envia ‘0’ para a saída e reinicia os blocos somadores. Além disso, aguarda o sinal *startloop* obter o valor ‘1’, mudando para “atualiza_entradas”;
- atualiza_entradas: envia os valores de amostras e filtros para o bloco convolucional e

o *bias* para o segundo bloco somador, além de definir em '1' o *start* do bloco convolucional. A mudança de estado em seu caso depende do sinal *done*, se for nulo vai para o estado "espera_resultado", caso contrário o próximo estado é o "soma";

- espera_resultado: aguarda o sinal *s_ready* do bloco convolucional se tornar '1', com isso o próximo estado é o "acumula";
- acumula: coleta a saída do bloco convolucional e envia para o bloco somador e realiza a retroalimentação do mesmo, obtendo o processo de acumular as saídas. Também, atualiza os índices das matrizes auxiliares para os sinais de entrada que vão para o bloco convolucional, caso os índices cheguem em 0 o sinal *done* é definido como '1'. O seu próximo estado é sempre "atualiza_entradas";
- soma: o sinal de *done* funciona como *start* para o segundo e último bloco somador, que recebe a saída do acumulador e o *bias*, após essa soma ser realizada a função RELU é utilizada obtendo uma saída final. O próximo estado se mantém como "soma" enquanto *sready_soma* for nulo. O sinal *sready_soma* é definido como '1' após o sinal de *ready* do segundo bloco somador for ativado;
- fim: nesse estado não ocorre nenhuma mudança, funciona como um atraso proposital e o seu próximo estado é o "inicio".

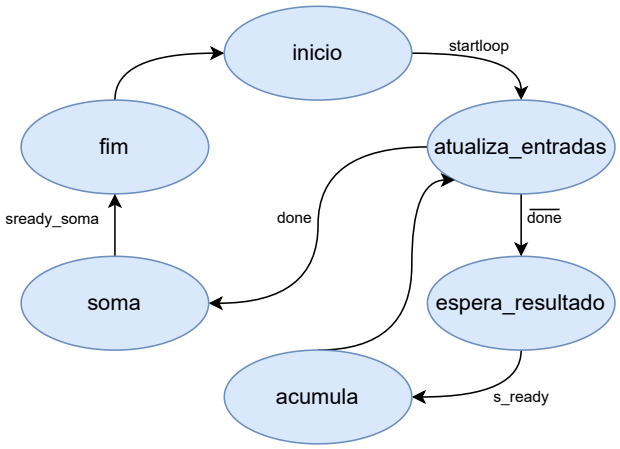


Figura 3.4 – FSM da camada intermediária. Fonte: Autor

3.3.3 Bloco principal

O bloco principal é representado no projeto a nível de transferidor como cnn_conv4b na Figura 3.2. Sua estrutura funcional é mostrada no diagrama de blocos da Figura 3.5, de modo que o sinal *startloop* inicia o bloco operacional e o sinal *ready_soma* indica quando uma saída foi obtida. A matriz de entrada *samples* é 32x30, ou seja, são 960 amostras. A quarta camada se encontra no Apêndice A.1, já as outras duas em (Raspante, 2024).

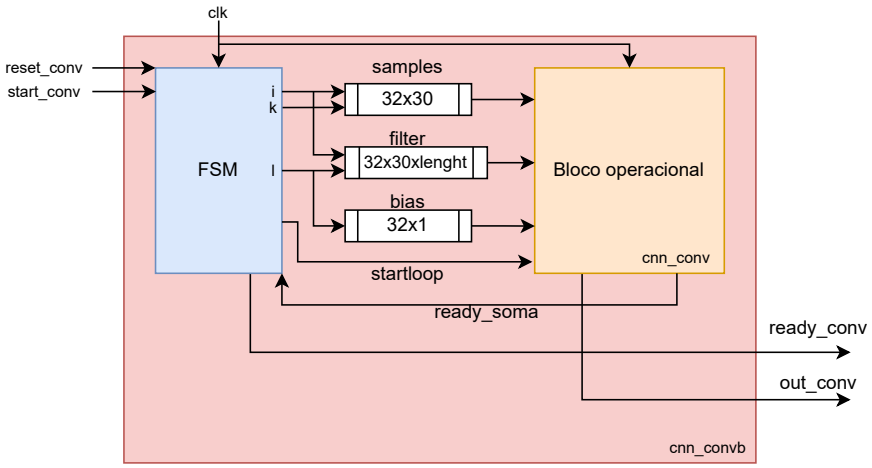


Figura 3.5 – Diagrama para o bloco externo das camadas convolucionais. Fonte: Autor

Sua máquina de estados finitos é composta por quatro estados, mostrados na Figura 3.6. De modo que, estão organizados para funcionar da seguinte maneira:

- inicio: envia '0' para a saída e reinicia os blocos somadoras. Além disso, aguarda a entrada *start_conv* receber o valor '1', mudando para "atualiza_entradas";
- atualiza_entradas: envia os valores de amostras, filtros e *bias* para o bloco operacional e controla o índice i, além de definir em '1' o sinal *startloop*. A mudança de estado em seu caso depende dos sinais *done* e *ready_soma*, se *done* for '1' vai para o estado "fim", e se *ready_soma* for '1' o próximo estado é o "reg_saida";
- reg_saida: armazena o sinal de saída do bloco operacional e atualiza os índices l e k das matrizes, sendo que retorna para o estado "atualiza_entradas";
- fim: define o sinal *ready_conv* em '1' e retorna para o estado "inicio".

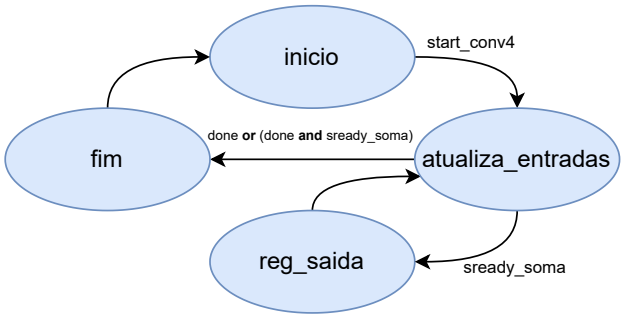


Figura 3.6 – *Finite State Machine* para o bloco externo das camadas convolucionais. Fonte: Autor

3.4 ILA core

Utilizando as ferramentas disponíveis foi realizada a validação em *hardware* das três camadas, na *ZedBoard* e utilizando do bloco *IP ILA core* para analisar as entradas e saídas do circuito. Além disso, as matrizes entradas de cada camada foram armazenadas separadamente em arquivo de memoria .coe, para que se implementasse o bloco *IP* da memória *BROM* em cada um dos casos. Criando uma maquina de estados finita foi possível controlar o envio de dados da memória para a camada convolucional, como mostrado o *topmodule* na Figura 3.7.

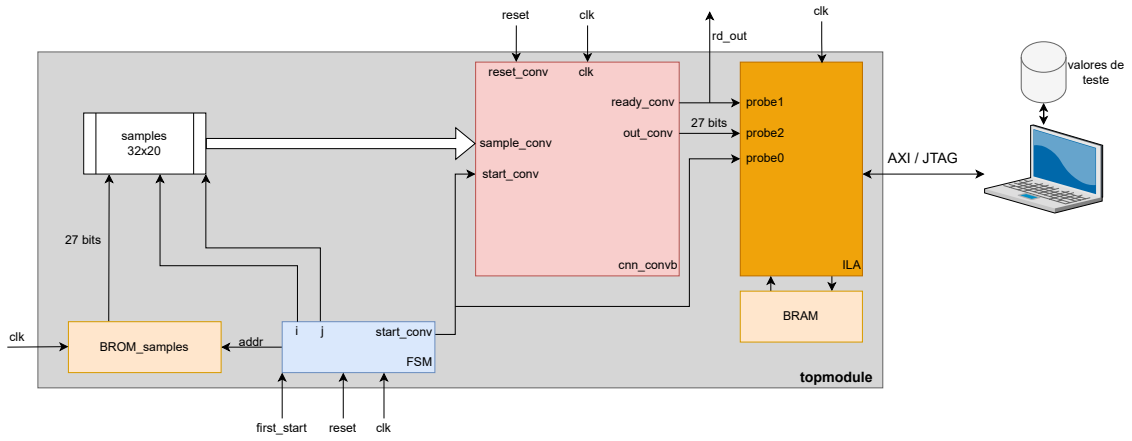


Figura 3.7 – *Topmodule* com ILA para validação de cada camada. Fonte: Autor

3.5 Encapsulamento IP e comunicação via AXI DMA

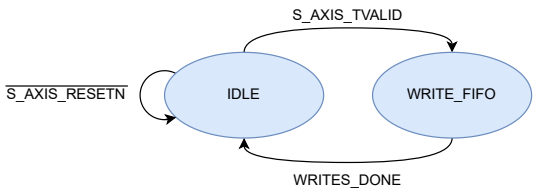
Após a validação das camadas com a utilização do *ILA*, foi encapsulado como uma IP (*Intellectual Property*) a estrutura quarta camada. Utilizando o protocolo de comunicação *AXI4-Stream*, foi gerado o bloco IP com uma interface escravo e uma mestre. A comunicação com o IP DMA é baseada no método *handshake*, em que a transição só é valida quando os sinais *tready* e *tvalid* são verdadeiros.

3.5.1 Estrutura do bloco IP

O bloco IP proposto é composto por uma interface *slave*, que recebe as amostras do sistema de processamento (PS) da ZedBoard via DMA, e por uma interface mestre, que envia a matriz de saída para o PS via DMA. Sendo baseado na estrutura de um registrador FIFO oferecida após criar o IP baseado em *AXI4_Stream*. O código em VHDL do mesmo está no Apêndice A.2.

O funcionamento do *slave* é baseado em uma FSM de dois estados, observados na Figura 3.7a. O estado *IDLE* primeiro aguarda o sinal de *reset* ser ‘1’ (ativo quando igual a ‘0’), após isso aguarda o sinal *s_axis_tvalid* enviado pelo DMA se definir em ‘1’. Com isso, próximo estado é o *WRITE_FIFO* nele o sinal *s_axis_tready* é definido como ‘1’ enquanto as 960 amostras não foram recebidas, ocorrendo o recebimento e armazenamento dos dados. Além disso, ocorre a atualização dos índices da matriz *sample_conv4*. No momento da última transação, o sinal *s_axis_tlast* é verdadeiro, funcionando como o sinal de *start_conv4*. Por fim, após as 960 transações o sinal *writes_done* se torna ‘1’ e ocorre o retorno ao estado *IDLE*.

(a) FSM da interface *slave*.



(b) FSM da interface mestre.

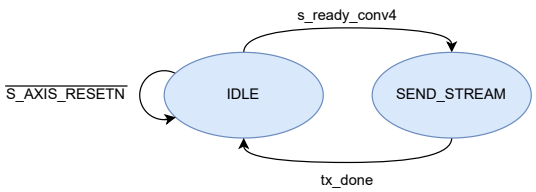


Figura 3.8 – FSMs do bloco IP encapsulado. Fonte: Autor.

A interface mestre funciona de maneira semelhante, como se observa na Figura 3.7b. Porém, em seu caso o bloco DMA é o responsável pelo controle do sinal *m_axis_tready*, e o sinal *m_axis_tvalid* é controlado internamente.

O estado *IDLE* aguarda até que o sinal *s_ready_conv4* se torne ‘1’, ou seja, até que as operações da camada convolucional terminem. Após isso, no estado *SEND_STREAM* o sinal *m_axis_tvalid* é verdadeiro, indicando que os dados enviados são válidos. Além disso, controla os índices da matriz de saída para enviar valor por valor na saída (*M_AXIS_TDATA*). Por fim, quando os índices ficarem iguais a 0, o sinal *tx_done* é definido como ‘1’, e ocorre o retorno ao estado *IDLE*. Com isso, obteve-se a estrutura representada como *AXIS_S2M*, seu diagrama da blocos é a Figura 3.11. Além disso, a matriz saída do bloco *cnn_conv4* está em 27 bits, portanto foram concatenados cinco zeros no fim de cada valor, para se obter os 32 bits. A saída *intr_conv4* foi utilizada apenas na fase de verificação do sistema.

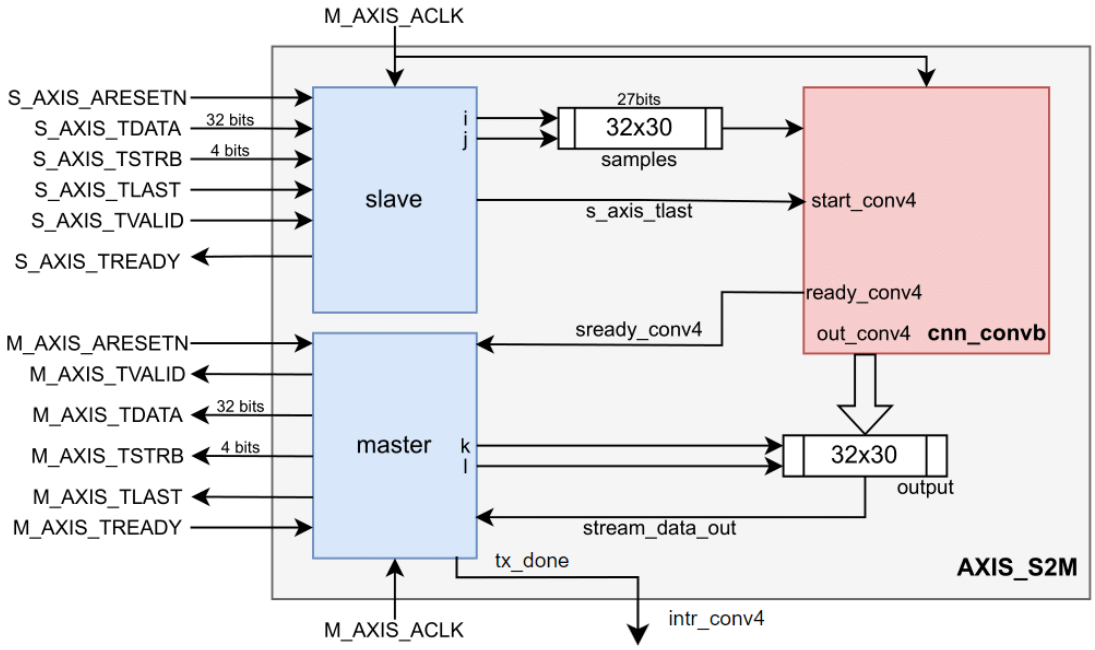


Figura 3.9 – Diagrama de blocos do IP encapsulado. Fonte: Autor

A validação inicial do bloco *AXIS_S2M* foi feita por meio de um *testbench*, para isso foi o bloco encapsulado foi utilizado no *block design* com portas e interfaces internas, como se observa na Figura 3.10. Com isso, a partir do *wrapper* gerado foi feito um *testbench* disponibilizado no Apêndice A.3.

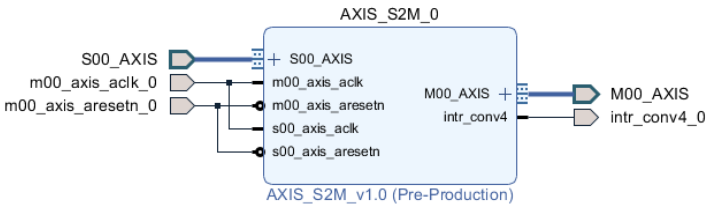


Figura 3.10 – *Block Design* para testes da IP. Fonte: Autor

3.5.2 *Block Design* do projeto

Após encapsular a quarta camada, foi feito o block design para realizar a comunicação entre o bloco de lógica programável (PL) e o sistema de processamento (PS). Para isso, foi utilizado o bloco AXI DMA, configurado em modo simples (Scatter/Gather desativado) e sem interrupção ativada.

Além disso, o IP *Zynq Processing System* foi configurado com: uma porta de alta performance para realizar a comunicação via DMA pelo IP *core* AXI *SmartConnect* (conecta dispositivos com transições AXI *memory-mapped*), já a configuração e inicialização do DMA ocorre pela interface *AXI4-Lite* via AXI *Interconnect* (XILINX, 2017); e uma porta para um sinal de interrupção, que foi utilizado apenas para analisar o funcionamento da quarta camada. Utilizando do IP *System ILA* foi possível verificar o funcionamento da comunicação entre PS-PL. A Figura 3.11 mostra o *Block Design* finalizado.

Com isso, utilizando a ferramenta SDK a rede neural convolucional feita em C foi modificada para realizar a quarta camada convolucional em *hardware* e o restante da mesma ser executada no processador ARM presente na ZedBoard, o código adaptado está disponibilizado no Apêndice B. De forma que, o DMA foi configurado para realizar transferências no modo *simple poll*. Coletando os arquivos .csv gerados na interface do ILA foi permitido calcular o erro quadrático médio.

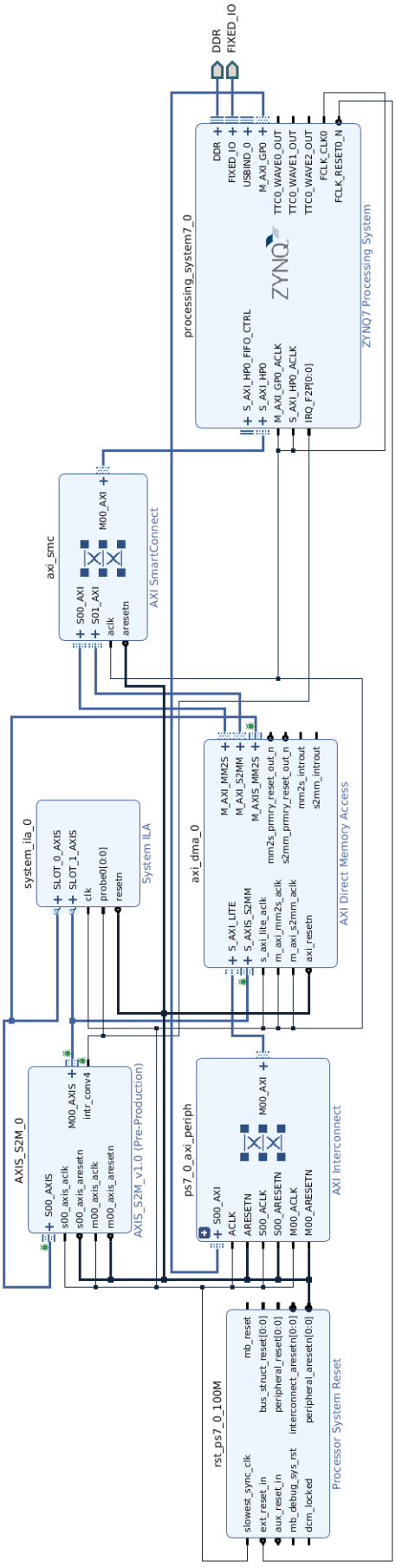


Figura 3.11 – *Block Design* do projeto. Fonte: Autor

4 Resultados

4.1 *Profiling*

A análise de *p*rofiling da rede neural convolucional em C realizada através do gprof, utilizando o XSDK em conjunto com o microprocessador ARM da placa Zynq-7000, é mostrada na Figura 4.1. Observa-se que a segunda, a terceira e a quarta camada juntas necessitam de 74,62% do tempo de execução total da CNN, ou seja, são as camadas de maior gargalo da rede, portanto as de maior necessidade de implementação e aceleração em hardware.

gmon file: C:\Users\Alunos\Desktop\teste\teste_ip2\teste_ip2.sdk\teste_conv\Debug\gmon.out
program file: C:/Users/Alunos/Desktop/teste/teste_ip2/teste_ip2.sdk/teste_conv/Debug/teste_conv.elf
16 bytes per bucket, each sample counts as 10.000us

type filter text

Name (location)	Samples	Calls	Time/Call	% Time
Summary	3720			100,0%
conv2	1558	1	15.580ms	41,88%
conv3	889	1	8.890ms	23,9%
fully1	740	1	7.400ms	19,89%
conv4	329	1	3.290ms	8,84%
conv1	126	1	1.260ms	3,39%
fully2	51	1	510.000us	1,37%
maxpooling	11	1	110.000us	0,3%
adjustSecondConvInput	5	1	50.000us	0,13%
adjustThirdConvInput	5	1	50.000us	0,13%
flatten	4	1	40.000us	0,11%
_vfprintf_r	1			0,03%
_write	1	0		0,03%
XScuGic_DeviceInitialize	0	1	0ns	0,0%
XScuGic_GetCpuID	0	1	0ns	0,0%
XScuGic_RegisterHandler	0	1	0ns	0,0%
XUartPs_SendByte	0	18	0ns	0,0%
__any_on	0			0,0%
_sbrk_r	0			0,0%
adjust_input	0	1	0ns	0,0%
cleanup_platform	0	1	0ns	0,0%
cortexa9_init	0	0		0,0%
disable_caches	0	1	0ns	0,0%
enable_caches	0	1	0ns	0,0%
init_platform	0	1	0ns	0,0%
init_uart	0	1	0ns	0,0%
main	0	0		0,0%
outputLayer	0	1	0ns	0,0%

Figura 4.1 – Análise de *p*rofiling da CNN no ARM. Fonte: Autor.

A segunda camada convolucional, que mais consome tempo de execução, consiste em quatro *l*oops e uma matriz de filtros de 32x32x5, junto a 32 valores de *b*ias, além de receber uma matriz entrada de 32x30, sendo assim a cada execução da função são enviados 960 amostras de sinal aECG. Consumindo cerca de 41,88% do tempo de execução no ARM-Zynq, durando 15,580ms a cada chamada.

4.2 *Delay* dos blocos operacionais

Nos blocos convolucionais mostrados nas figuras 3.2c, 3.2b e 3.2a, foi observado por meio de *testbenchs* os tempos de resposta das mesmas. Logo, o filtro 1x1 teve um delay de 10ns, o filtro 3x1 um atraso de 60ns e o filtro 5x1 leva 80ns após o *start*.

Também, os blocos intermediários das camadas foram analisadas, inserindo as matrizes auxiliares e obtendo um valor de saída. De forma que, o bloco intermediário da quarta camada tem um atraso de 2,29us. Já o delay para a terceira camada é de 3,89us. Enquanto que, para a segunda foi obtido 4,54us.

Por fim, os blocos principais das três camadas. De forma que, a quarta camada preenche a sua matriz de saída em 3,14ms. A terceira leva cerca de 4,67ms com um MSE igual a 0,0003257. E, por último, a segunda camada necessita de 5,30ms tendo um EQM igual a 0,0001895. Os *testbenchs* utilizados podem ser encontrados em (Raspante, 2024).

4.3 *ILA core*

Foi implementada e validada na ZedBoard, com um *clock* de 100MHz, as três camadas convolucionais mais custosas, utilizando-se do ILA para analisar as pontas de prova ligadas aos sinais internos. A quarta camada, que foi a encapsulada em IP, foi testada e validada utilizando o ILA core. Primeiramente, armazenando em arquivo memória .coe a matriz de entrada, que será tratada pelos filtros e *bias*, com isso será coletado 960 valores de saída, vide Figura 4.2. com um erro quadrático médio de $2{,}860758 \cdot 10^{-12}$ em relação ao valores esperados. Os *topmodules* das outras duas camadas se encontram em (Raspante, 2024).

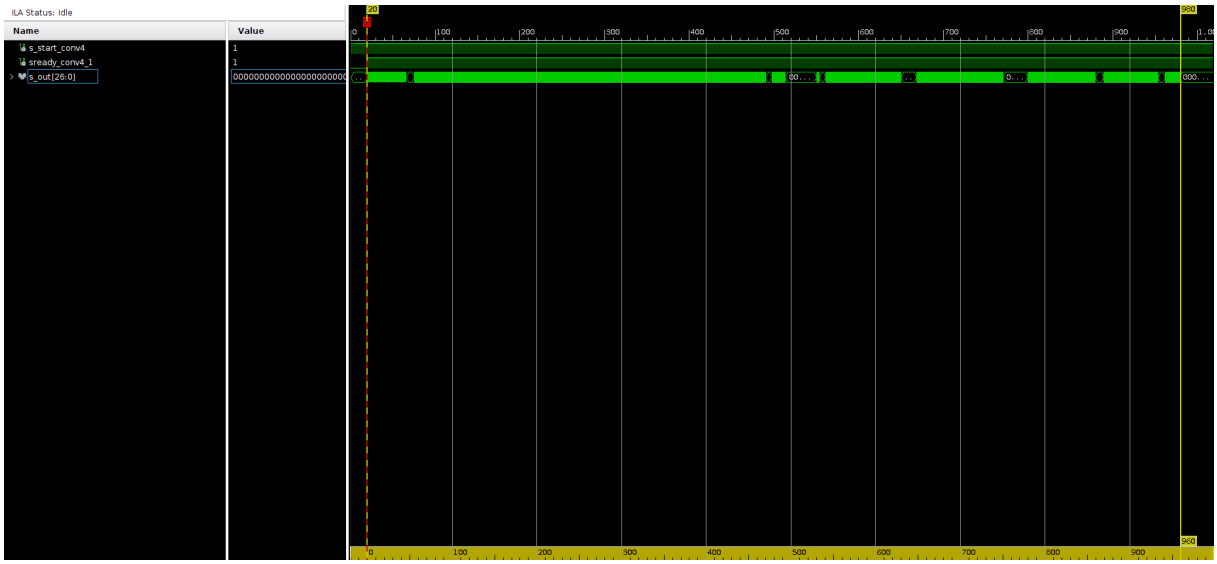


Figura 4.2 – Quarta camada no ILA. Fonte: Autor.

4.4 *Testbench* do bloco IP *AXIS_S2M*

Com o encapsulamento da quarta camada implementada em hardware, para o observar o comportamento da IP encapsulada foi realizado um *testbench*, mostrado na Figura 4.3.

Quando a entrada *S_AXIS_TVALID* recebe '1' a FSM do *slave* muda para *WRITE_FIFO*, logo a saída *S_AXIS_TREADY* envia '1', ocorrendo o *handshake* e iniciando o procedimento para receber os 960 valores de entrada.

Seguidamente, após a entrada *S_AXIS_TLAST* receber '1', indicando o último valor, a quarta camada recebe o *start* realizando o procedimento de convolução. Por fim, após o sinal *ready_conv4* indicar o fim da operação, a FSM do mestre muda para o estado *SEND_STREAM*, logo a saída *M_AXIS_TVALID* envia '1'. Além disso, a entrada *M_AXIS_TREADY* está recebendo '1', ocorrendo o *handshake* e iniciando o envio dos 960 valores da matriz de saída obtida pela quarta camada convolucional.

Quando a saída *M_AXIS_TLAST* está em '1' o último valor válido está sendo enviado, e com isso o sinal *tx_done* muda para '1' e podemos o observar pela saída *intr_conv4*.

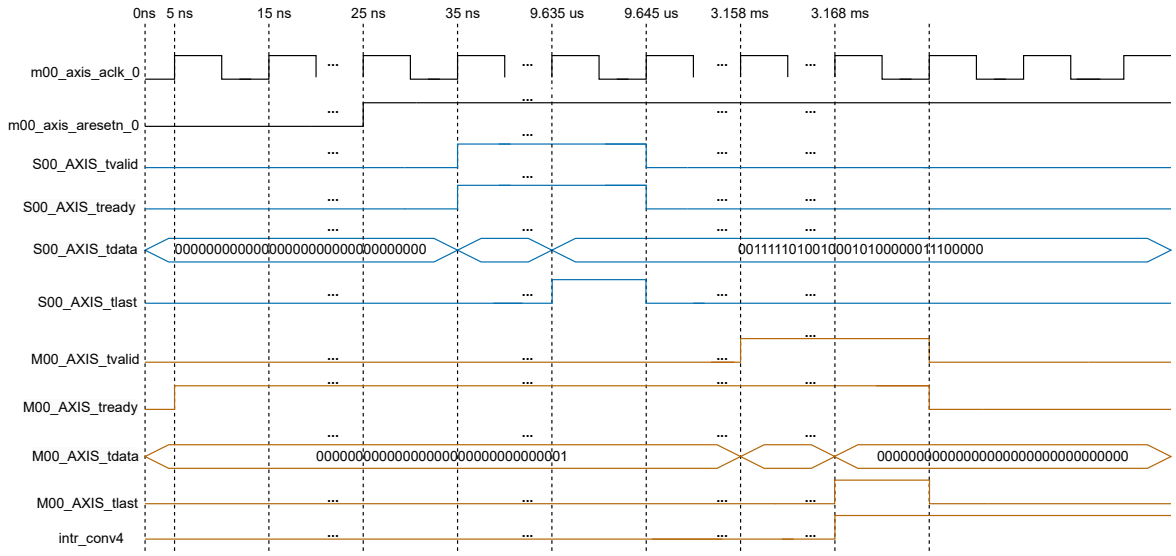


Figura 4.3 – *Testbench* do bloco AXIS_S2M. Fonte: Autor.

4.5 Funcionamento do *Block Design*

Para a verificação da comunicação PS-PL da ZedBoard, foi utilizado o bloco IP *System ILA*, permitindo visualizar os sinais do *slave* e do *master* da quarta camada encapsulada, como mostram as figuras 4.4 e 4.5. Além de analisar o sinal interno *tx_done* por meio da saída *intr_conv4*, que também foi inclusa no ILA, permitindo verificar o funcionamento da estrutura da interface mestre do bloco encapsulado.

Figura 4.4 – Transação de recebimento dos dados pelo escravo. Fonte: Autor.

Primeiramente, foi verificado o funcionamento do envio de dados do PS e o recebimento dos mesmos pelo PL. Ou seja, o envio dos 960 valores de amostra para o bloco implementado em *hardware*. Tal procedimento ocorreu de maneira correta e se observa na Figura 4.4.

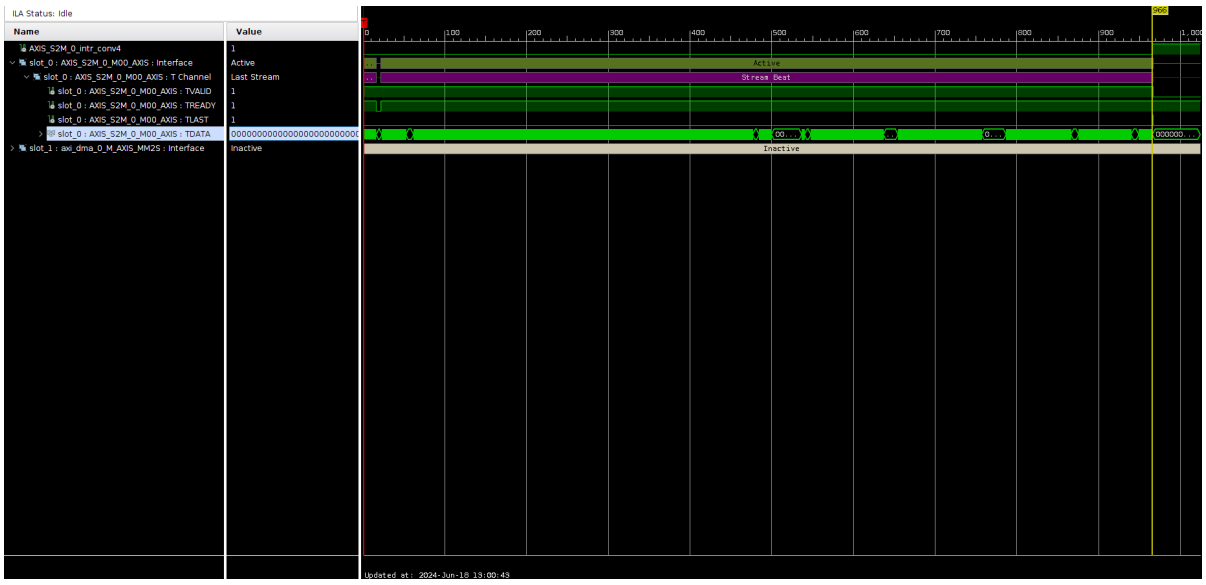


Figura 4.5 – Transação de envio dos dados pelo mestre. Fonte: Autor.

Após isso, foi analisado o comportamento do envio de dados do PL para o PS. Logo, observar os 960 valores da matriz de saída da quarta camada convolucional implementada, com um erro quadrático médio de 2,860758e-12, como mostra a Figura 4.6. Além disso, foi comparado o valor obtido sem e com a utilização do bloco IP implementado, 0,778923 e 0,778922, respectivamente. Obtendo um erro médio no *output* da CNN de 0,000128%. Sendo

assim, o comportamento mostrado na Figura 4.5 foi o correto.

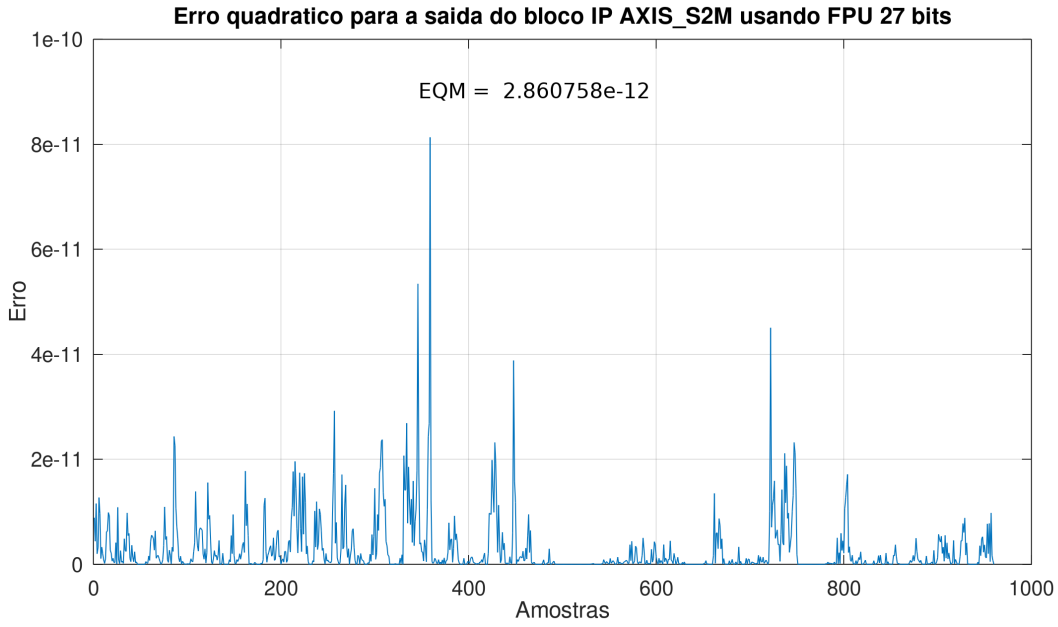


Figura 4.6 – EQM dos valores de saída do bloco implementado. Fonte: Autor.

4.6 *Reports* de *timing*, recursos e energia

Para a validação do *block design* implementado não foi demonstrado nenhuma violação de folgas negativas de tempo, funcionando corretando na frequência de *clock* de 100MHz.

A análise da utilização de recursos do bloco PL da ZedBoard após o processo de implementação do *block design* mostrado na Figura 3.11, de modo que foram utilizados 43,93% das *LookUp Tables*, 57,94% dos *flip-flops*, 3,21% da BRAM e 0,45% dos DSPs, vide Tabela 4.1.

Recursos	Utilização	Disponível	Utilização %
LUT	23372	53200	43,93
LUTRAM	816	17400	4,69
FF	61644	106400	57,94
BRAM	4,50	140	3,21
DSP	1	220	0,45
BUFG	2	32	6,25

Tabela 4.1 – Análise de recursos. Fonte: Autor.

O consumo energético total foi de 1,825W de potência, sendo que o principal fator de consumo é bloco de processamento com 1,533W, enquanto que a IP AXIS_S2M consome 0,11W. Com isso, para uma alimentação de 3,3V se obtém um consumo de 553mA, portanto uma bateria de 1200mAh forneceria uma autonomia de 2h10min.

Com isso, é possível realizar a comparação entre o uso da estrutura implementada e o não uso, ou seja, a rede neural convolucional acelerada com a quarta camada convolucional em *hardware*, contra a CNN apenas sendo executada no ARM Cortex-A9. Como se observa na Tabela 4.2, tem-se um menor tempo de execução para a quarta camada, além de uma melhor performance computacional, junto a um erro relativo baixo na saída final da CNN.

	Com o co-projeto	**Sem o co-projeto**
Tempo de execução da conv4	3,14ms	3,29ms
Valor final obtido	0,778922	0,778923
Erro relativo	0,000128%	-
Performance	455 MFLOPS/W	163 MFLOPS/W

Tabela 4.2 – Comparativo entre com e sem o co-projeto. Fonte: Autor.

Além disso, o PAR (*Place and Route*) da implementação na ZedBoard, encontra-se na Figura 4.7, mostrando em cores distintas a área de ocupação dos blocos utilizados. Os *reports* e os PARs de cada uma três camadas estão disponibilizados na pasta *PRINTS* do repositório (Raspante, 2024).

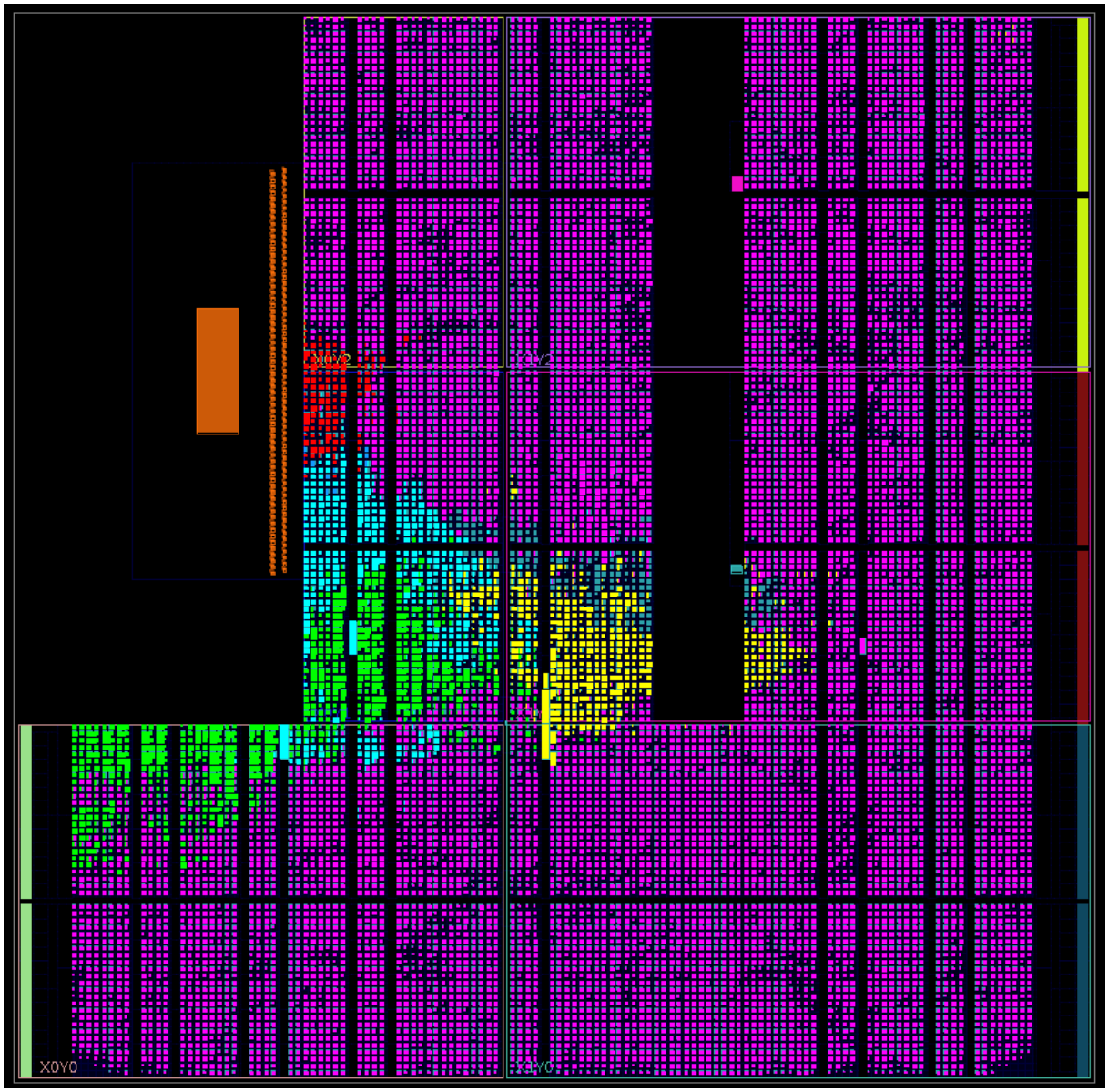


Figura 4.7 – *Place and Route* do projeto. Fonte: Autor.

Cada cor representa um dos blocos IPs organizados da seguinte forma:

- Rosa: Bloco IP *AXIS_S2M*;
- Azul Claro: AXI DMA;
- Amarelo: *System* ILA;
- Azul Escuro: *Processor System Reset*;
- Vermelho: AXI *Interconnect*;
- Verde: AXI *SmartConnect*.

5 Conclusões e Passos futuros

5.1 Conclusão

O projeto tinha como objetivo acelerar a CNN de estimação de frequência cardíaca fetal implementada em C por (Junior, 2018). Ao longo do projeto, foi constatado que as funções de maior consumo de tempo eram a segunda, a terceira e a quarta camada convolucionais.

Dessa maneira, foram implementadas para ser possível a aceleração em *hardware* de uma delas, diminuindo o tempo de cada uma. As três camadas foram validadas e funcionaram de maneira satisfatória, obtendo um MSE baixo em todas, principalmente na quarta e, ao realizar a comparação entre *software* e *hardware*, observa-se uma execução de 3,25 vezes mais rápido para a segunda camada, 2 vezes para a terceira camada em *hardware* e 1,05 vezes para a quarta, em relação com o processador ARM Cortex-A9 presente na ZedBoard que utiliza uma frequência de *clock* 6,67 vezes maior que os blocos implementados. Constatando que a aceleração da CNN por meio da implementação em *hardware* das camadas convolucionais é possível.

A quarta camada foi encapsulada como uma IP com protocolo *AXI4-Stream*, pois as transações não tem necessidade de serem mapeadas em memória, necessitando apenas do *handshake* entre os sinais *ready* e *valid*. Também, permitindo a transação de uma palavra de 32 *bits* a cada ciclo de *clock*, necessitando de 9,60us para realizar as transações de leitura e escrita dos 960 valores das matrizes de entrada e saída. Permitindo a comunicação com o bloco de processamento via o IP core AXI DMA.

Além disso, como os blocos somadores e multiplicadores geram uma saída em 20ns, o *throughput* para os blocos implementados é de 50 MFLOPS, dessa forma o bloco AXIS_S2M, que contém a quarta camada encapsulada, tem uma eficiência computacional de 455 MFLOPS/W, sendo que o ARM Cortex-A9 tem de 160 MFLOPS/W. Indicando que a aceleração em *hardware* de uma das camadas convolucionais torna o projeto mais eficiente.

A implementação desse projeto que realiza comunicação entre o software e a quarta camada encapsulada ocorreu de maneira esperada, com um erro percentual de 0,000128%. Portanto, mostra que o co-projeto *hardware-software* é válido e não causa impactos negativos na precisão da CNN proposta por (Junior, 2018).

Porém, não foi validada a CNN implementada em C com dados reais, por não encontrar um banco de dados com valores compatíveis com a mesma.

5.2 Próximos Passos

Um feito interessante seria conseguir verificar o comportamento da CNN implementada em C com valores de dados reais. Além disso, criar e testar novas estruturas em hardware para encapsular mais de uma camada, ou reaproveitar um mesmo IP para diferentes camadas convolucionais. Um dos caminhos para isso é a reconfiguração dinâmica para alterar entre os blocos internos de cada uma das camadas. Uma outra possibilidade seria utilizar a ferramenta de Síntese de Alto Nível (HLS) da Xilinx para obter novas estruturas em hardware das camadas convolucionais.

Outro ponto seria criar um algoritmo com interface de usuário que sendo configurada implementaria uma camada convolucional em hardware, encapsulando a mesma com a interface *AXI4-Stream* e gerando o *block design* com as IPs necessárias. Permitindo usar a IP criada para gerar um *layout* em *softwares* como o Cadence Innovus, podendo ser usado em aplicações mais específicas.

Além disso, buscar um novo modelo sem o uso do processador ARM, pois o mesma causa um alto consumo energético no projeto como um todo. Utilizando um processador *softcore*, por exemplo o AMD MicroBlaze, em seu lugar.

Referências

ABDULHAY, E. W.; OWEIS, R. J.; ALHADDAD, A. M.; SUBLABAN, F. N.; RADWAN, M. A.; ALMASAEED, H. M. Review Article: Non-Invasive Fetal Heart Rate Monitoring Techniques. **Biomedical Science and Engineering**, 2014. Citado na p. 18.

AMD. **AXI DMA v7.1 LogiCORE IP Product Guide**. 2022. https://docs.amd.com/r/en-US/pg021_axi_dma/AXI-DMA-v7.1-LogiCORE-IP-Product-Guide. Acessado em: 10-05-2024. Citado nas pp. 9, 28 e 29.

AMD. **Zynq-7000 SoC Technical Reference Manual**. 2023. https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM. Acessado em: 10-05-2024. Citado nas pp. 9, 25 e 26.

AMD. **Vitis High-Level Synthesis User Guide**. 2024. https://docs.amd.com/r/en-US/ug1399-vitis-hls. Acessado em: 10-05-2024. Citado nas pp. 10, 28, 79 e 80.

ANDRADE, M. d. S. Implementação em FPGA da Camada Convolucional de um Algoritmo de Redes Neurais para um Módulo Estimador da Frequência Cardíaca Fetal. 2022. Citado nas pp. 15 e 30.

AREIBI, S.; SAUNDERS, M. Tutorial: Building an Embedded Processor System on a Xilinx Zynq FPGA (Profiling). 2020. Citado na p. 30.

ARM. AMBA AXI and ACE Protocol Specification. 2011. Citado nas pp. 9, 26 e 27.

BARBOSA, I. J. T. Aceleração de Algoritmos para Estimativa da Frequência Cardíaca Fetal Utilizando FPGA. 2016. Citado na p. 15.

BERBERT, W.; BERTINI, L.; COPETTI, A. Aceleração de hardware em sistemas embarcados para aprendizado de máquina utilizando KNN em FPGA. *In*: **Anais do XII Computer on the Beach - COTB '21**. Online: Universidade do Vale do Itajaí, 2021. p. 400–407. Disponível em: https://siaiap32.univali.br/seer/index.php/acotb/article/view/17431. Citado nas pp. 24 e 25.

CARDON, A.; MüLLER. Introdução Às Redes Neurais Artificiais. 1994. Citado nas pp. 9, 21 e 22.

CARLSON, B. M. **Embryology and Developmental Biology**. 5. ed. Michigan: Saunders, 2013. Citado nas pp. 9 e 17.

HASAN, M. A.; IBRAHIMY, M. I.; REAZ, M. B. I. Fetal ECG Extraction from Maternal Abdominal ECG Using Neural Network. **Journal of Software Engineering and Applications**, v. 02, n. 05, p. 330–334, 2009. ISSN 1945-3116, 1945-3124. Disponível em: http://www.scirp.org/journal/doi.aspx?DOI=10.4236/jsea.2009.25043. Citado na p. 18.

HAYKIN, S. S. **Neural networks: a comprehensive foundation**. 2nd ed. ed. Upper Saddle River, N.J: Prentice Hall, 1999. ISBN 978-0-13-273350-2. Citado nas pp. 20, 21 e 23.

HUNTER, J. C. A.; BRAULIN, R. J.; LANSFORD, K. G.; KNOEBEL, S. B. Method for obtaining a fetal electrocardiogram. p. 1–3, 1964. Citado na p. 14.

JUNIOR, H. Proposta de uma Arquitetura de Redes Neurais para Estimativa da Frequência Cardíaca Fetal a Partir do ECG Abdominal em Gestantes. 2018. Citado nas pp. 15 e 48.

KAHANKOVA, R.; RADEK, M.; RENE, J.; BEHBEHANI, K.; ADAM, M.; MICHAL, J.; A., B. J. A Review of Signal Processing Techniques for Non-Invasive Fetal Electrocardiography. p. 51–73, 2020. Citado nas pp. 9, 14, 15, 18 e 19.

KOVACS, Z. L. **Redes neurais artificiais: fundamentos e aplicacoes**. [*S.l.: s.n.*], 1996. Citado na p. 20.

LECUN, Y.; KAVUKCUOGLU, K.; FARABET, C. Convolutional networks and applications in vision. *In*: **Proceedings of 2010 IEEE International Symposium on Circuits and Systems**. Paris, France: IEEE, 2010. p. 253–256. ISBN 978-1-4244-5308-5. Disponível em: http://ieeexplore.ieee.org/document/5537907/. Citado nas pp. 9 e 22.

LECUN, Y. A.; BOTTOU, L.; ORR, G. B.; MÜLLER, K.-R. Efficient backprop. *In*: ____. **Neural Networks: Tricks of the Trade: Second Edition**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 9–48. ISBN 978-3-642-35289-8. Disponível em: https://doi.org/10.1007/978-3-642-35289-8_3. Citado nas pp. 9 e 23.

LISENBEE, N.; TYNDALL, J. A. Fetal Heart Rate Monitoring. *In*: GANTI, L. (Ed.). **Atlas of Emergency Medicine Procedures**. New York, NY: Springer, 2016. p. 639–642. ISBN 978-1-4939-2507-0. Disponível em: https://doi.org/10.1007/978-1-4939-2507-0_109. Citado nas pp. 9 e 18.

MAXFIELD, C. **The Design Warrior's Guide to FPGAs**. 1st ed. ed. [*S.l.*]: Elsevier, 2004. Citado na p. 24.

MOORE, K. L.; PERSAUD, T. V. N. **Embriologia Clínica**. 8. ed. Rio de Janeiro: Elsevier, 2008. Citado na p. 17.

MUñOZ D. F. SANCHEZ, C. H. L. M. A.-R. D. M. Tradeoff of fpga design of a floating-point library for arithmetic operators. **Journal of Integrated Circuits and Systems 5**, 2010. Citado na p. 31.

MUñOZ, D. M.; SANCHEZ, D. F.; LLANOS, C. H.; AYALA-RINCóN, M. Fpga based floating-point library for cordic algorithms. **IEEE Southern Programmable Logic Conference (SPL)**, 2010. Citado na p. 31.

RAFIE, N.; KASHOU, A. H.; NOSEWORTHY, P. A. ECG Interpretation: Clinical Relevance, Challenges, and Advances. **Hearts**, v. 2, n. 4, p. 505–513, nov. 2021. ISSN 2673-3846. Disponível em: https://www.mdpi.com/2673-3846/2/4/39. Citado na p. 14.

RASPANTE, G. **TCC-PIBIC**. 2024. Disponível em: https://github.com/GRaspante/TCC-PIBIC. Citado nas pp. 32, 33, 34, 42 e 46.

RAUBER, T. W. Redes neurais artificiais. 2005. Citado nas pp. 9, 20 e 21.

RODRIGUES, J. A. Implementação da Comunicação Sem Fio de um Módulo Estimador da Frequência Cardíaca Fetal Baseado em FPGA. 2016. Citado na p. 15.

SAMENI, R.; CLIFFORD, G. D. A Review of Fetal ECG Signal Processing Issues and Promising Directions. **The Open Pacing, Electrophysiology & Therapy Journal**, 2010. ISSN 1876536X. Disponível em: http://benthamopen.com/ABSTRACT/TOPETJ-3-4. Citado nas pp. 14, 17, 18 e 19.

SANTANA, G. D. de. ArchLearn: Implementação de acelerador em hardware baseado em FPGA para redes neurais artificiais. 2020. Citado na p. 24.

SMITH, V.; ARUNTHAVANATHAN, S.; NAIR, A.; ANSERMET, D.; COSTA, F. D. S.; WALLACE, E. M. A systematic review of cardiac time intervals utilising non-invasive fetal electrocardiogram in normal fetuses. **BMC Pregnancy and Childbirth**, v. 18, n. 1, p. 370, dez. 2018. ISSN 1471-2393. Disponível em: https://bmcpregnancychildbirth.biomedcentral.com/articles/10.1186/s12884-018-2006-8. Citado nas pp. 9 e 19.

SOLT, I.; DIVON, M. Y. Fetal surveillance tests. *In*: **The Embryo: Scientific Discovery and Medical Ethics**. [*S.l.: s.n.*], 2004. v. 3, p. 291–308. Citado na p. 14.

STEINBURG, S. P. V.; BOULESTEIX, A.-L.; LEDERER, C.; GRUNOW, S.; SCHIERMEIER, S.; HATZMANN, W.; SCHNEIDER, K.-T. M.; DAUMER, M. What is the "normal" fetal heart rate? **PeerJ**, v. 1, p. e82, jun. 2013. ISSN 2167-8359. Disponível em: https://peerj.com/articles/82. Citado na p. 18.

TOMASI, H. F. Acelerador de hardware em fpga para aplicações em aprendizado de máquina. UnB, 2020. Citado na p. 24.

TUTIDA, H. I. C. Implementação de um Módulo de Aquisição de ECG Abdominal em Gestantes para Estimativa da Frequência Cardíaca Fetal Usando FPGA. 2016. Citado na p. 15.

XILINX. **7-Series Architecture Overview**. 2013. https://fpga.eetrend.com/files-eetrend-xilinx/forum/201509/9204-20390-7_series_architecture_overview.pdf. Acessado em: 10-05-2024. Citado nas pp. 9 e 25.

XILINX. Vivado AXI Reference Guide - UG1037. 2017. Citado nas pp. 26, 27 e 39.

ZHANG, Y.; GU, A.; XIAO, Z.; XING, Y.; YANG, C.; LI, J.; LIU, C. Wearable Fetal ECG Monitoring System from Abdominal Electrocardiography Recording. **Biosensors**, v. 12, n. 7, p. 475, jun. 2022. ISSN 2079-6374. Disponível em: https://www.mdpi.com/2079-6374/12/7/475. Citado nas pp. 9 e 20.

Apêndices

Apêndice A – Descrição em *Hardware* e *Testbenchs*

A.1 Bloco Principal da Quarta Camada

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use ieee.std_logic_unsigned.all;
4  use IEEE.NUMERIC_STD.ALL;
5  use work.fpupack.all;
6
7  entity cnn_conv4b is
8      Port (  reset_conv4 : in STD_LOGIC;
9              clk : in STD_LOGIC;
10             start_conv4 : in STD_LOGIC;
11             samples_conv4 : in in_Conv4;
12 --            filter_conv4 : in filter4;
13 --            bias_conv4 : in biasConv;
14             ready_conv4: out STD_LOGIC;
15             output_conv4: out out_Conv2);
16 --            out_conv4: out std_logic_vector (FP_WIDTH-1 downto 0));--ILA
17 end cnn_conv4b;
18
19 architecture Behavioral of cnn_conv4b is
20
21 component cnn_conv4 is
22 Port (  reset : in STD_LOGIC;
23         clk : in STD_LOGIC;
24         startloop : in std_logic;
25         sample_adjusted_aux : in aux_filter4;
26         conv4_Bias : in std_logic_vector (FP_WIDTH-1 downto 0);
27         filter_aux : in aux_filter4;
28         ready_loop : out std_logic;
29         ready_soma : out std_logic;
30         output_final : out std_logic_vector (FP_WIDTH-1 downto 0));
31 end component;
32
33 --signal sfilter_conv4 : filter4 := fourthConvfilter;
34 --signal sbias_conv4 : biasConv := fourthConvBias;
35 signal soutput_conv4 : out_Conv2;
36 signal start_conv4_delay : std_logic;
37 signal s_sample_adjusted_aux : aux_filter4;
38 signal sfilter_aux : aux_filter4;
```

```
39  signal sconv4_Bias, soutput_final : std_logic_vector (FP_WIDTH-1 downto 0);
40  signal s_startloop, sready_conv4 : std_logic;
41  --signal s_save : std_logic;
42
43  signal sready_loop, sready_soma : std_logic;
44  signal done : std_logic := '0';
45
46  signal i : integer range 0 to (numberOfFilters-1):= numberOfFilters-1;
47  signal l : integer range 0 to (numberOfFilters-1):= numberOfFilters-1;
48  signal k : integer range 0 to (outConvs-1):= outConvs-1;
49
50  --signal m : integer range 0 to (numberOfFilters-1):= numberOfFilters-1; --INDICES DA SAIDA P/ ILA
51  --signal n : integer range 0 to (outConvs-1):= outConvs-1;
52
53  TYPE estados is (inicio, atualiza_entradas, reg_saida, fim);
54  signal estado_atual, proximo_estado : estados;
55
56  begin
57      loop_soma :         cnn_conv4 port map(
58                          reset => reset_conv4,
59                          clk => clk,
60                          startloop => s_startloop,
61                          sample_adjusted_aux => s_sample_adjusted_aux,
62                          conv4_Bias => sconv4_Bias,
63                          filter_aux => sfilter_aux,
64                          ready_loop => sready_loop,
65                          ready_soma => sready_soma,
66                          output_final => soutput_final);
67
68  state_reg: process(clk, reset_conv4)
69  begin
70      if reset_conv4 = '1' then
71          estado_atual <= inicio;
72      elsif rising_edge (clk) then
73          estado_atual <= proximo_estado;
74      end if;
75  end process;
76
77  next_state_logic: process(clk)
78  begin
79      if rising_edge(clk) then
80          case estado_atual is
81          when inicio =>
82              if (start_conv4 or start_conv4_delay) = '1' then
83                  proximo_estado <= atualiza_entradas;
84              else
85                  proximo_estado <= inicio;
```

```
86              end if;
87
88          when atualiza_entradas =>
89              if done = '1' then
90                  proximo_estado <= fim;
91              elsif (s_startloop and sready_soma) = '1' then
92                  proximo_estado <= reg_saida;
93              else
94                  proximo_estado <= atualiza_entradas;
95              end if;
96
97          when reg_saida =>
98              if done = '1' then
99                  proximo_estado <= fim;
100             elsif sready_soma = '1' then
101                 proximo_estado <= reg_saida;
102             else
103                 proximo_estado <= atualiza_entradas;
104             end if;
105
106         when fim =>
107             proximo_estado <= inicio;
108
109         when others =>
110             proximo_estado <= inicio;
111         end case;
112     end if;
113 end process;
114
115 output_logic: process(clk, estado_atual)
116 variable count : std_logic_vector(1 downto 0);
117 variable count1 : std_logic_vector(1 downto 0);
118
119     begin
120         if rising_edge(clk) then
121             case estado_atual is
122             when inicio => -- ESTADO INICIAL QUE AGUARDA O START
123                 s_startloop <= '0';
124                 count := "10";
125                 count1 := "10";
126                 i <= 31;
127                 l <= 31;
128                 k <= 29;
129                 done <= '0';
130                 sready_conv4 <= '0';
131                 start_conv4_delay <= start_conv4;
132
```

```
133             when atualiza_entradas => -- ESTADO QUE ATUALIZA AS ENTRADAS DA CONVOLUÇÃO
134                 s_sample_adjusted_aux(i) <= samples_conv4(i,k);
135                 sfilter_aux(i) <= fourthConvfilter(l,i);
136                 sconv4_Bias <= fourthConvBias(l);
137 
138                 if count1 = "10" then
139                         if i<=0 then
140                             i <= i;
141                             s_startloop <= '1';
142                             count1 := "00";
143                             count := "10";
144                         else
145                             i <= i-1;
146                             count1 := "00";
147                         end if;
148                 else
149                     count1 := std_logic_vector(unsigned(count1)+1);
150                 end if;
151 
152             when reg_saida => -- ESTADO QUE VARIA OS ÍNDICES DAS MATRIZES E COLETA A SAIDA
153                 s_startloop <= '0';
154                 i <= 31;
155                 if count = "10" then
156                     soutput_conv4(l,k) <= soutput_final;
157                     count := "00";
158                         if k<=0 and l>0 then
159                             k <= 29;
160                             l <= l-1;
161                         elsif k<=0 and l=0 then
162                             k <= k;
163                             l <= l;
164                             done <= '1';
165                         else
166                             k <= k-1;
167                             l <= l;
168                         end if;
169                 else
170                     count := std_logic_vector(unsigned(count)+1);
171                 end if;
172 
173             when fim => -- ESTADO FIM SETA O READY;
174                 s_startloop <= '0';
175                 sready_conv4 <= '1';
176                                 --out_conv4 <= soutput_conv4(m,n); -- SAIDA ILA
177                 --if n<=0 and m>0 then
178                   --  n <= 29;
179                   --  m <= l-1;
```

```
180                 --elsif n<=0 and m=0 then
181                   --  n <= n;
182                   --  m <= m;
183                 --else
184                    -- n <= n-1;
185                    -- m <= m;
186                --end if;
187
188             when others => -- outros estados
189                 s_startloop <= '0';
190             end case;
191 end if;
192 end process;
193
194 ready_conv4 <= sready_conv4;
195 output_conv4 <= soutput_conv4;
196
197 end Behavioral;
198
```

A.2 Descrição de AXIS_S2M

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5  use work.fpupack.all;
6
7  entity AXIS_S2M_v1_0_M00_AXIS is
8          generic (
9                  -- inicio: parametros do usuario
10
11                 -- fim: parametros do usuario
12
13                 -- Tamanho dos dados que o slave recebe.
14                 C_S_AXIS_TDATA_WIDTH        : integer        := 32;
15
16                 -- Largura do barramento de endereço S_AXIS. \\
17                 -- O escravo aceita os endereços de leitura e escrita com largura C_M_AXIS_TDATA_WIDTH.
18                 C_M_AXIS_TDATA_WIDTH        : integer        := 32;
19
20                 -- O numero de ciclos de clock que o mestre \\
21                 -- aguardará antes de iniciar qualquer transacao (opcional).
22                 C_M_START_COUNT         : integer        := 32
23         );
24         port (
```

```
25                  -- inicio: portas do usuario
26                  INTR_CONV4      : out std_logic;
27                  -- fim: portas do usuario
28
29                  -- Global ports
30                  -- AXI4Stream slave: Clock.
31                  S_AXIS_ACLK        : in std_logic;
32                  -- AXI4Stream slave: Reset.
33                  S_AXIS_ARESETN        : in std_logic;
34                  -- AXI4Stream slave: Ready -> indicando para o mestre (DMA) que esta apto para receber
35                  S_AXIS_TREADY        : out std_logic;
36                  -- AXI4Stream slave: TDATA -> entrada do slave para dados de tamanho C_S_AXIS_TDATA_WID
37                  S_AXIS_TDATA        : in std_logic_vector(C_S_AXIS_TDATA_WIDTH-1 downto 0);
38                  -- AXI4Stream slave: indica quais bytes sao validos (opcional).
39                  S_AXIS_TSTRB        : in std_logic_vector((C_S_AXIS_TDATA_WIDTH/8)-1 downto 0);
40                  -- AXI4Stream slave: indica qual o ultimo valor da transacao.
41                  S_AXIS_TLAST        : in std_logic;
42                  -- AXI4Stream slave: indica quando os dados da trasacao sao validos.
43                  S_AXIS_TVALID        : in std_logic;
44
45                  -- AXI4Stream mestre: Clock
46                  M_AXIS_ACLK        : in std_logic;
47                  -- AXI4Stream mestre: Reset
48                  M_AXIS_ARESETN        : in std_logic;
49                  -- AXI4Stream mestre: TVALID indica que o mestre está conduzindo uma transferência vál
50                  -- Uma transferência ocorre quando tanto TVALID quanto TREADY sao iguais a 1.
51                  M_AXIS_TVALID        : out std_logic;
52                  -- AXI4Stream mestre: entrada do mestre para dados de tamanho C_M_AXIS_TDATA_WIDTH.
53                  M_AXIS_TDATA        : out std_logic_vector(C_M_AXIS_TDATA_WIDTH-1 downto 0);
54                  -- AXI4Stream mestre: indica quais bytes sao validos (opcional).
55                  M_AXIS_TSTRB        : out std_logic_vector((C_M_AXIS_TDATA_WIDTH/8)-1 downto 0);
56                  -- AXI4Stream mestre: ndica qual o ultimo valor da transacao.
57                  M_AXIS_TLAST        : out std_logic;
58                  -- AXI4Stream mestre: TREADY indica que o slave (DMA) pode aceitar uma transferência no
59                  M_AXIS_TREADY        : in std_logic
60          );
61  end AXIS_S2M_v1_0_M00_AXIS;
62
63  architecture implementation of AXIS_S2M_v1_0_M00_AXIS is
64  component cnn_conv4b is
65      Port (  reset_conv4 : in STD_LOGIC;
66              clk : in STD_LOGIC;
67              start_conv4 : in STD_LOGIC;
68              samples_conv4 : in in_Conv4;
69  --            filter_conv4 : in filter4;
70  --            bias_conv4 : in biasConv;
71              ready_conv4: out STD_LOGIC;
```

```
72              output_conv4: out out_Conv2);
73  --            out_conv4: out std_logic_vector (FP_WIDTH-1 downto 0));--ILA
74  end component;
75
76          -- Quantidade de dados de saída
77          constant NUMBER_OF_OUTPUT_WORDS : integer := 960;
78
79          -- Função chamada clogb2 que retorna um inteiro com o valor do teto do logaritmo na base 2.\\
80          -- Essa função é útil para determinar o número de bits necessários para representar um determin
81          function clogb2 (bit_depth : integer) return integer is
82                   variable depth  : integer := bit_depth;
83                   variable count  : integer := 1;
84           begin
85                    for clogb2 in 1 to bit_depth loop  -- Works for up to 32 bit integers
86                if (bit_depth <= 2) then
87                  count := 1;
88                else
89                  if(depth <= 1) then
90                          count := count;
91                       else
92                          depth := depth / 2;
93                     count := count + 1;
94                       end if;
95                    end if;
96             end loop;
97             return(count);
98           end;
99
100         -- WAIT_COUNT_BITS e o tamanho do contador (opcional).
101         constant  WAIT_COUNT_BITS  : integer := clogb2(C_M_START_COUNT-1);
102
103         -- bit_num e o número mínimo de bits necessários para \\
104         -- endereçar uma profundidade de um FIFO de tamanho 'depth'.
105         constant bit_num : integer := clogb2(NUMBER_OF_OUTPUT_WORDS-1);
106
107         -- Estados da FSM do mestre
108         type state is ( IDLE,        -- Estado idle.
109                         SEND_STREAM);  -- Neste estado, os dados do fluxo são enviados através de M_AXI
110         signal  mst_exec_state : state;
111
112         -- Ponteiro da matriz de entrada do mestre, usado no exemplo de FIFO.
113         signal read_pointer : integer range 0 to bit_num-1;
114
115         -- AXI-Stream Master
116         -- wait counter. The master waits for the user defined number of clock cycles before initiating
117      -- signal count        : std_logic_vector(WAIT_COUNT_BITS-1 downto 0); --(opcional)
118
```

```
119         -- streaming data valid
120     signal axis_tvalid        : std_logic;
121         -- streaming data valid delayed by one clock cycle
122         signal axis_tvalid_delay        : std_logic;
123         -- Last of the streaming data
124         signal axis_tlast        : std_logic;
125         -- Last of the streaming data delayed by one clock cycle
126         signal axis_tlast_delay        : std_logic;
127         -- FIFO implementation signals
128         signal stream_data_out        : std_logic_vector(C_M_AXIS_TDATA_WIDTH-1 downto 0);
129         signal tx_en        : std_logic;
130         -- Indica o fim das transacoes do mestre.
131         signal tx_done        : std_logic;
132 
133     -- AXI-Stream Slave
134     -- Quantidade de dados de entrada.
135         constant NUMBER_OF_INPUT_WORDS  : integer := 960;
136         -- bit_num_slave e o número mínimo de bits necessários para \\
137         -- endereçar uma profundidade de um FIFO de tamanho 'depth'.
138         constant bit_num_slave  : integer := clogb2(NUMBER_OF_INPUT_WORDS-1);
139 
140         -- Estados da FSM do slave
141         type state_slave is ( IDLE,  -- Estado idle.
142                         WRITE_FIFO); -- Neste estado, os dados do fluxo são recebidos através de S_AXIS
143         signal  mst_exec_state_slave : state_slave;
144 
145         -- TREADY do escravo.
146         signal axis_tready        : std_logic;
147         -- FIFO implementation signals
148         signal  byte_index : integer;
149         -- FIFO write enable
150         signal fifo_wren : std_logic;
151         -- FIFO full flag
152         signal fifo_full_flag : std_logic; --(opcional)
153         -- FIFO write pointer
154         signal write_pointer : integer range 0 to bit_num_slave-1 ;
155         -- sink has accepted all the streaming data and stored in FIFO
156         signal writes_done : std_logic;
157     -- type FIFO_TYPE is array (0 to (NUMBER_OF_INPUT_WORDS-1)) of std_logic_vector((C_S_AXIS_TDATA_WID
158     -- signal stream_data_fifo        :  FIFO_TYPE; -- usado para receber a entrada de dados (opcional)
159 
160         -- TLAST do escravo com um delay de um ciclo de clock.
161     signal S_AXIS_TLAST_DELAY: std_logic;
162 
163     -- CNN
164     signal sreset_conv4, sready_conv4, sready_conv4_delay: std_logic;
165     signal s_samples_conv4 : in_Conv4;
```

```
166     signal soutput_conv4: out_Conv2;
167     signal i : integer range 0 to (numberOfFilters-1):= numberOfFilters-1;
168     signal j : integer range 0 to (outConvs-1):= outConvs-1;
169 
170     signal k : integer range 0 to (numberOfFilters-1):= numberOfFilters-1;
171     signal l : integer range 0 to (outConvs-1):= outConvs-1;
172 
173 begin
174 
175         -- I/O Connections assignments
176 
177         M_AXIS_TVALID        <= axis_tvalid_delay;
178         M_AXIS_TDATA        <= stream_data_out;
179         M_AXIS_TLAST        <= axis_tlast_delay;
180         M_AXIS_TSTRB        <= (others => '1');
181     INTR_CONV4      <= tx_done;
182 
183         -- Control state machine implementation (master)
184         process(M_AXIS_ACLK)
185         begin
186           if (rising_edge (M_AXIS_ACLK)) then
187             if(M_AXIS_ARESETN = '0') then
188               -- Synchronous reset (active low)
189               mst_exec_state      <= IDLE;
190              -- count <= (others => '0');
191             else
192               case (mst_exec_state) is
193                 when IDLE     =>
194                   -- Aguarda o ready da camada conv.
195                   if (sready_conv4_delay = '1')then
196                     mst_exec_state <= SEND_STREAM;
197                   else
198                     mst_exec_state <= IDLE;
199                   end if;
200                   --else
201                   --  mst_exec_state <= IDLE;
202                   --end if;
203 
204                 when SEND_STREAM  =>
205                   -- Apos o ready, envia a matriz de saida
206                   if (tx_done = '1') then
207                     mst_exec_state <= IDLE;
208                   else
209                     mst_exec_state <= SEND_STREAM;
210                   end if;
211 
212                 when others    =>
```

```
213                 mst_exec_state <= IDLE;
214
215             end case;
216           end if;
217         end if;
218       end process;
219     --tvalid generation
220       --axis_tvalid e igual a 1 quando a FSM esta no estado de envio de dados
221       axis_tvalid <= '1' when (mst_exec_state = SEND_STREAM) else '0';
222
223       -- AXI tlast generation
224       -- axis_tlast e ativado quando os indices da matriz de saida estao iguais a 0 e o estado atual
225       axis_tlast <= '1' when ((l=0 and k=0) and (mst_exec_state = SEND_STREAM)) else '0';
226       -- Delay the axis_tvalid and axis_tlast signal by one clock cycle
227       -- to match the latency of M_AXIS_TDATA
228       process(M_AXIS_ACLK)
229       begin
230         if (rising_edge (M_AXIS_ACLK)) then
231           if(M_AXIS_ARESETN = '0') then
232             axis_tvalid_delay <= '0';
233             axis_tlast_delay <= '0';
234             sready_conv4_delay <= '0';
235           else
236         axis_tvalid_delay <= axis_tvalid;
237             axis_tlast_delay <= axis_tlast;
238             sready_conv4_delay <= sready_conv4;
239           end if;
240         end if;
241       end process;
242
243       -- processo de envio da matriz de saida.
244       process(M_AXIS_ACLK)
245       begin
246         if (rising_edge (M_AXIS_ACLK)) then
247           if(M_AXIS_ARESETN = '0') then
248             read_pointer <= 0;
249             tx_done  <= '0';
250           else
251             if (read_pointer <= NUMBER_OF_OUTPUT_WORDS-1) then
252               if (tx_en = '1') then
253               if l=0 and k>0 then
254                   l <= 29;
255                   k <= k-1;
256               elsif l=1 and k=0 then
257                   l <= 0;
258                   k <= 0;
259                   tx_done <= '1';
```

```
260                 elsif l=0 and k=0 then
261                     l <= 29;
262                     k <= 31;
263                     read_pointer <= 0;
264                     tx_done <= '0';
265                 else
266                     k <= k;
267                     l <= l-1;
268                 end if;
269                   read_pointer <= read_pointer + 1;
270                 end if;
271               end  if;
272             end  if;
273           end  if;
274         end process;
275 
276 
277         -- tx_en indica que a transacao e valida quando e igual a 1.
278 
279         tx_en <= M_AXIS_TREADY and axis_tvalid;
280 
281 
282         -- Streaming output data is read from FIFO
283           process(M_AXIS_ACLK)
284           variable  sig_one : integer := 1;
285           begin
286             if (rising_edge (M_AXIS_ACLK)) then
287               if(M_AXIS_ARESETN = '0') then
288                     stream_data_out <= std_logic_vector(to_unsigned(sig_one,C_M_AXIS_TDATA_WIDTH));
289               elsif (tx_en = '1') then -- && M_AXIS_TSTRB(byte_index)
290                 stream_data_out <= soutput_conv4(k, l) & "00000";
291               end if;
292              end if;
293            end process;
294 
295         -- AXIS SLAVE
296     S_AXIS_TREADY        <= axis_tready;
297         -- FSM
298         process(S_AXIS_ACLK)
299         begin
300           if (rising_edge (S_AXIS_ACLK)) then
301             if(S_AXIS_ARESETN = '0') then
302               -- Synchronous reset (active low)
303               mst_exec_state_slave      <= IDLE;
304             else
305               case (mst_exec_state_slave) is
306                 when IDLE     =>
```

```
307                         -- Aguarda tvalid
308                   if (S_AXIS_TVALID = '1')then
309                     mst_exec_state_slave <= WRITE_FIFO;
310                   else
311                     mst_exec_state_slave <= IDLE;
312                   end if;
313
314                 when WRITE_FIFO =>
315                   -- Recebe os dados
316                   if (writes_done = '1') then --quando finiliza retorna ao estado inicial
317                     mst_exec_state_slave <= IDLE;
318                   else
319                     mst_exec_state_slave <= WRITE_FIFO;
320                   end if;
321
322                 when others    =>
323                   mst_exec_state_slave <= IDLE;
324
325               end case;
326             end if;
327           end if;
328       end process;
329       -- AXI Streaming Sink
330       --
331       -- The example design sink is always ready to accept the S_AXIS_TDATA  until
332       -- the FIFO is not filled with NUMBER_OF_INPUT_WORDS number of input words.
333       axis_tready <= '1' when ((mst_exec_state_slave = WRITE_FIFO) and (write_pointer <= NUMBER_OF_IN
334
335   process(S_AXIS_ACLK)
336   begin
337       if (rising_edge (S_AXIS_ACLK)) then
338         S_AXIS_TLAST_DELAY <= S_AXIS_TLAST;
339       end if;
340   end process;
341
342       process(S_AXIS_ACLK)
343       begin
344         if (rising_edge (S_AXIS_ACLK)) then
345           if(S_AXIS_ARESETN = '0') then
346             write_pointer <= 0;
347             writes_done <= '0';
348           else
349             if (write_pointer <= NUMBER_OF_INPUT_WORDS-1) then
350               if (fifo_wren = '1') then
351                 write_pointer <= write_pointer + 1;
352               if j=0 and i>0 then
353                   j <= 29;
```

```
354                     i <= i-1;
355                 elsif j=1 and i=0 then
356                     j <= 0;
357                     i <= 0;
358                     writes_done <= '1';
359                 elsif j=0 and i=0 then
360                     j <= 29;
361                     i <= 31;
362                     write_pointer <= 0;
363                 else
364                     i <= i;
365                     j <= j-1;
366                     writes_done <= '0';
367                 end if;
368                 end if;
369               end  if;
370             end if;
371           end if;
372         end process;
373 
374         -- FIFO write enable generation
375         fifo_wren <= S_AXIS_TVALID and axis_tready;
376 
377           -- Streaming input data is stored in FIFO
378           process(S_AXIS_ACLK)
379           begin
380             if (rising_edge (S_AXIS_ACLK)) then
381               if (fifo_wren = '1') then
382                 s_samples_conv4(i, j) <= S_AXIS_TDATA(31 downto 5);
383               end if;
384             end  if;
385           end process;
386 
387         -- Add user logic here
388         sreset_conv4 <= not M_AXIS_ARESETN;
389 
390     CONV4: cnn_conv4b port map (    reset_conv4       => sreset_conv4,
391                                     clk               => M_AXIS_ACLK,
392                                     start_conv4       => S_AXIS_TLAST_DELAY,
393                                     ready_conv4       => sready_conv4,
394                                     samples_conv4     => s_samples_conv4,
395                                     output_conv4      => soutput_conv4);
396         -- User logic ends
397 
398 end implementation;
```

A.3 Testbench do bloco encapsulado

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use ieee.std_logic_unsigned.all;
4  use IEEE.NUMERIC_STD.ALL;
5  use work.fpupack.all;
6
7  entity design_1_wrapper_tb is
8  --  Port ( );
9  end design_1_wrapper_tb;
10
11 architecture Behavioral of design_1_wrapper_tb is
12 component design_1_wrapper is
13   port (
14     M00_AXIS_tdata : out STD_LOGIC_VECTOR ( 31 downto 0 );
15     M00_AXIS_tlast : out STD_LOGIC;
16     M00_AXIS_tready : in STD_LOGIC;
17     M00_AXIS_tstrb : out STD_LOGIC_VECTOR ( 3 downto 0 );
18     M00_AXIS_tvalid : out STD_LOGIC;
19     S00_AXIS_tdata : in STD_LOGIC_VECTOR ( 31 downto 0 );
20     S00_AXIS_tlast : in STD_LOGIC;
21     S00_AXIS_tready : out STD_LOGIC;
22     S00_AXIS_tstrb : in STD_LOGIC_VECTOR ( 3 downto 0 );
23     S00_AXIS_tvalid : in STD_LOGIC;
24     intr_conv4_0 : out STD_LOGIC;
25     m00_axis_aclk_0 : in STD_LOGIC;
26     m00_axis_aresetn_0 : in STD_LOGIC
27   );
28 end component design_1_wrapper;
29
30 signal  M00_AXIS_tdata : STD_LOGIC_VECTOR ( 31 downto 0 );
31 signal  M00_AXIS_tlast, M00_AXIS_tready, M00_AXIS_tvalid : STD_LOGIC;
32 signal  M00_AXIS_tstrb : STD_LOGIC_VECTOR ( 3 downto 0 ) := "1111";
33
34 signal  S00_AXIS_tdata : STD_LOGIC_VECTOR ( 31 downto 0 );
35 signal  S00_AXIS_tlast, S00_AXIS_tready, S00_AXIS_tvalid : STD_LOGIC;
36 signal  S00_AXIS_tstrb : STD_LOGIC_VECTOR ( 3 downto 0 ) := "1111";
37
38 signal  intr_conv4_0 : STD_LOGIC;
39
40 signal  m00_axis_aclk_0 : STD_LOGIC;
41 signal  m00_axis_aresetn_0 : STD_LOGIC;
42
43 signal i : integer range 0 to 31 := 31;
44 signal j : integer range 0 to 29 := 29;
45
```

```
46  signal s_samples_conv4 : in_Conv4 := (("00111110010110010010110000101","00111110011010101010111110111","001
47  ("00111101100000100101100101010","00000000000000000000000000000","00000000000000000000000000000","00000000000
48  ("00000000000000000000000000000","00000000000000000000000000000","00000000000000000000000000000","00000000000
49  ("00111110000001001000000000000","00111110000100010010101010","00111110001001110000010111","00111110001
50  ("00000000000000000000000000000","00000000000000000000000000000","00000000000000000000000000000","00000000000
51  ("00111110110000101000111010101","00111110110011010101000100","00111110110000000010000011","00111110101
52  ("00111110011111000111100111111","00111110110000110001000101010","00111110101110000100101100101","00111110101
53  ("00111101111010001101000010000","00111110000101001100110101","00111110010000111101110001000","00111110011
54  ("00000000000000000000000000000","00000000000000000000000000000","00000000000000000000000000000","00000000000
55  ("00000000000000000000000000000","00111101110101100100010000","00111110001001010010001110","00111110010
56  ("00111110011110110001001100","00111110100011100100110000","00111110011101000110100111","00111110001
57  ("00000000000000000000000000000","00000000000000000000000000000","00000000000000000000000000000","00000000000
58  ("00000000000000000000000000000","00000000000000000000000000000","00000000000000000000000000000","00000000000
59  ("00111110001110001101110011","00111110100100101010010010","00111110100110011010010010","00111110101
60  ("00111110000111011011000101","00111110100101001101000110","00111110101001100100000101","00111110101
61  ("00000000000000000000000000000","00000000000000000000000000000","00000000000000000000000000000","00000000000
62  ("00000000000000000000000000000","00000000000000000000000000000","00000000000000000000000000000","00000000000
63  ("00111110010100001010100101","00111110110001110011101001","00111110110101101100111100","00111110110
64  ("00000000000000000000000000000","00000000000000000000000000000","00000000000000000000000000000","00000000000
65  ("00111110101001011001011011","00111110101110011001011010","00111110101100010010110101","00111110100
66  ("00000000000000000000000000000","00000000000000000000000000000","00000000000000000000000000000","00000000000
67  ("00111110001100011101111111","00111110011001110101000110","00111110010111001111011110","00111110001
68  ("00111101001001010000111100","00111110001111001010101100","00111110011011100000010101","00111110100
69  ("00111101101011011111000001","00111110000111011011001011","00111110010001111001100110","00111110011
70  ("00111110000010011111100011","00111101110010010000000110","00111101100010110111101010","00111101001
71  ("00000000000000000000000000000","00111101101011011111011110","00111101111110001111010101","00111101111
72  ("00111110001010011000010010","00111110010000011010110111","00111110010010101100101011","00111110010
73  ("00111110101000111000000001","00111110100100111011101010","00111110100001011101001111","00111110011
74  ("00000000000000000000000000000","00000000000000000000000000000","00000000000000000000000000000","00000000000
75  ("00111110011001110011001000","00111110110011000010000011","00111110110100011011000010","00111110110
76  ("00000000000000000000000000000","00000000000000000000000000000","00000000000000000000000000000","00000000000
77  ("00111101101100110101001100","00111110011000100001101101","00111110011111100101000101","00111110100
78
79  begin
80  uut: design_1_wrapper port map (
81                                      M00_AXIS_tdata      =>  M00_AXIS_tdata,
82                                      M00_AXIS_tlast      =>  M00_AXIS_tlast,
83                                      M00_AXIS_tready     =>  M00_AXIS_tready,
84                                      M00_AXIS_tstrb      =>  M00_AXIS_tstrb,
85                                      M00_AXIS_tvalid     =>  M00_AXIS_tvalid,
86                                      S00_AXIS_tdata      =>  S00_AXIS_tdata,
87                                      S00_AXIS_tlast      =>  S00_AXIS_tlast,
88                                      S00_AXIS_tready     =>  S00_AXIS_tready,
89                                      S00_AXIS_tstrb      =>  S00_AXIS_tstrb,
90                                      intr_conv4_0    =>  intr_conv4_0,
91                                      S00_AXIS_tvalid     =>  S00_AXIS_tvalid,
92                                      m00_axis_aclk_0     =>  m00_axis_aclk_0,
```

```
93                                      m00_axis_aresetn_0  =>  m00_axis_aresetn_0
94                                  );
95  clk: process
96  begin
97     m00_axis_aclk_0 <= '0';
98     wait for 5ns;
99     m00_axis_aclk_0 <= '1';
100    wait for 5ns;
101 end process;
102
103 resetN: process
104 begin
105     m00_axis_aresetn_0 <= '0';
106     wait for 25ns;
107     m00_axis_aresetn_0 <= '1';
108     wait;
109 end process;
110
111 input_conv: process
112 begin
113 wait for 10ns;
114 if S00_AXIS_tvalid = '1' then
115     S00_AXIS_tdata <= s_samples_conv4(i, j) & "00000";
116         if j=0 and i>0 then
117             j <= 29;
118             i <= i-1;
119         elsif j=0 and i=0 then
120             j <= 29;
121             i <= 31;
122             wait;
123         else
124             i <= i;
125             j <= j-1;
126         end if;
127 end if;
128 end process;
129
130 ready: process
131 begin
132     M00_AXIS_tready <= '0';
133     S00_AXIS_tvalid <= '0';
134     S00_AXIS_tlast <= '0';
135     wait for 35ns;
136     M00_AXIS_tready <= '1';
137     S00_AXIS_tvalid <= '1';
138     wait for 9600ns;
139     S00_AXIS_tlast <= '1';
```

```
140     wait for 10ns;
141     S00_AXIS_tlast <= '0';
142     S00_AXIS_tvalid <= '0';
143     wait for 200ns;
144     wait;
145 end process;
146 
147 end Behavioral;
```

Apêndice B – Códigos de programação

B.1 CNN com a quarta camada em *hardware*

Código B.1 – CNN em C no ARM

```
1  #include <stdio.h>
2  #include "platform.h"
3  #include "xaxidma.h"
4  #include "xparameters.h"
5  #include "xil_printf.h"
6  #include "xil_cache.h"
7  #include "xil_io.h"
8  /*#include "conv4_file.h"*/
9  #include "sample_file.h"
10 #include "xscugic.h"
11 #include "xil_exception.h"
12 #include <stdlib.h>
13 #include <math.h>
14
15 #define numberOfFilters 32
16
17 #define sampleLength 60
18 #define firstFilterLength 7
19 #define poolLength 2
20 #define secondFilterLength 5
21 #define thirdFilterLength 3
22 #define fourthFilterLength 1
23 #define flattenLenght numberOfFilters*(sampleLength/poolLength)
24 #define firstFullyLenght 128
25 #define secondFullyLenght 64
26
27 #define NEW_MAX(x,y) ((x) >= (y)) ? (x) : (y)
28 #define RELU(x)(x>0?x:0)
29 #define SIGMOID(x) (1.f / (1 + exp(-x)))
30
31 float firstConvOutput[numberOfFilters][sampleLength] = {0};
32 float maxpoolingOutput[numberOfFilters][sampleLength/poolLength] = {0};
33 float secondConvOutput[numberOfFilters][sampleLength/poolLength] = {0};
34 float thirdConvOutput[numberOfFilters][sampleLength/poolLength] = {0};
35 float fourthConvOutput[numberOfFilters][sampleLength/poolLength] = {0};
36 float flattenOutput[flattenLenght] = {0};
37 float firstFullyOutput[firstFullyLenght] = {0};
38 float secondFullyOutput[secondFullyLenght] = {0};
39 float output = 0;
40
41 float sampleAdjusted[sampleLength+(firstFilterLength-1)] = {0};
42 float secondConvInput[numberOfFilters][sampleLength/poolLength+(secondFilterLength-1)] = {0};
43 float thirdConvInput[numberOfFilters][sampleLength/poolLength+(thirdFilterLength-1)] = {0};
44
45 float out_data[conv4_input_length] = {0};
46 XAxiDma_Config *DMA_config;
47 XAxiDma DMA;
48
```

```
49
50 //Functions==================
51 void adjust_input(float samples[])
52 {
53   for (int i=0; i<sampleLength; i++)
54   {
55     sampleAdjusted[i+(firstFilterLength/2)] = samples[i];
56     // printf("%.32f, ",sampleAdjusted[i+(firstFilterLength/2)] );
57   }
58   // printf("\n");
59 }
60
61 // SIPO (7 saidas paralelas)
62 void conv1()
63 {
64   // j - number
65   for(int j=0; j<numberOfFilters; j++)
66   {
67     for(int i=0; i<sampleLength; i++)
68     {
69       for(int k=0; k<firstFilterLength; k++)
70       {
71         firstConvOutput[j][i] += (sampleAdjusted[i+k]*firstConvFilter[j][k]);
72       }
73       firstConvOutput[j][i] += firstConvBias[j];
74       firstConvOutput[j][i] = RELU(firstConvOutput[j][i]);
75       // printf("%.32f, ", firstConvOutput[j][i]);
76     }
77     // printf("\n");
78   }
79   // printf("\n");
80   // printf("1conv: %f\n",firstConvOutput[1][21]);
81 }
82
83
84 void maxpooling()
85 {
86   for(int j=0; j<numberOfFilters; j++)
87   {
88     for(int i=0; i<sampleLength; i=i+poolLength)
89     {
90       maxpoolingOutput[j][i/poolLength] =
             NEW_MAX(firstConvOutput[j][i],firstConvOutput[j][i+1]);
91       // printf("%f\t", maxpoolingOutput[j][i/2]);
92     }
93     // printf("\n");
94   }
95   // printf("\n");
96   // printf("max: %f\n",maxpoolingOutput[1][10]);
97
98 }
99
100 void adjustSecondConvInput(float samples[numberOfFilters][sampleLength/poolLength])
101 {
102   for(int j=0; j<numberOfFilters; j++)
103   {
104     for(int i=0; i<sampleLength/poolLength; i++)
105     {
106       secondConvInput[j][i+(secondFilterLength/2)] = samples[j][i];
```

```
107       }
108     }
109   }
110
111
112   void conv2()
113   {
114   // l - the number of samples
115     for(int l=0; l<32; l++)
116     {
117       //i - size of the samples
118       for(int i=0; i<sampleLength/poolLength; i++)
119       {
120         //j - number of filters
121         for(int j=0; j<numberOfFilters; j++)
122         {
123           //k - size of the filters
124           for(int k=0; k<secondFilterLength; k++)
125           {
126             secondConvOutput[l][i] += (secondConvInput[j][i+k]*secondConvFilter[l][j][k]);
127             //printf("%f\t", secondConvOutput[l][i]);
128           }
129         }
130         secondConvOutput[l][i] += secondConvBias[l];
131         secondConvOutput[l][i] = RELU(secondConvOutput[l][i]);
132
133       }
134       //printf("\n");
135     }
136     // printf("\n");
137   }
138
139   void adjustThirdConvInput(float samples[numberOfFilters][sampleLength/poolLength])
140   {
141     for(int j=0; j<numberOfFilters; j++)
142     {
143       for(int i=0; i<sampleLength/poolLength; i++)
144       {
145         thirdConvInput[j][i+(thirdFilterLength/2)] = samples[j][i];
146         }
147     }
148
149   }
150
151
152   void conv3()
153   {
154
155     // l - is the number of samples
156     for(int l=0; l<32; l++)
157     {
158       //i - size of the samples
159       for(int i=0; i<sampleLength/poolLength; i++)
160       {
161         //j - number of filters
162         for(int j=0; j<numberOfFilters; j++)
163         {
164           //k - size of the filters
165           for(int k=0; k<thirdFilterLength; k++)
```

```
166         {
167           thirdConvOutput[l][i] += (thirdConvInput[j][i+k]*thirdConvFilter[l][j][k]);
168         }
169         // printf("%d\t", secondLayerOutput[j][i]);
170       }
171       thirdConvOutput[l][i] += thirdConvBias[l];
172       thirdConvOutput[l][i] = RELU(thirdConvOutput[l][i]);
173 
174       // printf("%f\t", thirdConvOutput[l][i]);
175     }
176     // printf("\n");
177   }
178   // printf("\n");
179 
180 }
181 
182 static int DMA_setup(XAxiDma_Config *DMA_config, XAxiDma *DMA){
183   int status;
184   float in_data[conv4_input_length];
185   int j = 0;
186   int k = 0;
187 
188 for(int l = 0; l < 32; l++){
189   for (int i = 0; i < 30; i++){
190     in_data[j] = thirdConvOutput[l][i];
191     j++;
192   }
193 }
194   DMA_config = XAxiDma_LookupConfigBaseAddr(XPAR_AXI_DMA_0_BASEADDR);
195 
196   status = XAxiDma_CfgInitialize(DMA, DMA_config);
197   if(status != XST_SUCCESS){
198     print("Inicializacao DMA falhou.\n");
199     return -1;
200   }
201   //print("Inicializacao DMA.\n");
202 
203   XAxiDma_IntrDisable(DMA, XAXIDMA_IRQ_ALL_MASK,  XAXIDMA_DEVICE_TO_DMA);
204   xil_printf("sem intr do DMA\r\n");
205   XAxiDma_IntrDisable(DMA, XAXIDMA_IRQ_ALL_MASK,  XAXIDMA_DMA_TO_DEVICE);
206 
207   if(XAxiDma_HasSg(DMA)){
208     xil_printf("DMA no modo SG \r\n");
209     return XST_FAILURE;
210   }
211   //xil_printf("DMA no modo simples \r\n");
212 
213   Xil_DCacheFlushRange((u32)in_data, 960*sizeof(float)); /*Libera os arquivos de cache*/
214   Xil_DCacheInvalidateRange((u32)out_data, 960*sizeof(float));
215 
216   status = XAxiDma_SimpleTransfer(DMA, (u32)out_data, 960*sizeof(float),
       XAXIDMA_DEVICE_TO_DMA);
217   if(status != XST_SUCCESS){
218       print("Transferencia da conv4 falhou.\n");
219       return -1;
220     }
221   status = XAxiDma_SimpleTransfer(DMA, (u32)in_data, 960*sizeof(float),
       XAXIDMA_DMA_TO_DEVICE);
222   if(status != XST_SUCCESS){
```

```
223         print("Transferencia para conv4 falhou.\n");
224         return -1;
225       }
226     while(XAxiDma_Busy(DMA, XAXIDMA_DMA_TO_DEVICE) || XAxiDma_Busy(DMA, XAXIDMA_DEVICE_TO_DMA));
227
228     print("Transferencia conv4 finalizada.\n");
229
230     for(int l = 0; l < 32; l++){
231       for (int i = 0; i < 30; i++){
232         fourthConvOutput[l][i] = out_data[k];
233         k++;
234       }
235     }
236
237     return XST_SUCCESS;
238 }
239
240 /*void conv4()
241 {
242 //  FILE *fptr;
243 //  fptr = fopen("fourthConv_out.txt","w");
244   // l - is the number of samples
245   for(int l=0; l<32; l++)
246   {
247     //i - size of the samples
248     for(int i=0; i<sampleLength/poolLength; i++)
249     {
250       //j - number of filters
251       for(int j=0; j<numberOfFilters; j++)
252       {
253         //k - size of the filters
254         for(int k=0; k<fourthFilterLength; k++)
255         {
256           fourthConvOutput[l][i] += (thirdConvOutput[j][i+k]*fourthConvFilter[l][j][k]);
257         }
258         // printf("%d\t", secondLayerOutput[j][i]);
259       }
260       fourthConvOutput[l][i] += fourthConvBias[l];
261       fourthConvOutput[l][i] = RELU(fourthConvOutput[l][i]);
262       //fprintf(fptr,"%f, ",fourthConvOutput[l][i]);
263       // printf("%f\t", fourthConvOutput[l][i]);
264     }
265     // printf("\n");
266   }
267   // printf("\n");
268 //fclose(fptr);
269 }*/
270
271
272 void flatten()
273 {
274   int count = 0;
275   for(int j=0; j<numberOfFilters; j++)
276   {
277     for(int i=0; i<sampleLength/poolLength; i++)
278     {
279       flattenOutput[count] = fourthConvOutput[j][i];
280       // printf("%d\n", flattenOutput[count]);
281       count++;
```

```
282       }
283     }
284     // printf("flatten: %f\n",flattenOutput[1]);
285 
286 }
287 
288 void fully1()
289 {
290     int offset = 0;
291     // j - number of neurons in this layer
292     for(int j=0; j<firstFullyLenght; j++)
293     {
294       // i - number of neurons in the last layer
295       for (int i=0; i<flattenLenght; ++i)
296       {
297         offset = j * firstFullyLenght + i;
298         firstFullyOutput[j] += (flattenOutput[i]*a[offset]);
299       }
300       firstFullyOutput[j] += firstFullyBias[j];
301       firstFullyOutput[j] = RELU(firstFullyOutput[j]);
302       // printf("%d\n", fully1Output[j]);
303     }
304     // printf("\n");
305 }
306 
307 void fully2()
308 {
309     int offset = 0;
310     // j - number of neurons in this layer
311     for(int j=0; j<secondFullyLenght; j++)
312     {
313       // i - number of neurons in the last layer
314       for (int i=0; i<firstFullyLenght; ++i)
315       {
316         offset = j * firstFullyLenght + i;
317         secondFullyOutput[j] += (firstFullyOutput[i]*secondFullyParameters[offset]);
318       }
319       secondFullyOutput[j] += secondFullyBias[j];
320       secondFullyOutput[j] = RELU(secondFullyOutput[j]);
321       // printf("%f\t", secondFullyOutput[j]);
322     }
323     // printf("\n");
324 }
325 
326 void outputLayer()
327 {
328     // i - numbert of neurons in the last layer
329     for (int i=0; i<secondFullyLenght; ++i)
330     {
331       output += (secondFullyOutput[i]*outputParameter[i]);
332     }
333 
334     output += outputBias;
335     output = SIGMOID(output);
336     printf("output: %f\n", output);
337 }
338 
339 int main()
340 {
```

```
341   u32 status;
342
343   adjust_input(sample);
344
345   conv1();
346
347   maxpooling();
348
349   adjustSecondConvInput(maxpoolingOutput);
350
351   conv2();
352
353   adjustThirdConvInput(secondConvOutput);
354
355   conv3();
356
357   // Not necessary to adjust the sample, for the filterLenght in the fourth conv layer is 1
358
359   //void conv4();
360   status = DMA_setup(DMA_config, &DMA);
361   if(status != XST_SUCCESS){
362     print("Setup DMA falhou.\n");
363     return -1;
364   }
365 //   print("Setup DMA....\n");
366
367   flatten();
368
369   fully1();
370
371   fully2();
372
373   outputLayer();
374
375     return 0;
376 }
```

Anexos

Anexo A – Tabela de sinais do AXI4

Tabela A.1 – Sinais de transferência AXI4. Fonte: (AMD, 2024).

AXI4 Sinais				
Write Address	Write Data	Write Response	Read Address	Read Data
AWID[M:0]	WDVALID	BRID[M:0]	ARID[M:0]	RDID[M:0]
AWVALID	WDREADY	BVALID	ARVALID	RDVALID
AWREADY	WDATA[N:0]	BREADY	ARREADY	RDREADY
AWADDR[31:0]	WDSTRB[N/8:0]	BRESP[1:0]	ARADDR[31:0]	RDATA[N:0]
AWLEN[7:0]	WDLAST		ARLEN[7:0]	RDRESP[1:0]
AWSIZE[2:0]			ARSIZE[2:0]	RDLAST
AWPROT[2:0]			ARPROT[2:0]	
AWBURST[1:0]			ARBURST[1:0]	
AWLOCK			ARLOCK	
AWCACHE[3:0]			ARCACHE[3:0]	
AWREGION[3:0]			ARREGION[3:0]	
AWQOS[3:0]			ARQOS[3:0]	

Anexo B – Tabela de Sinais do AXI4-Lite

Tabela B.1 – Sinais de transferência AXI4-Lite. Fonte: (AMD, 2024).

AXI4-Lite Sinais				
Write Address	Write Data	Write Response	Read Address	Read Data
AWVALID	WDVALID	BVALID	ARVALID	RDVALID
AWREADY	WDREADY	BREADY	ARREADY	RDREADY
AWADDR[31:0]	WDATA[N:0]	BRESP[1:0]	ARADDR[31:0]	RDATA[N:0]
AWPROT[2:0]	WDSTRB[N/8:0]		ARPROT[2:0]	RDRESP[1:0]
AWCACHE[3:0]			ARCACHE[3:0]	