


Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia Eletrônica

**Implementação em FPGA da Camada
Convolutacional de um Algoritmo de Redes
Neurais para um Módulo Estimador da
Frequência Cardíaca Fetal**

Autor: Misael de Souza Andrade
Orientador: Prof. Dr. Gilmar Silva Beserra

Brasília, DF
2022



Misael de Souza Andrade

Implementação em FPGA da Camada Convolucional de um Algoritmo de Redes Neurais para um Módulo Estimador da Frequência Cardíaca Fetal

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).

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Gilmar Silva Beserra

Coorientador: Prof. Dr. Daniel Mauricio Muñoz Arboleda

Brasília, DF

2022

Misael de Souza Andrade

Implementação em FPGA da Camada Convolucional de um Algoritmo de Redes Neurais para um Módulo Estimador da Frequência Cardíaca Fetal/ Misael de Souza Andrade. – Brasília, DF, 2022-

95 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Gilmar Silva Beserra

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2022.

1. Redes Neurais. 2. FPGA. I. Prof. Dr. Gilmar Silva Beserra. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Implementação em FPGA da Camada Convolucional de um Algoritmo de Redes Neurais para um Módulo Estimador da Frequência Cardíaca Fetal

CDU

Misael de Souza Andrade

Implementação em FPGA da Camada Convolutacional de um Algoritmo de Redes Neurais para um Módulo Estimador da Frequência Cardíaca Fetal

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, DF, 12 de maio de 2022:

Prof. Dr. Gilmar Silva Beserra
Orientador

Prof. Dr. Daniel Mauricio Muñoz Arboleda
Co-Orientador
Convidado 1

Prof. Dr. Daniel Chaves Café
Examinador
Convidado 2

Brasília, DF
2022

Agradecimentos

Agradeço em primeiro lugar, a Deus, pela minha vida, e por me permitir ultrapassar todos os obstáculos encontrados ao longo da realização deste trabalho. Aos meus pais, que por meio de muito trabalho, esforço e amor me deram a oportunidade de chegar até aqui. A minha esposa que me incentivou e apoiou nos momentos difíceis e compreendeu a minha ausência enquanto eu me dedicava à realização deste trabalho. Por fim, ao meu orientador e a todos os professores do curso, por sua dedicação, paciência e conhecimento.

Resumo

A necessidade de identificar padrões em sinais com alta confiabilidade vem crescendo de forma exponencial, devido à enorme quantidade de dados dispostos no mundo atual. A aplicação de Redes Neurais Artificiais para atingir esse objetivo é crescente e a redução no custo de implantação é constante à medida em que se popularizam ferramentas computacionais que permitem a inserção de novas técnicas para solução de problemas complexos. Devido ao alto custo computacional da implementação das redes neurais, torna-se necessária a abordagem de aceleração de algoritmos com FPGA (*Field Programmable Gate Arrays*, onde a parte com maior exigência de processamento é executada em hardware. Por meio deste embasamento, este trabalho apresenta o conceito de aceleração de algoritmos por meio da implementação em FPGA da primeira camada convolucional de uma Rede Neural Convolucional, que estima o ECG fetal (fECG) em um sinal de ECG abdominal (aECG). O projeto da camada convolucional em FPGA mostrou-se 5.85 vezes mais rápido do que o modelo em software executado no microprocessador ARM-Zynq. Também foi avaliado a eficiência de cálculo da convolução, que mostrou um erro médio quadrático calculado de apenas 0.000123 para os resultados entre hardware e software, usando uma representação aritmética de ponto flutuante de 27 bits e 32 bits, respectivamente. Os resultados satisfatórios obtidos neste trabalho confirmam a importância da aceleração em hardware de algoritmos de redes neurais convolucionais.

Palavras-chave: fECG. Redes Neurais. CNN. VHDL. FPGA. Aceleração de algoritmos. SoC.

Abstract

The need to identify patterns in signals with high reliability has been growing exponentially, due to the huge amount of data available in the current world. The application of Artificial Neural Networks to achieve this goal is increasing and the reduction in the cost of implementation is constant as computational tools are popularized that allow the insertion of new techniques to solve complex problems. Due to the high computational cost of implementing neural networks, it is necessary to approach algorithms with FPGA (*Field Programmable Gate Arrays*), where the part with the highest processing demand is executed in hardware. , this work presents the concept of algorithm acceleration through the FPGA implementation of the first convolutional layer of a Convolutional Neural Network, which estimates the fetal ECG (fECG) in an abdominal ECG signal (aECG). FPGA proved to be 5.85 times faster than the software model run on the ARM-Zynq microprocessor. The convolution calculation efficiency was also evaluated, which showed a calculated mean square error of only 0.000123 for the results between hardware and software, using a 27-bit and 32-bit floating point arithmetic representation, respectively. The satisfactory results obtained in this work confirm the importance of ia of hardware acceleration of convolutional neural network algorithms.

Key-words: fECG. Neural Network. CNN. VHDL. FPGA. Algorithm acceleration. SoC.

Lista de ilustrações

Figura 1 – Sistema de aquisição aECG e extração do fECG para estimativa do FHR. FONTE: (JAGANNATH; SELVAKUMAR, 2014)	22
Figura 2 – Protótipo de sistema para estimativa da FHR através da aquisição de sinais aECG. Em desenvolvimento pela Universidade de Brasília - FGA.	23
Figura 3 – Estágios de desenvolvimento do coração fetal de (a) a (f): a) tubos endocárdicos. b) comunicações entre tubos endocárdicos. c) arcos aórticos, sacos aórticos, mioepicárdio, átrios e seios venosos d) arcos aórticos, bulbus cordis, ventrículo e átrio e) arcos aórticos I, arcos aórticos II, átrios primitivos, ventrículos primitivos f) veia cava superior, aorta, tronco pulmonar, átrios, ventrículos, veia cava inferior. FONTE: (SAMENI; CLIFFORD, 2011)	25
Figura 4 – O ciclo de ativação do coração fetal. FONTE: (SAMENI; CLIFFORD, 2011)	26
Figura 5 – Ilustração dos sistemas de extração AES e CS. FONTE: (KAHANKOVA et al., 2020)	27
Figura 6 – Composição de um neurônio biológico. FONTE: (BRITO, 2019)	28
Figura 7 – Modelo não linear de um neurônio. FONTE: (HAYKIN, 2001)	29
Figura 8 – Modelo de uma rede neural convolucional e suas diferentes camadas. FONTE: (VARGAS; CARVALHO; VASCONCELOS, 2016)	30
Figura 9 – Diagrama esquemático do algoritmo de treinamento de retropropagação e modelo de neurônio típico. FONTE: (KIM; SEO, 2015)	32
Figura 10 – Arquitetura interna de um chip FPGA típico. FONTE: (BAJAJ; FAHMY, 2015)	33
Figura 11 – Diagrama de Blocos estrutural do Zynq-7000 Soc. FONTE: (XILINX, 2021)	35
Figura 12 – Vivado Design Suite IP Flow. FONTE: Adaptado de (FEIST, 2012)	41
Figura 13 – Símbolo do ILA Core. FONTE: (XILINX, 2016)	41
Figura 14 – comunicação Host-Target para o xSDK. FONTE: (XILINX, 2019)	42
Figura 15 – Basys 3 Artix-7 FPGA. FONTE: (DIGILENT, 2016)	43
Figura 16 – Sinais mECG e fECG sem ruído. Fonte: Autor.	45
Figura 17 – Sinal aECG final gerado no MatLab. Fonte: Autor.	46
Figura 18 – Arquitetura da Rede Neural Convolucional. FONTE: (JUNIOR, 2018)	47
Figura 19 – Diagrama de blocos do microprocessador ARM-Zynq instanciado como IP. FONTE: Autor.	48

Figura 20 – Código em C para a primeira camada convolucional. [<i>numberOfFilters</i> =32; <i>sampleLength</i> =60; <i>firstFilterLength</i> =7.] FONTE: Adaptado de (JUNIOR, 2018)	49
Figura 21 – Arquitetura proposta para iteração do loop interno da primeira camada convolucional. FONTE: Autor.	51
Figura 22 – Esquemático RTL do bloco do loop interno da convolução descrito em VHDL com 7 multiplicadores, 7 somadores e registradores para saída. A disposição dos blocos foi definida pela ferramenta do Vivado. FONTE: Autor.	51
Figura 23 – Diagrama de blocos da FSM descrita em VHDL utilizando 3 processos. FONTE: Autor.	52
Figura 24 – Diagrama de estados simplificado da FSM. FONTE: Autor.	53
Figura 25 – Diagrama de blocos do módulo principal. FONTE: Autor.	54
Figura 26 – Diagrama de blocos do topmodule com instancição do ILA core, camada convolucional, BRAM e FSM para controle. FONTE: Autor.	54
Figura 27 – Análise de <i>profiling</i> da CNN com o <i>gprof</i> no ARM-Zynq. FONTE: Autor.	55
Figura 28 – Simulação comportamental do bloco do loop interno da convolução: <i>conv7samples</i> . FONTE: Autor.	57
Figura 29 – Simulação comportamental do módulo principal:registro dos resultados da camada convolucional na matriz de saída 60x32. FONTE: Autor.	58
Figura 30 – Simulação comportamental do módulo principal:transição dos estados e contadores de índices matriciais i e j. FONTE: Autor.	59
Figura 31 – Funcionamento da leitura e registro temporário das 66 amostras de entrada da primeira camada convolucional. FONTE: Autor.	59
Figura 32 – Implementação da primeira camada convolucional com testes pelo ILA Core. Tempo geral da convolução. FONTE: Autor.	60
Figura 33 – Implementação da primeira camada convolucional com testes pelo ILA Core. Tempo de saída de resultado da convolução. FONTE: Autor.	61
Figura 34 – Distribuição de erro numérico para os resultados da primeira camada convolucional implementada em hardware, utilizando FPU de 27 bits. FONTE: Autor.	62
Figura 35 – Representação gráfica dos recursos utilizados no FPGA da Basys3 para implementação do circuito + ILA. FONTE: Autor.	62
Figura 36 – Report de utilização de recursos pelo circuito. FONTE: Autor.	63
Figura 37 – Report de consumo de energia da Basys3 pelo circuito + ILA. FONTE: Autor.	63
Figura 38 – Report de <i>timing</i> do circuito implementado no FPGA + ILA. FONTE: Autor.	63

Lista de tabelas

Lista de abreviaturas e siglas

FHR	<i>Fetal Heart Rate</i> (Frequência Cardíaca Fetal)
ECG	Eletrocardiograma
NI-fECG	<i>Non Invasive fetal ECG</i> (Eletrocardiografia Fetal Não Invasiva)
fECG	Eletrocardiograma Fetal
aECG	Eletrocardiograma Abdominal
mECG	Eletrocardiograma Materno
FGA	Faculdade do Gama da Universidade de Brasília
BPM	Batimento Por Minuto
SA	Nó Sinoatrial
AV	Nó Atrioventricular
VFC	Variabilidade da Frequência Cardíaca
CTG	Cardiotocografia
AES	<i>Abdominal Electrode-Sourced</i> (fonte de Eletrodo Abdominal)
CS	<i>Combined Source</i> (Fonte Combinada)
RNAs	Redes Neurais Artificiais
CNN	<i>Convolutional Neural Network</i> (Rede Neural Convolucional)
RELU	<i>Rectified Linear Unit</i> (Unidade Linear Retificadora)
FPGA	<i>Field-Programmable Gate Array</i>
CLB	<i>Configurable Logic Block</i> (Bloco Lógico Configurável)
LUTs	<i>Look-Up Tables</i>
IOB	<i>In/Out Block</i> (Bloco de entrada e saída)
SB	<i>Switch Box</i> (Bloco de seletoras)
BRAM	<i>Block Random Access Memory</i> (Bloco de Memória de acesso Aleatório)

DSP	<i>Digital Signal Process</i> (Processador de Sinal Digital)
HDL	<i>Hardware Description Language</i> (Linguagem de Descrição de Hardware)
VHDL	<i>VHSIC Hardware Description Language</i> (VHSIC Linguagem de Descrição de Hardware)
ASIC	<i>Application Specific Integrated Circuits</i> (Circuitos Integrados de Aplicação Específica)
CPU	<i>Central Process Unit</i> (Unidade de Processamento Central)
SoC	<i>System on a Chip</i> (Sistema em um chip)
ARM	<i>Advanced RISC Machine</i>
ASSP	<i>Application-specific standard parts</i> (Circuitos integrados para uma aplicação específica padrão)
ADALINE	<i>Adaptative Linear Netwok</i>
TDL	<i>Tapped Delay Line</i>
IP	<i>Intellectual Property</i> (Blocos de Propriedade Intelectual)
MatLab	<i>MATrix LABoratory</i>
XSDK	<i>Xilinx Software Development Kit</i>
IDE	<i>Integrated Development Environment</i>
USB	<i>Universal Serial Bus</i>
VGA	<i>Video Graphics Array</i>
LEDs	<i>Light-Emitting Diodes</i>
PLL	<i>Phase Lock Loop</i> (Bloqueio de fase em loop)
xADC	Conversor analógico-digital on-chip
RTL	<i>Register Transfer Level</i>
SIPO	<i>Single Input Parallel Output</i>
ns	Nanossegundos

Lista de símbolos

Γ	Letra grega Gama
Λ	Lambda
ζ	Letra grega minúscula zeta
\in	Pertence

Sumário

1	INTRODUÇÃO	21
1.1	Contextualização	21
1.2	Definição do Problema	23
1.3	Objetivos	23
1.3.1	Objetivos Gerais	23
1.3.2	Objetivos Específicos	24
1.4	Estrutura do Trabalho	24
2	FUNDAMENTAÇÃO TEÓRICA	25
2.1	ECG Fetal	25
2.1.1	Formação e Atividade Cardíaca	25
2.1.2	Estimativa da FHR	26
2.2	Redes Neurais Artificiais	28
2.2.1	Definição	28
2.2.2	Redes Neurais Convolucionais	29
2.2.3	Treinamento por <i>Backpropagation</i>	31
2.3	FPGAs	32
2.3.1	Aceleração de Algoritmos	34
2.3.2	SoCs	34
2.4	Estado da Arte	36
3	ASPECTOS METODOLÓGICOS	39
3.1	Metodologia	39
3.2	Ferramentas	39
3.2.1	MatLab	40
3.2.2	Vivado	40
3.2.3	ILA Core	41
3.2.4	XSDK	41
3.2.5	Basys 3 Artix-7	42
4	IMPLEMENTAÇÃO DA ARQUITETURA PROPOSTA	45
4.1	Sinal aECG gerado no MatLab	45
4.2	Descrição da arquitetura proposta	46
4.3	Implementação da CNN em C no ARM	47
4.4	Descrição da Primeira Camada Convolucional em VHDL	48
4.4.1	Componente do <i>loop</i> interno	50

4.4.2	Módulo Principal	52
4.5	Implementação do projeto em Hardware com o ILA	53
5	RESULTADOS E DISCUSSÕES	55
5.1	Análise de <i>Profiling</i>	55
5.2	Simulações Comportamentais	56
5.2.1	<i>Testbench</i> do componente do loop interno	56
5.2.2	<i>Testbench</i> do módulo principal	57
5.3	Testes de implementação em hardware com o ILA Core	58
5.4	Análise de recursos, energia e <i>timing</i>	61
5.5	Hardware vs Software	64
6	CONCLUSÃO	65
6.1	Trabalhos futuros	65
	REFERÊNCIAS	67
	APÊNDICES	71
	APÊNDICE A – CÓDIGO VHDL DA BIBLIOTECA <i>FPUPACK</i>	73
	APÊNDICE B – CÓDIGO VHDL QUE DESCREVE O LOOP INTERNO DA PRIMEIRA CAMADA CONVOLUCIONAL.	77
	APÊNDICE C – CÓDIGO VHDL DO MÓDULO PRINCIPAL DA CAMADA CONVOLUCIONAL COM FSM.	83
	APÊNDICE D – CÓDIGO VHDL DO MÓDULO QUE INSTANCIA O ILA PARA TESTES EM HARDWARE.	91

1 Introdução

1.1 Contextualização

A frequência cardíaca fetal (FHR - *Fetal Heart Rate*) é o intervalo entre as batidas consecutivas do coração fetal em desenvolvimento por unidade de tempo, ou seja, representa o batimento cardíaco e a atividade do coração. Ele fornece informações para determinar as condições de saúde do feto o que ajuda no prognóstico gestacional. De acordo com (SAKURAI; LIMA; FARIA, 2001), a curva de normalidade para a FHR entre a 10^a e a 14^a semana de gestação é entre 136 e 178 BPM, havendo diminuição progressiva e significativa da FHR com o avanço da idade gestacional.

Em 1953, surgiu o primeiro método para monitorar continuamente a FHR por meio da eletrocardiografia fetal não invasiva (NI-fECG). Em paralelo, surgiu a cardiotocografia fetal ultrassônica, outro método não invasivo para monitoramento da FHR e das contrações uterinas. Já em relação aos métodos invasivos, tem-se o método de monitoramento fetal eletrônico interno, onde a FHR é determinada de forma mais precisa usando sinais medidos por um eletrodo de couro cabeludo fetal (KAHANKOVA et al., 2020).

O NI-fECG permanece como o método alternativo mais promissor para monitoramento fetal contínuo, pois a partir dele é possível obter informações fisiológicas únicas para identificar sofrimento fetal incapazes de serem obtidos pelo método de cardiotocografia, devido à natureza de suas medições. Além de ser um método não invasivo, em que não há exposição da mãe e do feto a radiação, o sinal do fECG (Eletrocardiograma Fetal) carrega informações valiosas, como estados patológicos (isquemia miocárdica, hipóxia intraparto ou acidose metabólica) que são identificadas como alterações na morfologia da forma de onda de fECG (segmento ST, intervalo QT). A partir do fECG também é possível estabelecer diagnósticos para má formação do coração e bradicardia (KAHANKOVA et al., 2020).

Sendo assim, o método do NI-fECG foi o mais adequado para abordagem de monitoramento do FHR do presente projeto. O seu sistema de aquisição, consiste em posicionar os eletrodos no abdômen materno e o sinal resultante coletado é o eletrocardiograma abdominal (aECG), que é composto pelo eletrocardiograma materno (mECG), pelo fECG e por ruídos diversos, como por exemplo das contrações uterinas. A partir do processamento do aECG, pode-se extrair o fECG e aplicar algoritmos de estimação para se obter a FHR (JAGANNATH; SELVAKUMAR, 2014).

A figura 1 apresenta o sistema de aquisição do aECG, junto ao processo de extração do fECG.

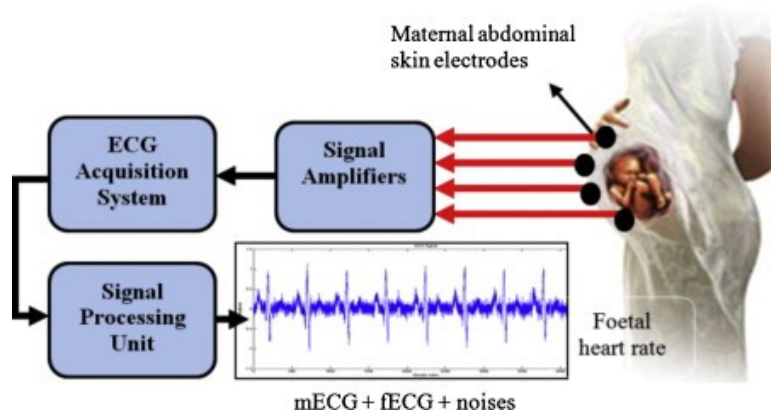


Figura 1 – Sistema de aquisição aECG e extração do fECG para estimativa do FHR.
 FONTE: (JAGANNATH; SELVAKUMAR, 2014)

Está sendo desenvolvido na Faculdade do Gama da Universidade de Brasília (FGA), um protótipo para estimar a FHR a partir do fECG utilizando um dispositivo reconfigurável, com co-projeto hardware-software, para a implementação e aceleração dos algoritmos de processamento de sinais. O sistema completo, vide Figura 2, é composto por 3 blocos:

- Aquisição: neste bloco serão coletadas as amostras de aECG. Será utilizada uma placa de front-end analógico para ECG do Medical Development Kit TMS320VC5505 da Texas Instruments, que efetuará a leitura dos sensores de ECG (eletrocardiograma) posicionados no abdômen da gestante e enviará os sinais amostrados para o bloco de processamento. Atualmente, o sistema de aquisição está configurado somente para escrever os dados em um cartão de memória (TUTIDA, 2016);
- Processamento: este bloco contém um kit com um dispositivo FPGA (*Field Programmable Gate Array*), no qual um processador softcore embarcado será utilizado para aceleração de algoritmos a partir da exploração das possibilidades de paralelismo oferecidas pela implementação em hardware e software. Até o presente momento, foram implementadas duas abordagens para processar o aECG e realizar a estimativa da FHR: filtro adaptativo (BARBOSA, 2016) e redes neurais (JUNIOR, 2018);
- Comunicação: este bloco também será implementado no mesmo processador softcore embarcado em FPGA e realizará a comunicação sem fio com um dispositivo móvel. A implementação realizada até o momento permite realizar a transmissão da FHR via Bluetooth para um smartphone com sistema Android, que por sua vez contém um aplicativo que recebe os dados e emite alarmes caso os mesmos ultrapassem os limites pré-definidos (RODRIGUES, 2016).

O escopo do presente projeto está em implementar um acelerador de hardware em FPGA para o algoritmo da Redes Neural Convolutacional, proposta por (JUNIOR, 2018), para o bloco de processamento do protótipo.

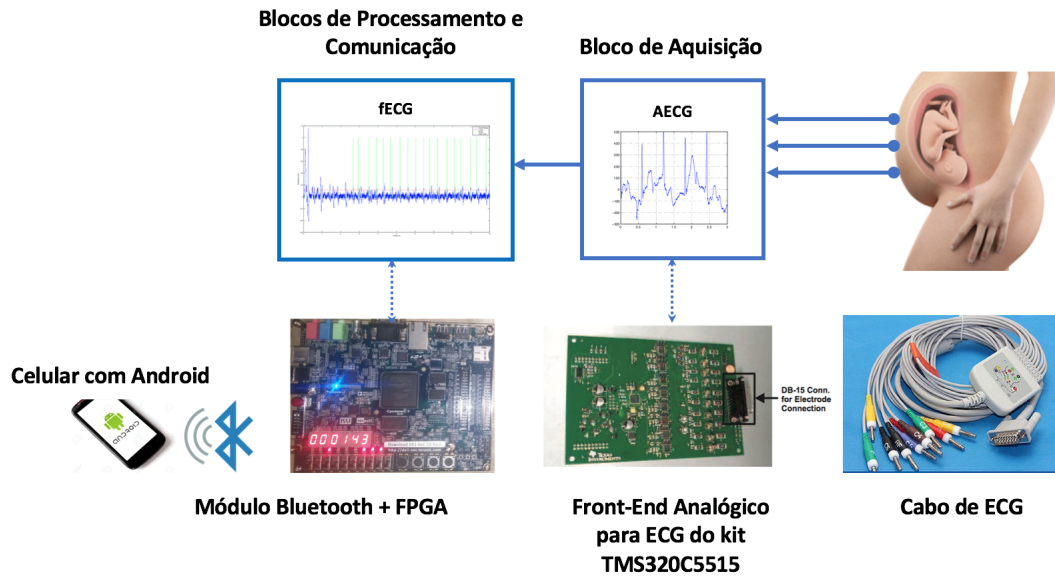


Figura 2 – Protótipo de sistema para estimativa da FHR através da aquisição de sinais aECG. Em desenvolvimento pela Universidade de Brasília - FGA.

1.2 Definição do Problema

Na arquitetura de rede neural convolucional proposta por (JUNIOR, 2018), foram utilizados sinais simulados de aECG, gerados em MatLab, onde a partir destes sinais a rede foi treinada e testada.

Entretanto, a arquitetura foi descrita apenas na linguagem C e executada em software, não sendo testada em um microprocessador instanciado em hardware devido a limitações de memória do dispositivo utilizado, onde houve a tentativa do uso do processador *softcore* MIPSfpga no chip FPGA do kit de desenvolvimento Basys3. Além disso, não houve a descrição de nenhuma das camadas da arquitetura da rede neural em VHDL e sua respectiva implementação em hardware em FPGA.

Portanto, o problema a ser resolvido no presente projeto é a aceleração em FPGA das camadas com alto custo computacional da arquitetura da Rede Neural Convolucional proposta.

1.3 Objetivos

1.3.1 Objetivos Gerais

Implementar em FPGA, a aceleração do algoritmo da Rede Neural Convolucional proposta por (JUNIOR, 2018), para a primeira camada convolucional da arquitetura da rede, haja vista que as camadas convolucionais da rede requerem alto custo computacional, demandando maior tempo de execução quando realizado via software.

1.3.2 Objetivos Específicos

Para obter sucesso no objetivo geral do projeto, os seguintes objetivos específicos são necessários:

- Implementação do código da arquitetura da Rede Neural Convolutacional no ARM;
- Descrição em VHDL do código para a primeira camada convolutacional da rede e realizando testes comportamentais;
- Implementação em hardware da primeira camada convolutacional com teste usando o ILA core;

1.4 Estrutura do Trabalho

O presente trabalho está composto de 6 capítulos. No Capítulo 1, tem-se a Introdução, onde são apresentados a contextualização e objetivos do trabalho. No Capítulo 2, é abordada a fundamentação teórica, explicitando conceitos bases para a compreensão do projeto, como o desenvolvimento cardíaco de um feto, o que são Redes Neurais Artificiais, o que é FPGA, SoC e sua aplicação em aceleração de algoritmos e, por fim, uma apresentação de trabalhos correlatos. No Capítulo 3, é mostrada a metodologia do projeto, como também as ferramentas utilizadas para o desenvolvimento do mesmo. O Capítulo 4 apresenta a implementação da primeira camada convolutacional que foi proposta na arquitetura da Rede Neural, por meio de esquemáticos e diagramas que mostram como foi descrito o bloco de circuito. No Capítulo 5 são apresentados os resultados obtidos e discussões a respeito da implementação em FPGA da primeira camada convolutacional descrita em VHDL, trazendo a comparação de desempenho entre hardware e software, dentre outros parâmetros de análise. Por fim, no Capítulo 6 tem-se a conclusão do projeto, onde é comentado os sucessos e fracassos para os objetivos propostos e sugestões para trabalhos futuros.

2 Fundamentação Teórica

2.1 ECG Fetal

2.1.1 Formação e Atividade Cardíaca

O coração está entre os primeiros órgãos desenvolvidos na formação do feto, onde nos primeiros estágios da gravidez, sofre um crescimento considerável. O período mais crítico desse desenvolvimento é entre a 3^a e 7^a semana, quando um tubo cardíaco simples assume a forma de um coração de quatro câmaras (SAMENI; CLIFFORD, 2011).

A partir da 3^a semana de vida, o coração já começa a bater e realizar a circulação de seu próprio sangue, em um sistema circulatório fechado e separado do da mãe. A figura 3 apresenta os estágios de desenvolvimento do coração fetal durante a gestação em ordem cronológica.

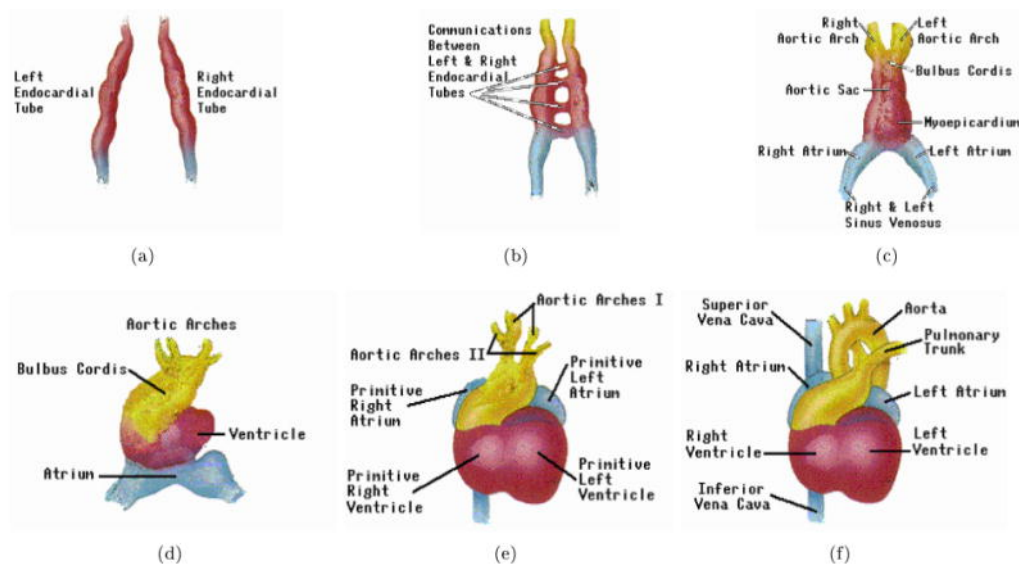


Figura 3 – Estágios de desenvolvimento do coração fetal de (a) a (f): a) tubos endocárdicos. b) comunicações entre tubos endocárdicos. c) arcos aórticos, sacos aórticos, mioepicárdio, átrios e seios venosos d) arcos aórticos, bulbus cordis, ventrículo e átrio e) arcos aórticos I, arcos aórticos II, átrios primitivos, ventrículos primitivos f) veia cava superior, aorta, tronco pulmonar, átrios, ventrículos, veia cava inferior. FONTE: (SAMENI; CLIFFORD, 2011)

Por volta da 7^a a 9^a semana, o coração fetal pode ser monitorado externamente por meio de ultrassom. Na 20^a semana, é possível escutar o batimento cardíaco fetal sem amplificação, com uma taxa entre 120-160 BPM (Batimento Por Minuto), aproximadamente. Todavia, poucas informações adicionais de diagnóstico podem ser determinadas a partir da escuta da frequência cardíaca neste período. Surge então, a necessidade do

uso de equipamentos como o Eletrocardiógrafo para análises do fECG (Eletrocardiograma Fetal) e do mECG (Eletrocardiograma Materno), que contêm informações morfológicas da atividade cardíaca que são interessantes para análises clínicas (SAMENI; CLIFFORD, 2011).

Embora a função mecânica do coração fetal seja diferente da de um coração adulto, sua atividade elétrica pulsante é bastante semelhante. A ação de bombeamento em forma de onda do coração é realizada no miocárdio a partir de contrações e relaxamentos regulares. A estimulação miocárdica inicia no nó sinoatrial (SA), que funciona como o marca-passo natural do coração, enviando o impulso elétrico que dispara cada batimento cardíaco. Esse impulso estimula ainda o nó atrioventricular (AV) e posteriormente ocorre a despolarização dos músculos ventriculares. A etapa de contração do miocárdio é chamada de ciclo de despolarização que é seguido pelo ciclo de repolarização, no qual o miocárdio relaxa e fica pronto para a próxima ativação. (SAMENI; CLIFFORD, 2011).

O ECG medido na superfície corporal é o resultado da ativação em estágio do miocárdio e resulta no complexo PQRST, ciclo cardíaco completo representado na Figura 4. Nesta notação, a onda P é responsável pela propagação da frente de despolarização através dos átrios. Em seguida, os ventrículos são despolarizados resultando no complexo QRS. Ao mesmo tempo, os átrios são repolarizados; no entanto, essa repolarização é obscurecida pela despolarização dos ventrículos. Por fim, temos a onda T, que corresponde à repolarização dos ventrículos.

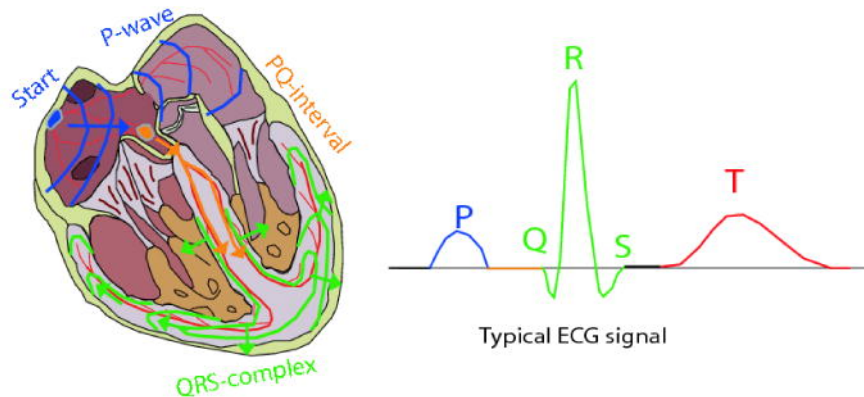


Figura 4 – O ciclo de ativação do coração fetal. FONTE: (SAMENI; CLIFFORD, 2011)

2.1.2 Estimativa da FHR

Morfologicamente, adultos e fetos apresentam padrões de ECG bastante semelhantes; mas as amplitudes relativas dos complexos fetais sofrem mudanças consideráveis ao longo da gestação e mesmo após o nascimento. A mudança mais considerável diz respeito às ondas T, que são bastante fracas para fetos e recém-nascidos. A partir da 20ª semana, a FHR (*Fetal Heart Rate* - Frequência Cardíaca Fetal) tem um valor basal de cerca de

120 a 160 batimentos por minuto (BPM). No entanto, embora as variações da frequência cardíaca fetal sejam diferentes para crianças e adultos, a FHR é conhecida por possuir variações circadianas que evoluem ao longo da gravidez. Estruturalmente, a variabilidade da frequência cardíaca (VFC) do feto também é conhecida por ser mais simples (menos variante) do que a de um adulto (SAMENI; CLIFFORD, 2011).

A medida da FHR é de extrema importância para o prognóstico gestacional. É necessário observar sempre se a mesma está dentro dos limites de normalidade conforme a idade gestacional, visto que contribui para a suspeita de anomalia cromossômica fetal, predição do risco para anomalias cromossômicas, diagnóstico para presença de infecção concomitante, prevenção de perda gestacional, etc (SILVA; BAILÃO, 2000).

O método mais promissor para monitoramento fetal é a Eletrocardiografia fetal não invasiva (NI-fECG). O principal motivo para essa afirmação é que o sinal de fECG não podem ser obtidos a partir de um monitoramento de cardiotocografia (CTG), devido à natureza de suas medições. Além disso, no método NI-fECG não há a exposição a nenhum tipo de radiação para a mãe e o feto. As contrações uterinas podem ser monitoradas pela detecção da atividade elétrica no abdômen materno (KAHANKOVA et al., 2020).

Para os algoritmos de extração do fECG, há duas categorias em relação a medição dos sinais: algoritmos que requerem apenas eletrodos abdominais, chamados métodos com fonte de eletrodo abdominal (*Abdominal Electrode-Sourced* - AES), e algoritmos que requerem eletrodos abdominais e torácicos, chamados métodos de fonte combinada (*Combined Source* - CS). Os princípios dessa categorização são ilustrados na Figura 5.

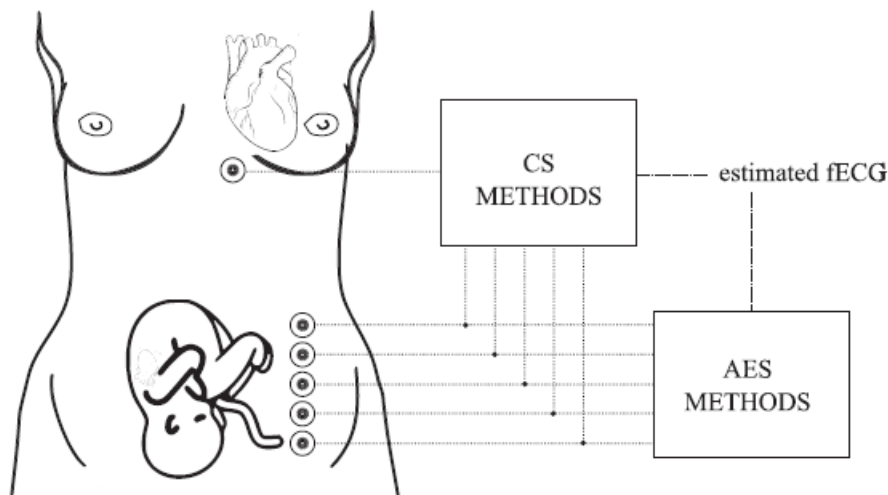


Figura 5 – Ilustração dos sistemas de extração AES e CS. FONTE: (KAHANKOVA et al., 2020)

Dentre as técnicas de processamento para o AES podemos destacar os filtros de Kalman e as transformadas Wavelet. Por outro lado, no método CS, tem-se os Filtros de Mínimos Quadrados Médios e Mínimos Quadrados Recursivos, assim como a aplicação

de RNAs. Entre várias possibilidades, o método mais eficiente para extração de fECG consiste na combinação de diferentes técnicas e na criação de sistemas híbridos, que se mostram mais promissores na finalidade de alcançar uma estimativa precisa da frequência cardíaca fetal (KAHANKOVA et al., 2020).

2.2 Redes Neurais Artificiais

2.2.1 Definição

As Redes Neurais Artificiais (RNAs) são máquinas computacionais de estrutura conexionista que têm como objetivo aproximar, generalizar, adaptar-se e gerar informações baseadas na experiência para execução de uma tarefa ou função. Seu sistema de processamento não-algorítmico é distribuído por um grande número de pequenas unidades interligadas que em algum nível lembram a estrutura do cérebro (FILHO, 2018).

De acordo com (HAYKIN, 2001):

Uma rede neural é um processador maciçamente paralelamente distribuído constituído de unidades de processamento simples, que têm a propensão natural para armazenar conhecimento experimental e torná-lo disponível para o uso. Ela se assemelha ao cérebro em dois aspectos:

1. O conhecimento é adquirido pela rede a partir de seu ambiente através de um processo de aprendizagem.
2. Forças de conexão entre neurônios, conhecidas como pesos sinápticos, são utilizadas para armazenar o conhecimento adquirido.

Os neurônios biológicos, por sua vez, são células presentes no sistema nervoso que possuem como principal função conduzir os impulsos nervosos e constituem-se de partes básicas. São elas: o corpo celular, os dendritos e o axônio (BEAR, 2008), formando a seguinte estrutura apresentada na figura 6. De forma semelhante, o neurônio artificial tenta aproximar-se do modelo biológico em estrutura e funcionalidade.

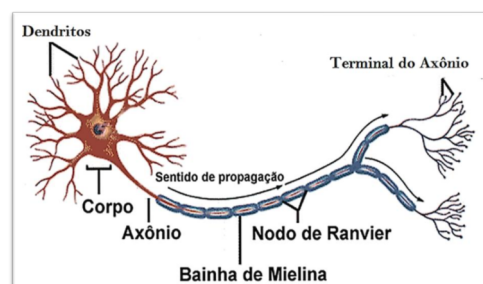


Figura 6 – Composição de um neurônio biológico. FONTE: (BRITO, 2019)

De acordo com (BRAGA, 2000) dentre os diferentes componentes que um modelo básico de RNA possui, temos:

- Conjunto de sinapses: conexões entre os neurônios da RNA, por onde entram os sinais. Para cada conexão é atribuído um peso sináptico;
- Junção aditiva: realiza as somas dos sinais de entrada da RNA, ponderados pelos pesos sinápticos;
- Bias: valor externo acrescido a cada neurônio com o objetivo de aumentar ou diminuir a entrada líquida da função de ativação.
- Função de ativação: função que limita a amplitude do valor do sinal de saída de um neurônio;

Na figura 7, tem-se a estrutura de um neurônio artificial de modelo não linear. Os sinais de entradas x_k são multiplicados pelos pesos sinápticos w_{kj} e somados entre si na função aditiva u_k , acrescidos por um bias b_k . Por fim, o resultado do somatório passa pela função de ativação φ , resultando no sinal de saída y_k .

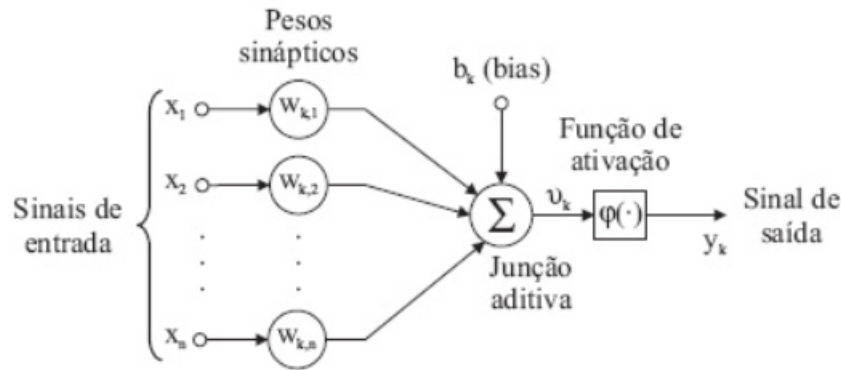


Figura 7 – Modelo não linear de um neurônio. FONTE: (HAYKIN, 2001)

As equações 2.1 e 2.2 fornecem o modelo matemático para o neurônio artificial:

$$u_k = \sum_{j=1}^m w_{kj} x_j \quad (2.1)$$

$$y_k = \varphi(u_k + b_k) \quad (2.2)$$

2.2.2 Redes Neurais Convolucionais

A Rede Neural Convolucional (ou *Convolutional Neural Network* - CNN) é uma arquitetura derivada do modelo de rede Perceptron de Múltiplas Camadas, sendo ampla-

mente aplicada no campo de visão computacional para dados visuais. A concepção da CNN consiste em múltiplas camadas com diferentes funcionalidades, onde os neurônios das camadas são responsáveis pela aplicação de filtros, cada um (VARGAS; CARVALHO; VASCONCELOS, 2016).

As RNAs são arquiteturas treináveis compostas de múltiplos estágios. A entrada e a saída de cada estágio são conjuntos de matrizes chamados *feature maps* (mapas de características). Na saída, cada *feature map* representa um recurso específico extraído em todos os locais da entrada. Cada estágio é composto de três camadas: uma camada de banco de filtros, onde é aplicada a convolução da entrada e é feito o acréscimo do bias treinado, uma camada de não linearidade e uma camada de *pooling* onde ocorre um agrupamento das características. Uma CNN típica é composta por um, dois ou três desses estágios de 3 camadas, seguidos por um módulo de classificação (LECUN; KAVUKCUOGLU; FARABET, 2010).

A figura 8 mostra um modelo de CNN e a disposição das camadas, que são: camada de convolução, camada de não linearidade, camada de *pooling*, camada *flatten*, camada densa (*fully-connected*) e camada de saída.

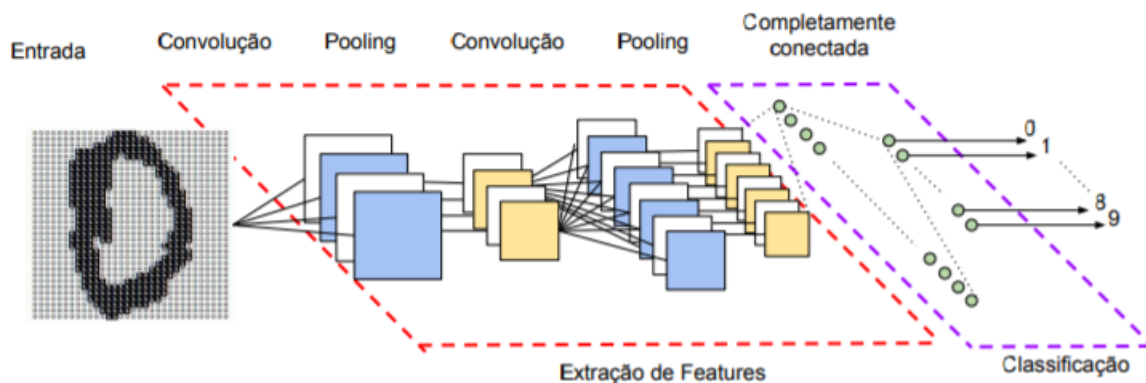


Figura 8 – Modelo de uma rede neural convolucional e suas diferentes camadas. FONTE: (VARGAS; CARVALHO; VASCONCELOS, 2016)

- Camada de convolução - Segundo (KHAN et al., 2018) esta é a camada mais importante da CNN, pois nela ocorre a convolução da entrada com um conjunto de filtros chamados *kernels*, gerando o mapa de características.
- Camada de não linearidade - Considerada por alguns autores como a função de ativação do neurônio da camada convolucional, sua função é amenizar os efeitos da linearidade em redes neurais, onde a maioria dos problemas são não lineares. Atualmente utiliza-se a função ReLU (*Rectified Linear Unit*) para a maioria das aplicações.

- Camada de pooling - nesta camada temos um processo simples de *downsampling*, onde a dimensionalidade/*feature maps* é reduzida. A variância da informação a pequenas alterações é reduzida, assim como a quantidade de parâmetros treinados pela rede.
- Camada flatten - É responsável por operar uma transformação na matriz da fornecida pela camada anterior, alterando seu formato para um vetor. Basicamente, os dados são alinhados e concatenados de forma que a camada densa possa lê-los corretamente.
- Camada densa - Trata-se de uma RNA simples, que recebe os dados tratados pelas camadas anteriores e, a partir deles, generaliza e realiza a classificação dos dados para a camada de saída.

2.2.3 Treinamento por *Backpropagation*

O algoritmo de retropropagação de erro, referido na literatura como *Backpropagation*, é um processo de treinamento supervisionado para RNAs baseado na regra de aprendizagem por correção de erro. Consiste de dois passos realizados dentro das camadas da rede: a propagação e a retropropagação. Na propagação, vetores de entrada são aplicados aos nós da rede, propagando o seu efeito, camada por camada, produzindo um conjunto de saídas como a resposta real da rede. Durante este passo, os pesos sinápticos são todos estáticos. Já na retropropagação, todos os pesos sinápticos são ajustados conforme uma regra de correção de erro e a resposta real da rede é subtraída do valor esperado, produzindo um sinal de erro que é propagado para trás pela rede, contrário aos pesos sinápticos. Por fim, o procedimento é realizado repetidamente, sempre ajustando os pesos sinápticos para fazer com que a resposta real da rede aproxime o máximo do valor esperado, estatisticamente (HAYKIN, 2001).

A Figura 9, apresenta um diagrama esquemático básico do algoritmo de treinamento de retropropagação, onde ao obter a saída da rede neural, calcula-se o erro pela diferença entre o resultado de saída ("Obs") e o resultado esperado ("Model"), avaliando se este erro é tolerável ou não. Caso o erro não esteja satisfazendo a tolerância, ocorre a retropropagação do erro no sentido contrário nas camadas da rede neural, onde são calculados os valores dos gradientes para cada peso da rede tomando o sentido contrário dele, na tentativa de minimizar o valor da função de erro. Uma vez que tem-se o vetor gradiente calculado, os pesos são atualizados no sentido de modo iterativo, sempre recalculando os gradientes em cada passo de iteração, até que o erro diminua e alcance um valor menor que a tolerância estabelecida, finalizando assim o algoritmo de treinamento.

O desenvolvimento do algoritmo de retropropagação representou um marco nos conceitos de aprendizagem das redes neurais, que embora não forneça uma solução perfeita

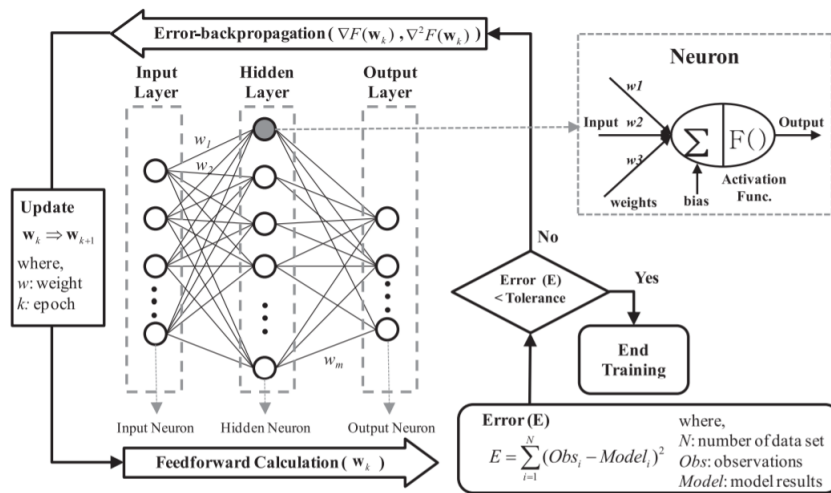


Figura 9 – Diagrama esquemático do algoritmo de treinamento de retropropagação e modelo de neurônio típico. FONTE: (KIM; SEO, 2015)

para os problemas solúveis, fornece um método computacional eficiente para o treinamento das RNAs, em especial as redes perceptrons de múltiplas camadas (HAYKIN, 2001).

2.3 FPGAs

FPGA (*Field-Programmable Gate Array*) é um dispositivo lógico programável que possui uma arquitetura baseada em blocos lógicos configuráveis alocados em forma de uma matriz. Em geral, a funcionalidade e o roteamento destes blocos são configuráveis, e as vezes reconfiguráveis, via software. Os FPGAs proporcionam um ambiente de trabalho simplificado e de baixo custo, possibilitando operar com um número ilimitado de circuitos por meio da configuração do próprio dispositivo (ORDONEZ, 2003).

A arquitetura básica de um FPGA é mostrada na Figura 10. Esta pode variar de acordo com a fabricante do dispositivo, com as categorias ou até em uma mesma categoria podem existir variações, que dependem da sua área de aplicação. Todavia, podemos destacar os principais elementos que compõe um FPGA:

- CLB (*Configurable Logic Block*): são as unidades lógicas de uma FPGA, formados por *flip-flops* e *look-up tables* (LUTs).
- IOB (*In/Out Block*): bloco dos periféricos de entrada e saída do FPGA, localizado na periferia dos FPGAs, sendo responsáveis pela interface com o ambiente.
- SB (*Switch Box*): são as interconexões programáveis entre os CLBs, através dos canais de roteamento.

- BRAM (*Block Random Access Memory*): é o bloco de memória intrínseco ao chip, não presente em todos FPGAs. Dentro da limitação do tamanho do BRAM, as células de memória interna podem ser subdivididas e cascadeadas para formarem blocos conforme necessidade de armazenamento configurado no projeto.
- DSP (*Digital Signal Process*): são unidades especializados em processamento digital de sinal, também não presente em todos FPGAs. Estes blocos são dedicados para processar funções como filtragem, multiplicação, conversão analógico-digital e vice versa, etc. São consideravelmente mais eficientes quando comparados aos CLBs.

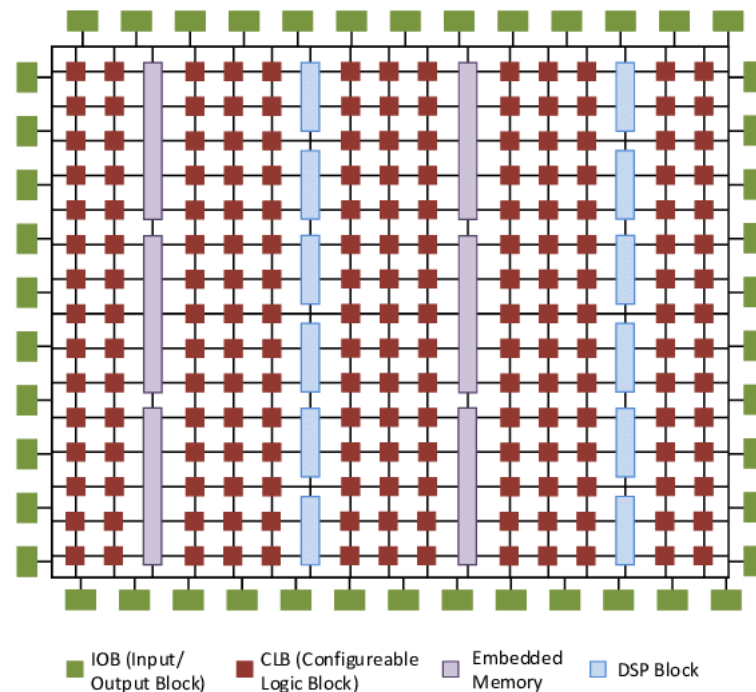


Figura 10 – Arquitetura interna de um chip FPGA típico. FONTE: (BAJAJ; FAHMY, 2015)

A chamada *Hardware Description Language* (HDL), é a linguagem utilizada programar os circuitos lógicos eletrônicos dentro dos FPGAs, que descreve as estruturas, operação e projeto dos circuitos nos dispositivos. A linguagem HDL possui uma sintaxe composta de operadores, expressões, estruturas e operações de entrada e saída. Ao contrário das linguagens de programação de software, que geram um assembly executável no computador, a HDL provê um mapa de portas, algumas vezes mencionados como *bitstream*. O *bitstream* obtido é carregado no dispositivo FPGA com objetivo de verificar sua operação. A linguagem permite descrever qualquer circuito digital de forma estrutural e comportamental (BERBERT; BERTINI; COPETTI, 2021).

2.3.1 Aceleração de Algoritmos

Devido às características de flexibilidade, confiabilidade e paralelismo fornecidas pelos FPGAs, destaca-se a possibilidade de utilizar esses recursos para executar tarefas específicas de alto desempenho em que é necessário o máximo de eficiência, tanto em poder computacional quanto em velocidade na execução, como por exemplo o processamento de dados em tempo real. Com o FPGA, é possível ajustar os recursos de hardware em tempo de execução de modo a atender as mudanças de requisitos de energia, desempenho e tolerância a falhas (BERBERT; BERTINI; COPETTI, 2021).

A principal vantagem dos sistemas com processadores FPGA é a flexibilidade. Todavia, quando a premissa de um projeto é o desempenho, os sistemas ASIC (*Application Specific Integrated Circuits*) são preferíveis em sacrifício à flexibilidade. Já quando o desempenho é necessário, mas não é possível deixar a flexibilidade completamente de lado, uma solução comum é a implementação dos sistemas híbridos, onde os processadores em ASIC são associados com partes em FPGA, combinando desempenho e eficiência e mantendo parte da flexibilidade (BERBERT; BERTINI; COPETTI, 2021).

Aceleração de hardware é a técnica do uso de uma arquitetura feita exclusivamente para realizar uma tarefa específica que executa as operações de forma mais eficiente do que se a mesma fosse feita utilizando um software para uma CPU (*Central Process Unit*) de uso geral. A aceleração de hardware implementada em FPGA tem sido bastante aplicada e com sucesso na aceleração de algoritmos como por exemplo na área de *Deep Learning*, em particular nas CNNs em aplicações de reconhecimento de imagens relacionados à Visão Computacional (BERBERT; BERTINI; COPETTI, 2021).

2.3.2 SoCs

O SoC (*System on a Chip*) é um circuito integrado no qual pode-se ter um sistema eletrônico ou de computador completo, com processador, memória, arquitetura de conexão, etc. Adicionando a lógica programável do FPGA, tem-se então a possibilidade de realizar modificações na estrutura de hardware. Sendo possível adicionar novos módulos ao SoC, aqueles podem acelerar e auxiliar os processadores embarcados em tratamentos que requerem alta capacidade de processamento. A comunicação do sistema de processamento com o sistema programável é feita através de um barramento interno, estabelecendo a integração do software com o hardware (CUNHA, 2014).

A utilização de SoCs acrescenta as seguintes vantagens em suas aplicações:

- redução do tempo de projeto;
- redução na quantidade de componentes eletrônicos, diminuindo o custo e aumentando a confiabilidade do sistema;

- aumento no desempenho, uma vez que todos os componentes estão dentro do mesmo circuito integrado;
- maior flexibilidade de adaptações e atualizações nos algoritmos descritos em hardware.

A figura 11 mostra um exemplo da arquitetura do Zynq@-7000 SoC, um dispositivo SoC da Xilinx que integra a programabilidade do software de um processador baseado em ARM(*Advanced RISC Machine*) com a programação do hardware de um FPGA, permitindo análises importantes e aceleração de hardware enquanto integra CPU, DSP, ASSP(*Application-specific standard parts*) e funcionalidade de sinal misto em um único dispositivo (XILINX, 2021). A região da Lógica Programável representa a parte do FPGA, enquanto o Sistema de Processamento pode ser um processador dedicado ou CPU, por exemplo.

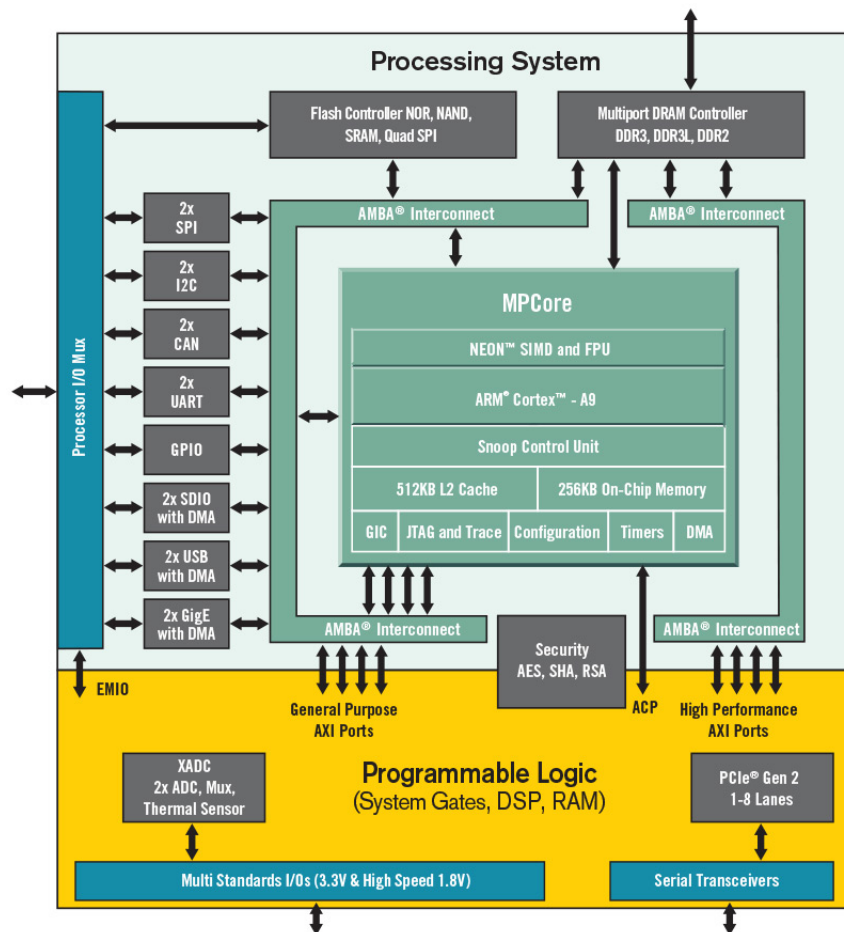


Figura 11 – Diagrama de Blocos estrutural do Zynq-7000 Soc. FONTE:(XILINX, 2021)

2.4 Estado da Arte

Após ampla pesquisa, destacou-se apenas um autor, ou grupo de autores, que difundiu trabalhos específicos para a extração de sinais fECG por meio de redes neurais implementadas em FPGA.

Em uma primeira abordagem, (HASAN; IBRAHIMY; REAZ, 2009) apresentou apenas a arquitetura e algoritmo da extração do fECG através de redes neurais com abordagem de filtros adaptativos que combinam ADALINE (*Adaptative Linear Netwok*) e TDL (*Tapped Delay Line*), onde concluiu que a técnica de filtragem de rede neural adaptativa trouxe resultados satisfatórios, mas que a precisão da saída depende de quantas variações de sinais são usadas como entrada e o alvo na rede neural.

Já em (HASAN et al., 2009), foi realizada a modelagem em VHDL (*VHSIC Hardware Description Language*) do algoritmo desenvolvido em (HASAN; IBRAHIMY; REAZ, 2009) para implementações em FPGAs. O modelo proposto foi sintetizado e inserido no Stratix II EP2S15F484C3 da Altera usando o Quartus II versão 7.2 Web Edition. Os resultados obtidos nas simulações do Quartus II, para o modelo em VHDL, foram similares aos resultados obtidos pelas simulações no MatLab, porém não foi implementado na FPGA.

Em continuidade, em (HASAN; REAZ, 2012), foram abordadas a implementação e a prototipagem do algoritmo em hardware no FPGA, onde foi afirmado que o algoritmo desenvolvido é capaz de extrair completamente o fECG do aECG e que o desempenho obtido para detecção de pico-R no monitoramento da FHR foi de 93,75%, onde relatórios de síntese indicaram que a utilização lógica do FPGA foi de 36%, com 1721 registradores e 56 DSPs usados no total.

Saindo do contexto de extração de fECG, (MAIA, 2020) propôs um sistema automático para geração de arquiteturas SoC para a execução de CNNs em FPGA, por meio da implementação de subsistemas co-design entre software e hardware. Neste projeto foi utilizado a técnica de análise de alto nível, gerando um IP de coprocessador para a camada de convolução da rede. Foram apresentados resultados em que a camada de convolução implementada em hardware obteve tempo de execução 15,2 vezes mais rápido do que a convolução implementada em software.

Um trabalho desenvolvido por (TOMASI, 2020) apresentou um acelerador de hardware em FPGA para aplicações em aprendizado de máquina, por meio da implementação de duas arquiteturas dedicadas para o cálculo da entrada da função de ativação de uma rede neural perceptron. Os resultados mostraram um bom compromisso de design onde a primeira arquitetura obteve tempo de execução de 270ns, enquanto a segunda pode ser computada em 330ns, sendo esta cerca de 4 vezes mais rápida e 50 vezes mais eficiente que um script em Python.

O presente trabalho é oriundo do trabalho desenvolvido por (JUNIOR, 2018), onde foi proposta a arquitetura de Rede Neural Convolucional para estimativa da FHR baseado em FPGA, que em resultados de simulação por software a rede apresentou 96,78% como o menor percentual de acerto registrado e 98,36% como o maior, com um desvio padrão de 0,67% e valor médio igual a 97,43%.

3 Aspectos Metodológicos

3.1 Metodologia

No que tange à metodologia do projeto, foi utilizada uma abordagem qualitativa para a implementação da CNN no FPGA, com finalidade de analisar os resultados obtidos, desafios encontrados e proposição de soluções e melhorias, para obter então o melhor sistema para se adequar ao projeto de estimativa da FHR pelo fECG.

A fim de compreender o contexto do problema evidenciado, foi realizada uma pesquisa teórica para compreensão da formação cardíaca do feto, início do FHR e seus valores normais, métodos para estimativa do FHR e a sua importância, interpretação do ECG e meios de aquisição dessa informação. Também foi realizada pesquisa teórica a respeito das redes neurais artificiais, abordando os tipos de arquiteturas, as aplicações de cada uma e os principais métodos de implementação, em especial a Rede Neural Convolutiva, onde foram destacadas as vantagens, aplicação em visão computacional e o detalhamento funcional, camada por camada.

Com base no entendimento do trabalho desenvolvido por ([JUNIOR, 2018](#)), foi estudada a arquitetura proposta da CNN e a sua implementação em linguagem C. Foi reproduzida a geração do sinal sintético de aECG no MatLab para testes na CNN proposta. Então, foram pesquisados os métodos de aceleração de algoritmos e as vantagens de utilização de FPGAs e sistemas SoC.

Foi descrita em VHDL a primeira camada convolutiva da CNN proposta, uma vez que as demais camadas convolucionais podem se derivar desta, com alterações nas dimensões dos dados e das iterações. A camada convolutiva é a camada que mais exige poder computacional, sendo a mais indicada para aproveitar o paralelismo fornecidos pelos FPGAs.

Por fim, foram realizadas simulações comportamentais e validação da implementação do circuito da primeira camada convolutiva da rede em hardware, usando o ILA core na ferramenta Vivado, a fim de validar o seu correto funcionamento, comparando os resultados de desempenho do circuito em hardware com os resultados do modelo da camada executadas em software no ARM.

3.2 Ferramentas

Para o desenvolvimento do projeto, foram utilizadas as ferramentas descritas a seguir.

3.2.1 MatLab

MatLab (*MATrix LABoratory*) é um software interativo de alta performance aplicado para cálculo numérico, onde o elemento básico do cálculo é uma matriz que não exige dimensionamento. O software dispõe um ambiente de área de trabalho ajustado para análise iterativa e processos de projeto com uma linguagem de programação que expressa a matemática de matrizes e matrizes diretamente como é escrita, integrando análises numéricas, cálculos com matrizes, processamentos de sinais e construção de ferramentas gráficas ([MATHWORKS, 2005](#)).

O MatLab foi utilizado no presente projeto para a geração do sinal sintético de aECG (mECG + fECG) acrescido de ruídos aleatórios gerados para simular possíveis camadas de gordura, contrações uterinas, etc., buscando tornar o sinal mais próximo do real. Este sinal gerado foi usado para testes da rede neural e também para obtenção das amostras de entrada para a camada da rede neural em software e em hardware.

A ferramenta também foi utilizada para gerar os vetores de testes comportamentais, convertendo os valores de entrada e saída da camada de binário para ponto flutuante e vice-versa, além de ser utilizado para cálculo do erro quadrático médio entre os resultados do software e do hardware.

3.2.2 Vivado

O *Vivado Design Suite* é um conjunto de software produzido pela *Xilinx* para síntese e análise de projetos HDL. Substituto do Xilinx ISE, o Vivado possui recursos adicionais para o desenvolvimento de projetos de SoC e síntese de alto nível. O software também simplifica a tarefa de integração de blocos de propriedade intelectual, chamados IP (*Intellectual Property*) para montar, verificar e implementar e projetos de sistemas que incluem portfólios destes IPs, seja da Xilinx, de terceiros ou próprio ([FEIST, 2012](#)).

A integração destes IPs pelo Vivado dentro de um projeto segue o *flow* mostrado na Figura 12, onde tem-se que o IP *Package* do Vivado permite que a Xilinx, designers e provedores de IP terceirizados empacotem um projeto com suas restrições, bancos de teste e documentação para torná-lo disponível em um catálogo IP extensível em uma unidade local ou compartilhada. Uma vez catalogado, a integração deste amplo portfólio de IP em projetos verificados no nível do sistema pode ser realizada abstratamente no nível da interface usando o integrador Vivado IP com verificação de regras de projeto.

O software Vivado foi utilizado no projeto para a descrição dos códigos em VHDL, instanciação de IPs, testes comportamentais, síntese e implementação do circuito, além de testes em hardware utilizando o ILA core.

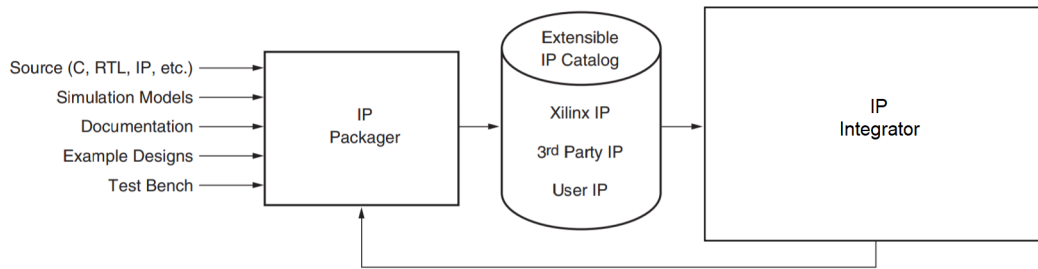


Figura 12 – Vivado Design Suite IP Flow. FONTE: Adaptado de (FEIST, 2012)

3.2.3 ILA Core

O ILA IP core (*Integrated Logic Analyzer*), é um analisador lógico customizável que pode ser usado para monitorar os sinais internos de qualquer aplicação que exija verificação ou depuração usando o analisador lógico Vivado. Seu núcleo inclui diversos recursos avançados de análise lógica moderna, incluindo como *triggers* as equações booleanas e transições de borda. O monitoramento do ILA é sincronizado com o circuito a ser monitorado, onde todas as restrições de *clock* aplicadas ao projeto também são aplicadas aos componentes do ILA. A figura 13 mostra a simbologia do componente do IP ILA core, mostrando entradas de *clock*, os *triggers* e pontas de provas de entrada e saída.

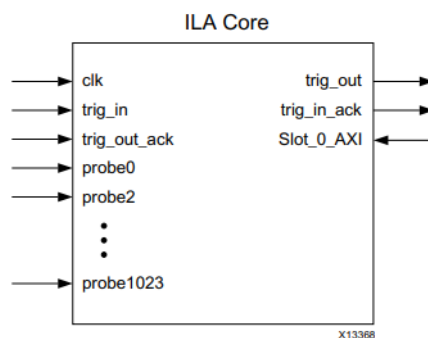


Figura 13 – Símbolo do ILA Core. FONTE:(XILINX, 2016)

O ILA Core foi instanciado e utilizado no projeto para validação do correto funcionamento da primeira camada convolucional descrita em hardware no FPGA.

3.2.4 XSDK

O XSDK (*Xilinx Software Development Kit*) é o IDE (*Integrated Development Environment*) da Xilinx que permite a aplicação de projetos multiprocessador inteiramente homogêneos e heterogêneos, com depuração e análise de desempenho. Possui um ambiente de projeto integrado para a criação de aplicativos incorporados em qualquer um dos microprocessadores da Xilinx: Zynq® UltraScale + MPSoC, Zynq-7000 SoCs e o micro-

processador soft-core MicroBlaze. o XSDK é baseado no padrão de código aberto Eclipse ([XILINX, 2019](#)).

O software XSDK foi proposto para o desenvolvimento do processador embarcado, onde ocorreria a comunicação do bloco da camada convolucional, descrito em hardware, com o algoritmo da rede neural, descrito em software. A Figura 14 exemplifica um diagrama de comunicação com a utilização do xSDK.

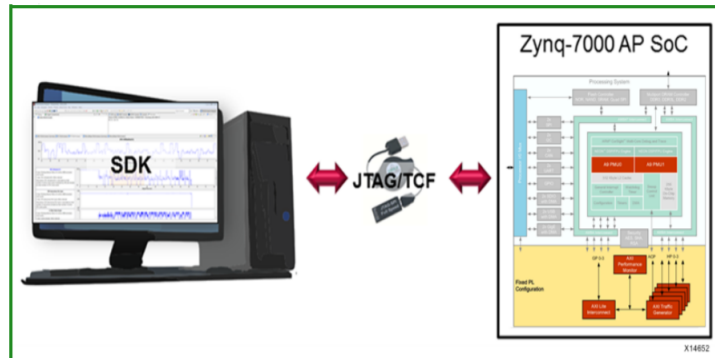


Figura 14 – comunicação Host-Target para o xSDK. FONTE:([XILINX, 2019](#))

O XSDK foi utilizado para executar a análise de *profile* da rede neural em C no ARM instanciado em IP core e implementado no SoC Zynq-7000 da placa de desenvolvimento Zybo do laboratório remoto da FGA/UnB.

3.2.5 Basys 3 Artix-7

Para implementação do projeto no FPGA, foi utilizado o kit Basys 3 (vide figura 15), que é uma plataforma completa de desenvolvimento de circuito digital e pronta para uso baseada no mais recente Artix-7 FPGA da Xilinx. O seu FPGA, o Artix-7, é de baixo custo, possui alta capacidade e uma série de portas USB, VGA e outras. O Basys 3 permite implementar projetos de circuitos combinacionais simples e também circuitos sequenciais complexos que usam processadores e controladores incorporados, além de possuir *switches*, LEDs e pinos I/O, o que faz esse kit ser amplamente utilizado no aprendizado e introdução do mundo dos FPGAs ([DIGILENT, 2016](#)).

O FPGA Artix-7 é otimizado para lógica de alto desempenho e oferece alta capacidade, desempenho e uma gama de recursos para o projetista. Os recursos do Artix-7 incluem:

- 33.280 células lógicas em 5200 blocos (cada bloco contém quatro LUTs de 6 entradas e 8 flip-flops);
- 1.800 Kbits de BRAM;

- 5 blocos de gerenciamento de *clock*, cada um com um PLL (*Phase Lock Loop* - bloqueio de fase em loop);
- 90 blocos DSP;
- Clock interno com velocidades de até 450 MHz;
- Conversor analógico-digital on-chip (xADC);

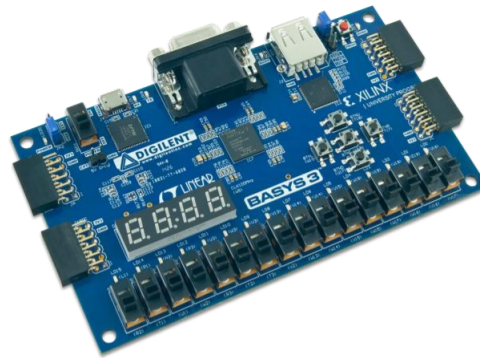


Figura 15 – Basys 3 Artix-7 FPGA. FONTE:([DIGILENT](#), 2016)

4 Implementação da Arquitetura Proposta

4.1 Sinal aECG gerado no MatLab

As ondas utilizadas nos testes da arquitetura foram geradas no software MatLab, por meio da função *ecg()*. Esta função recebe o parâmetro L , que indica o comprimento do ciclo do sinal gerado. Após a função *ecg()*, foi inserida a função *ecg(sgolayfilt())* para suavizar a onda gerada. Por fim, o ciclo de ECG foi repetido algumas vezes para obter um sinal periódico. Para tal, foi definido que a frequência de amostragem desejada é 1 KHz e o tempo de amostragem é de 10 segundos, resultando em um sinal com 10000 pontos, conforme realizado em (JUNIOR, 2018).

Para gerar o sinal ECG torácico materno, foi definido $L=675$. Tendo $10000/675$ ciclos em 10 segundos, chega-se à frequência de 88.8 BPM. Analogamente, para obter o sinal fECG, foi utilizado $L=420$ para a função *ecg()*, resultando em um sinal com FHR de 142.8 BPM. A figura 16 mostra a comparação entre o sinal materno e fetal, onde é possível perceber como o sinal materno possui amplitude muito maior. Observa-se que ambos sinais ainda não possuem ruídos.

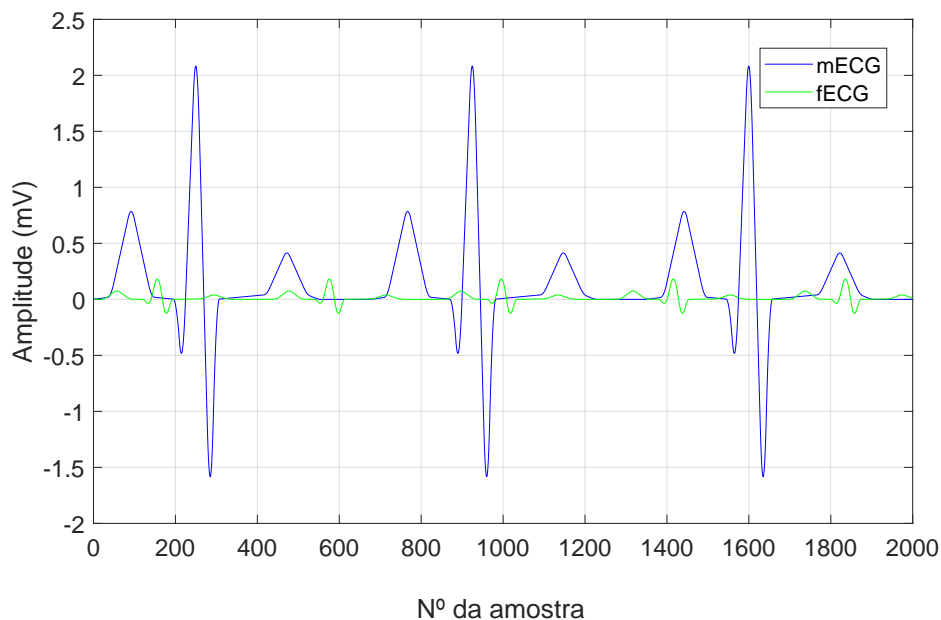


Figura 16 – Sinais mECG e fECG sem ruído. Fonte: Autor.

Por fim, para obter a fiel representação de um sinal aECG real, foram somados os sinais mECG e fECG e adicionados ruídos gaussianos brancos e ruídos com coeficientes aleatórios ao sinais, que tentam representar os mais diversos ruídos possíveis, como a

propagação da onda pelos tecidos do corpo, contrações uterinas, etc. O sinal aECG final está mostrado na Figura 17.

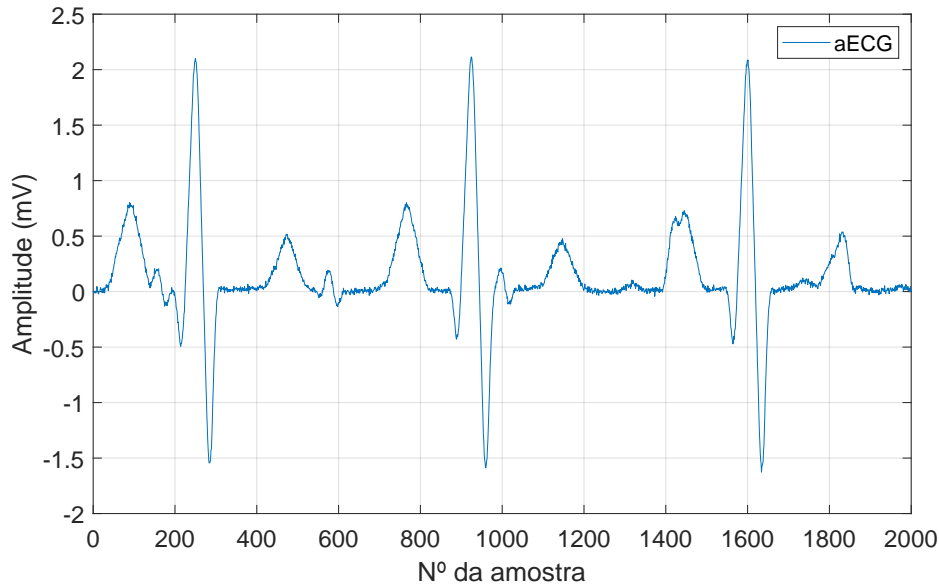


Figura 17 – Sinal aECG final gerado no MatILab. Fonte: Autor.

4.2 Descrição da arquitetura proposta

A arquitetura da CNN proposta por (JUNIOR, 2018) é composta de 9 camadas de processamento, cada uma com certa finalidade de extração de dados, redimensionamento ou classificação (vide Figura 18). O conjunto de entrada da CNN é um bloco de dados do sinal ECG com 60 amostras.

A primeira camada convolucional recebe o conjunto das 60 amostras e realiza um ajuste de tamanho, acrescentando 3 amostras nulas no início e ao fim do sinal, tornando o conjunto de entrada do sinal com 66 amostras. Esse ajuste na borda do sinal é necessário para que a convolução seja realizada de 7 em 7 amostras do conjunto de 66 amostras. Após o ajuste do sinal é iniciada a convolução da entrada com 32 filtros de tamanho 7x1, onde a saída da camada resulta em uma matriz de dados tamanho 60x32 que refletem a extração de características que sejam significativas na amostra.

A segunda camada, de *maxpooling*, realiza o agrupamento das amostras de 2 a 2 na matriz, retirando a amostra de menor valor. Dessa forma, tem-se o realce das características que foram extraídas na camada anterior. O dado resultante dessa camada é uma matriz de tamanho reduzido pela metade (para 30x32).

A terceira, quarta e quinta camada são convolucionais e têm como função realçar ainda mais as características mais relevantes do conjunto de amostras. O que altera de uma camada para outra são as dimensões dos filtros utilizados na convolução: na terceira

camada, são utilizados filtros 5x1; na quarta, são utilizados filtros 3x1, e na quinta, utiliza-se filtros unitários 1x1. Na saída dessas três camadas, a dimensão da matriz de dados não é alterada, mantendo-se em 30x32.

Na sexta camada, chamada *flatten*, ocorre o achatamento da matriz 30x32 resultante da camada anterior, onde os dados são reorganizados de forma que as linhas são transpostas e colocadas uma abaixo da outra, resultando em um vetor de 960 linhas e 1 coluna.

A partir da sétima camada, tem-se uma estrutura de rede neural artificial simples, com 2 camadas densas e 1 camada de saída. A primeira camada densa é composta de 128 neurônios, a segunda por 64 neurônios e a camada de saída composta de um neurônio, que realiza a classificação dos dados através da função de ativação sigmóide, classificando entre 1, para indicar pico de fECG, e 0, para indicar a não ocorrência de pico de fECG.

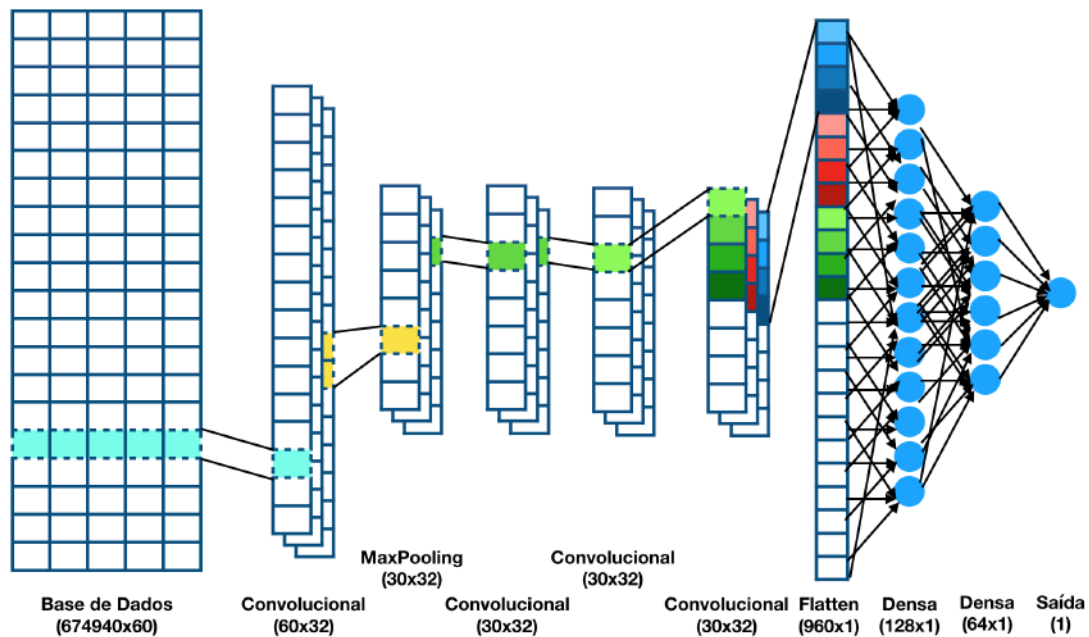


Figura 18 – Arquitetura da Rede Neural Convolutiva. FONTE: (JUNIOR, 2018)

4.3 Implementação da CNN em C no ARM

Para identificação dos gargalos computacionais do código em C da CNN proposta por (JUNIOR, 2018), optou-se pela execução do código em um processador ARM implementado em hardware que é mais apropriado, uma vez que em um co-projeto de hardware e software a parte de software comumente é executada por um microprocessador dedicado no SoC.

Sendo assim, foi instanciado um microprocessador ARM no SoC Zynq-7000 por meio da placa de desenvolvimento Zybo. Para instanciação foi utilizado a ferramenta de

Block Design do Vivado, onde foi projetado o bloco de processamento que inclui o IP do microprocessador Zynq e outros IPs adicionais como barramentos AXI, *timers*, blocos de memória e módulos de conexão GPIO. A figura 19 mostra o diagrama de blocos do microprocessador ARM instanciado.

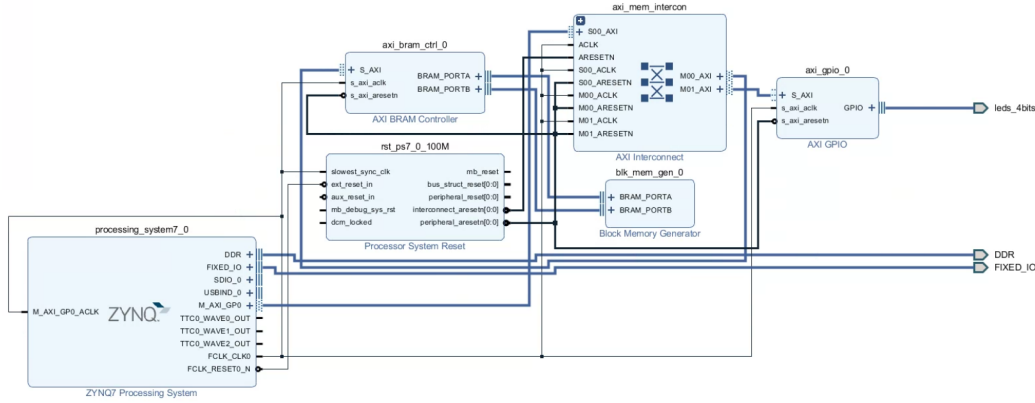


Figura 19 – Diagrama de blocos do microprocessador ARM-Zynq instanciado como IP.
FONTE: Autor.

A partir da validação do projeto do IP e implementação do *design* pelo Vivado, foi gerado o *Bitstream* do circuito e exportado como hardware para o software do XSDK.

No XSDK com o projeto criado para o microprocessador, foi incluído o arquivo do código em C da CNN e juntos com suas bibliotecas, os compiladores do SDK foram ajustado para reconhecimento de todas as funções utilizadas no código e também configurados conforme o tutorial de (AREIBI, 2020) para a implementação da análise de profile do código executado no microprocessador instanciado em hardware, que foi realizada pelo SDK e interpretada pelo *gprof*.

4.4 Descrição da Primeira Camada Convolutiva em VHDL

As camadas convolucionais são as mais importantes da arquitetura de uma Rede Neural Convolucional, conforme estabelecido no capítulo 2.2. Além disso, são as camadas que mais exigem poder computacional, devido à enorme quantidade de iterações de multiplicações e somas, além das funções de ativação aplicadas em cada amostra da saída. Dessa forma, a aceleração de algoritmos em hardware foi feita inicialmente para a primeira camada convolutiva da rede, pois esta apesar de não ser a camada de maior custo computacional da rede, é a que possui maior tamanho de filtros 7x1 e, uma vez que sua implementação seja bem sucedida, a replicação de sua estrutura para as outras camadas convolucionais se apresenta ser mais simples do que quando feito o contrário.

Os filtros, ou *kernel*, aplicados na primeira camada convolutiva possuem função de realçar as principais características a serem extraídas das amostras de entradas que

contribuem para a classificação do pico de fECG. A definição do tamanho e dos valores dos filtros foram obtidos de (JUNIOR, 2018), onde estes foram formados por pesos inicializados aleatoriamente e atualizados durante o processo de *backpropagation*.

O algoritmo dessa camada pode ser analisado pelo código de sua função na linguagem C (vide Figura 20). Observa-se que o algoritmo consiste em *loops* iterativos, que realizam a convolução através da multiplicação das amostras (ajustadas) pelos filtros convolucionais e da soma dos resultados a cada 7 amostras, juntamente com o acréscimo de um bias, e aplicando ao resultado em cada amostra de saída a função de ativação RELU.

A função de ativação RELU é uma função não linear descrita na equação 4.1, onde quando o resultado da saída da camada for negativo o valor é convertido para zero e quando for maior ou igual a zero o valor de saída é mantido, logo tem-se uma redução de erros na classificação da amostra de entrada.

$$f(x) = \max(0, x) \quad (4.1)$$

```
// SIPO (7 saidas paralelas)
void conv1()
{
    // j - number
    for(int j=0; j<numberOfFilters; j++)
    {
        for(int i=0; i<sampleLength; i++)
        {
            for(int k=0; k<firstFilterLength; k++)
            {
                firstConvOutput[j][i] += (sampleAdjusted[i+k]*firstConvFilter[j][k]);
            }
            firstConvOutput[j][i] += firstConvBias[j];
            firstConvOutput[j][i] = RELU(firstConvOutput[j][i]);
            printf("%.32f, ", firstConvOutput[j][i]);
        }
        // printf("\n");
    }
    printf("\n");
    // printf("Iconv: %f\n",firstConvOutput[1][21]);
}
```

Figura 20 – Código em C para a primeira camada convolutiva. [*numberOfFilters*=32; *sampleLength*=60; *firstFilterLength*=7.] FONTE: Adaptado de (JUNIOR, 2018)

Pelo código da Figura 20, pode ser observado que dentro do *loop* mais interno tem-se a multiplicação por cada um dos 7 filtros a cada 7 amostras do ECG. Após cada multiplicação, o resultado é somado/acumulado e, após as 7 iterações, tem-se a saída para cada amostra da matriz resultante, onde é efetuada a soma com o bias e passa-se pela função de ativação. No próximo loop, as colunas são percorridas dentro de uma mesma linha da matriz, e no loop mais externo são percorridas as colunas.

Para implementar em VHDL a função completa da camada convolutiva, foi proposta uma arquitetura de projeto RTL (*Register Transfer Level*). A estratégia utilizada foi primeiramente, descrever o *loop* mais interno da função, que realiza a multiplicação e soma das entradas com os filtros, bias e função de ativação e posteriormente, instanciá-lo

como componente no módulo principal, e realizar as iterações dos *loops* externos, que percorre e varia as linhas e colunas das matrizes das amostras de entradas, filtros, bias e também na matriz de saída resultante da camada convolucional.

Vale ressaltar que o tamanho do conjunto de entradas, do bias, dos filtros e dos contadores que varrem os conjuntos de entrada e saída da camada convolucional, foram descritas de forma parametrizável, porém a quantidade de multiplicadores e somadores instanciados foi descrita de forma estática. Todos as constantes e parametrizações das entradas podem ser consultados no código VHDL da biblioteca *fpupack* do Apêndice A, tal biblioteca foi obtida de (MUNOZ D. F. SANCHEZ, 2010b), importada e adaptada no presente projeto, conforme necessidade.

4.4.1 Componente do *loop* interno

Para descrição em VHDL do loop mais interno do código referência em C, foi aproveitada a paralelização do FPGA para tornar mais eficiente o processo das sucessivas multiplicações e somas.

A arquitetura RTL demonstrada na Figura 21 implementa a função do loop mais interno do algoritmo em C proposto por (JUNIOR, 2018). A entrada do bloco é coletada a cada 7 amostras do sinal de ECG, sinal este que é um conjunto 60 amostras, em um registrador de entrada serial e saída paralela (SIPO), A quantidade de 7 amostras como entrada do bloco vem da definição da própria arquitetura da CNN proposta, onde os filtros da primeira camada convolucional são de tamanho 7x1. Em seguida, cada amostra é multiplicada por cada um dos filtros. Assim, são utilizados 7 multiplicadores em paralelo no primeiro estágio. No segundo estágio, são utilizados 3 somadores (s0, s1 e s2) em paralelo que servem para somar o resultado das multiplicadores do estágio anterior, de 2 em 2. Como a quantidade de multiplicadores é ímpar, o último será somado no próximo estágio. No terceiro estágio, há mais 2 somadores (s3 e s4) em paralelo que acumulam o resultado de cada somador do estágio anterior e também o resultado da última multiplicação (m6). No quarto estágio, é somado o resultado de s3+s4 e, posteriormente, outro somador acrescenta o respectivo bias ao resultado. Por fim, tem-se a função comparadora RELU e então o resultado é registrado na matriz de saída da camada convolucional.

O bloco funcional implementado, chamado "*cnn_conv1*", utiliza 7 multiplicadores e 7 somadores, que são baseados em máquinas de estados finitas sequenciais síncronas, onde cada multiplicador e cada somador são operadores aritméticos e trigonométricos, que trabalham com representação aritmética em ponto flutuante de 27 bits encapsulados em IPs desenvolvidos em VHDL e disponibilizados por (MUNOZ D. F. SANCHEZ, 2010b) e (MUNOZ D. F. SANCHEZ, 2010a), instanciado em forma de componente e conectado por meio de sinais, seguindo uma lógica sequencial para ativação de cada componente. A figura 22 mostra o esquemático RTL do bloco da convolução fornecido pela descrição do

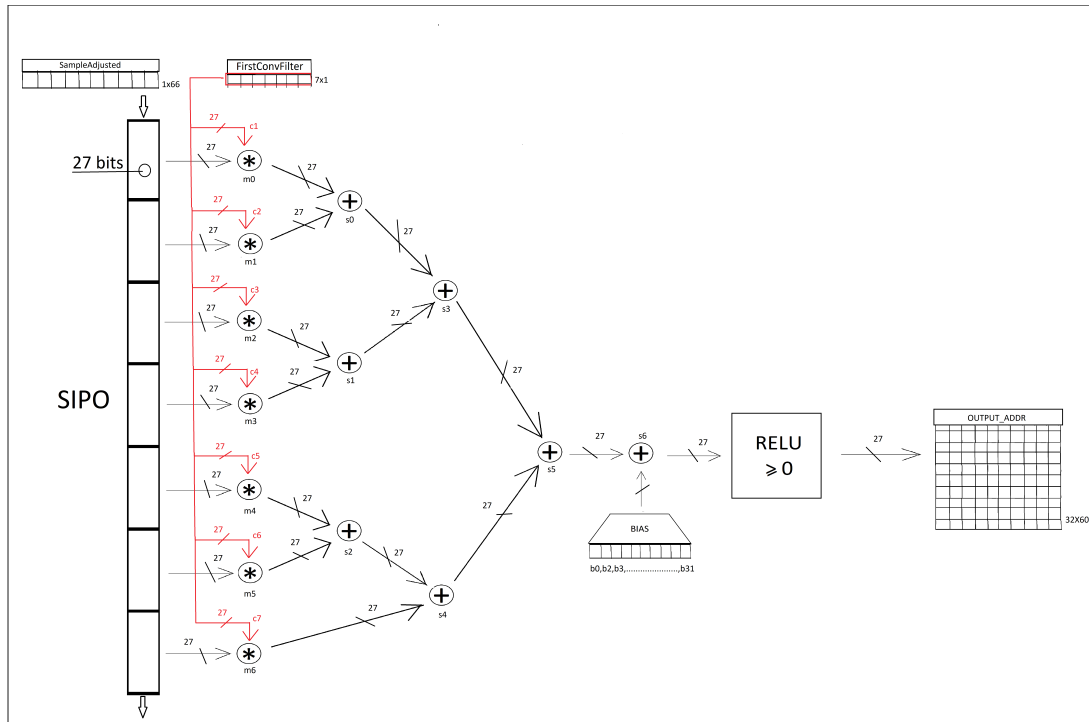


Figura 21 – Arquitetura proposta para iteração do loop interno da primeira camada convolutiva. FONTE: Autor.

código em VHDL. O código VHDL para implementação da arquitetura pode ser visto no Apêndice B.

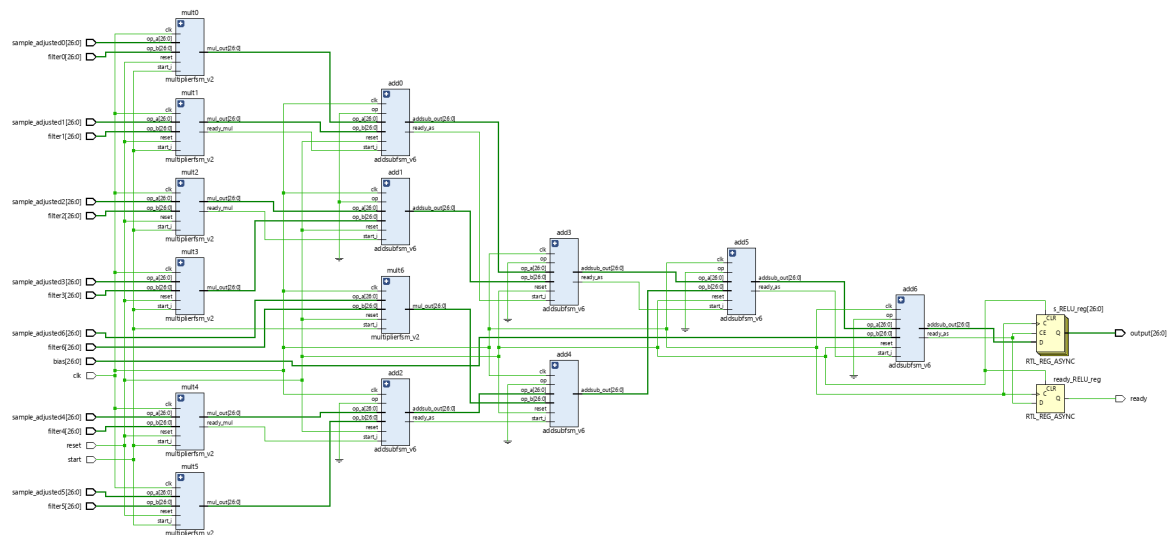


Figura 22 – Esquemático RTL do bloco do loop interno da convolução descrito em VHDL com 7 multiplicadores, 7 somadores e registradores para saída. A disposição dos blocos foi definida pela ferramenta do Vivado. FONTE: Autor.

4.4.2 Módulo Principal

Para descrição em VHDL do módulo principal da primeira camada convolucional, foi implementada uma máquina de estados finitos FSM (*Finite State Machine*), que é um modelo computacional usado para representar sistemas ordenados no tempo, em que seus estados representam os finitos modos possíveis de operação do sistema, sendo estes estados registrados em memórias ou registradores quando implementados em hardware (VIEIRA, 2006).

A FSM criada é do tipo Moore, onde as saídas do circuito dependem apenas do estado atual em que o sistema se encontra, as entradas interferem apenas em qual o próximo estado a ser assumido, seu diagrama de blocos é apresentado na figura 23. O objetivo da FSM descrita é realizar a função dos *loops* externos da camada convolucional, controlando a leitura das amostras de entradas do circuito, da matriz de filtros e bias, enquanto armazena cada saída na matriz 60x32 resultante da camada convolucional, através da variação dos índices das linhas e das colunas das matrizes.

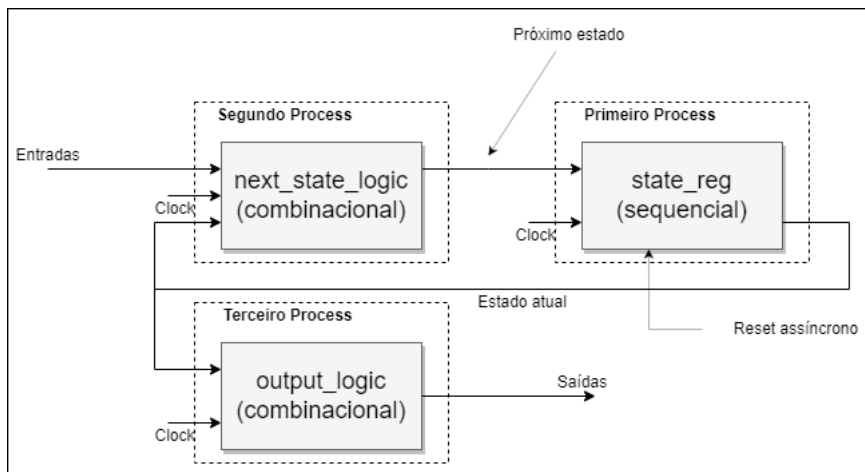


Figura 23 – Diagrama de blocos da FSM descrita em VHDL utilizando 3 processos.
FONTE: Autor.

A FSM é composta por 4 estados chamados *inicio*, *atualiza_entradas*, *reg_saida* e *fim*, abaixo tem-se a descrição do comportamento do sistema em cada estado e a figura 24 apresenta o diagrama de estados simplificado da FSM:

- *inicio*: Estado inicial da FSM ou do *reset*, neste estado as entradas da componente da convolução (start, 7 amostras, 7 filtros e bias) são zeradas, assim como os contadores *i* e *j* (índices das matrizes) são inicializados em 65 e 32, respectivamente. A saída também recebe valores nulos. Caso start da convolução seja igual a '1' o próximo estado é *atualiza_entradas*, caso contrário, permanece em *inicio*.
- *atualiza_entradas*: Neste estado as entradas da componente da convolução (7 amostras, 7 filtros e bias) recebem os valores atualizados conforme a variação dos con-

tadores i e j . O start do componente recebe '1', o que indica que a componente é ativada para calcular a convolução. Caso a flag *ready* da componente seja igual a '1', indicando que já tem um valor válido em sua saída, o próximo estado é *reg_saida*, caso contrário, permanece em *atualiza_entradas*.

- *reg_saida*: Neste estado há um contador que sincroniza a saída válida da componente após atualização das entradas e armazena a saída da componente na matriz de saída da camada convolucional 60x32. É neste estado que os contadores i e j se atualizam, percorrendo as linhas e colunas das matrizes das amostras, filtros, bias e saída. Este estado sempre retorna para *atualiza_entradas*, até a flag *done* for igual a '1', indicando que todas as amostras 1x60, os filtros 32x7, o bias 1x32 foram percorridos e cada valor resultante da iteração da convolução foi armazenado na matriz de saída 60x32.
- *fim*: Este é o estado que indica o fim da camada de convolução, com todos os resultados dessa camada armazenados, setando a flag *ready_conv* para '1'. O próximo estado é de volta para *inicio* independente de qualquer entrada.

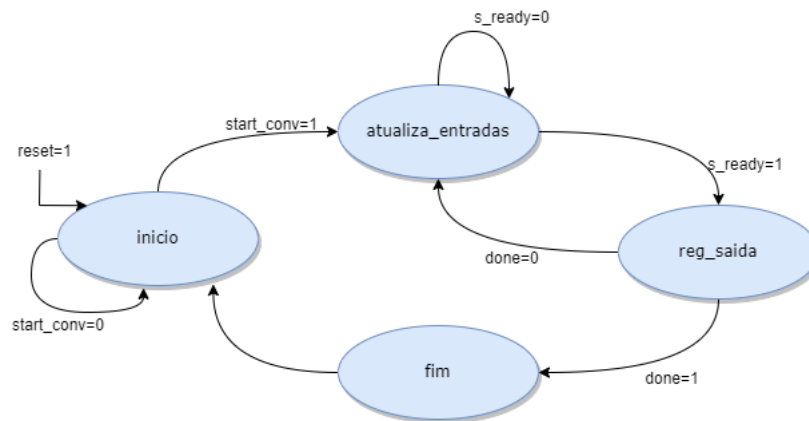


Figura 24 – Diagrama de estados simplificado da FSM. FONTE: Autor.

Sendo assim, com a implementação da FSM controlando as entradas e saídas da componente, tem-se a arquitetura representada no diagrama de blocos para o módulo principal "*cnn_conv1_top*" na figura 25. No Apêndice C pode ser consultado o código VHDL que descreve o módulo principal com a FSM.

4.5 Implementação do projeto em Hardware com o ILA

Com a finalidade de validar a implementação completa da camada convolucional no FPGA, foi utilizado o ILA core do Vivado para monitorar as entradas e saídas do circuito. Para tal, foi implementado um módulo descrito em VHDL chamado "*topmodule*", onde este instancia o IP core do ILA, uma BRAM que armazena os valores das amostras

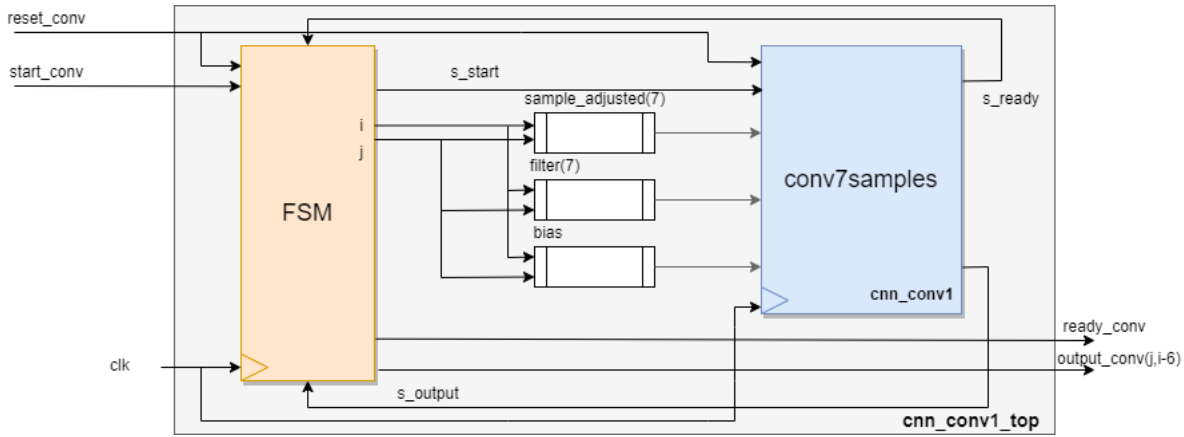


Figura 25 – Diagrama de blocos do módulo principal. FONTE: Autor.

de entradas 1x60 e, o próprio módulo principal da primeira camada convolucional a ser testado em hardware.

O diagrama de blocos deste módulo criado para testes em hardware pode ser visto na figura 26 e o seu código em VHDL está presente no Apêndice D.

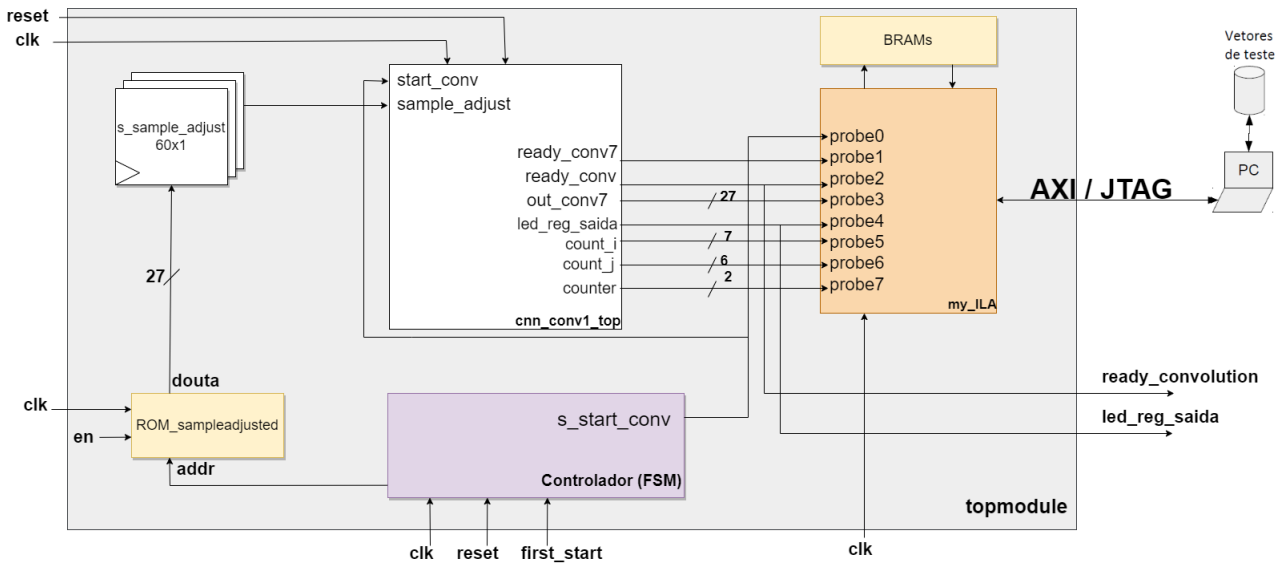


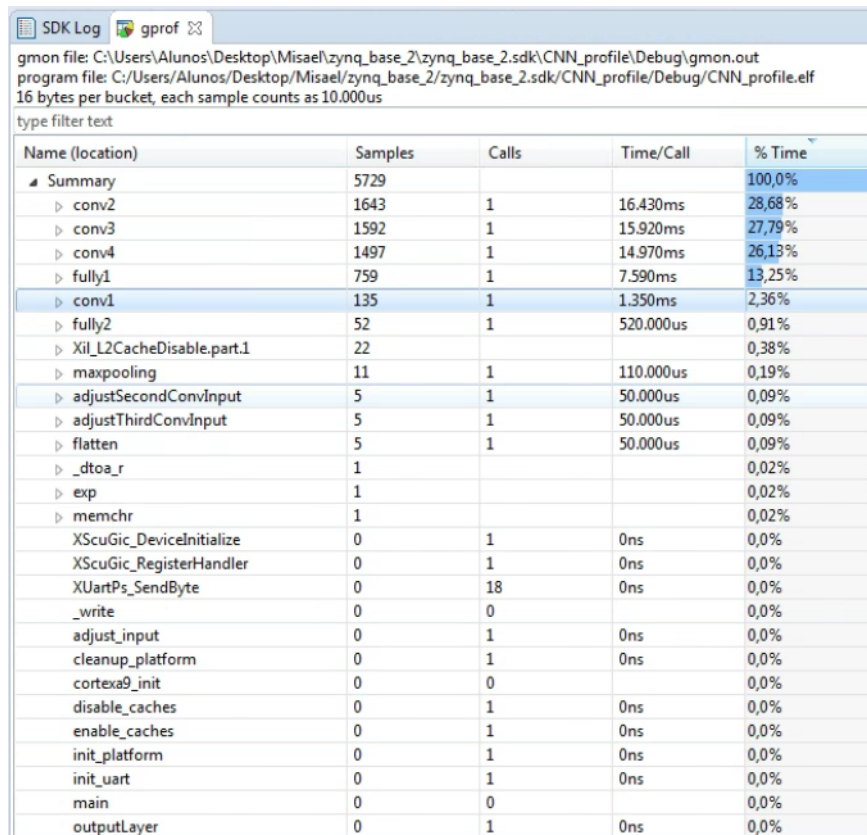
Figura 26 – Diagrama de blocos do topmodule com instanciação do ILA core, camada convolucional, BRAM e FSM para controle. FONTE: Autor.

No módulo descrito para o teste com o ILA, inicialmente com a instanciação da BROM que contém os valores de testes das 66 amostras (já incluso o ajuste de borda do conjunto de 60 amostras acrescidos de 6 amostras nulas), armazenadas em um arquivo .COE, a FSM de controle realiza a leitura de todos os valores dos registradores de saída da BROM e quando finalizado, é acionado o *start* para a camada convolucional, que possui saídas de testes adicionadas ao IP do ILA core para análise dos resultados em hardware.

5 Resultados e Discussões

5.1 Análise de *Profiling*

Após implementação do código em C da Rede Neural Convolutacional no microprocessador ARM-Zynq, usando o software XSDK foi realizada uma análise de *profile*, onde os resultados da análise importados do *gprof* são mostrados na figura 27.



Name (location)	Samples	Calls	Time/Call	% Time
Summary	5729			100,0%
conv2	1643	1	16.430ms	28,68%
conv3	1592	1	15.920ms	27,79%
conv4	1497	1	14.970ms	26,13%
fully1	759	1	7.590ms	13,25%
conv1	135	1	1.350ms	2,36%
fully2	52	1	520.000us	0,91%
Xil_L2CacheDisable.part.1	22			0,38%
maxpooling	11	1	110.000us	0,19%
adjustSecondConvInput	5	1	50.000us	0,09%
adjustThirdConvInput	5	1	50.000us	0,09%
flatten	5	1	50.000us	0,09%
_dtoa_r	1			0,02%
exp	1			0,02%
memchr	1			0,02%
XScuGic_DeviceInitialize	0	1	0ns	0,0%
XScuGic_RegisterHandler	0	1	0ns	0,0%
XUartPs_SendByte	0	18	0ns	0,0%
_write	0	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 27 – Análise de *profiling* da CNN com o *gprof* no ARM-Zynq. FONTE: Autor.

Nota-se pela análise de desempenho no ARM, que as quatro camadas convolucionais da rede estão dentre as cinco que mais comprometem o tempo de execução da função principal, onde juntas representam certa de 85% do tempo de execução da rede. Isto comprova que as camadas convolucionais são as camadas da CNN que mais exigem recursos computacionais e, portanto, as mais necessárias de serem aceleradas em hardware para redução dos gargalos encontrados.

Tratando-se especificamente da função da primeira camada convolutacional, esta é a camada convolutacional que menos exige consumo de tempo de execução em relação as outras, isso ocorre devido o fato de que as outras camadas possuem um número maior de amostras de entrada e também maior quantidade de *loops* em sua função. A primeira

camada convolucional em software no ARM representa 2,36% do tempo de execução da rede, com uma duração de 1.350ms por chamada da função, ressalta-se que a cada chamada de execução da função, são fornecidas um conjunto de 66 amostras do sinal aECG para a primeira camada.

5.2 Simulações Comportamentais

5.2.1 *Testbench* do componente do loop interno

Para simulação do bloco descrito em VHDL da função do loop interno da primeira camada convolucional, foi gerado um arquivo testbench, onde foram setados valores de inicialização para as primeiras 7 amostras do sinal de entrada e 7 amostras do filtro convolucional. Foram utilizadas funções do MatLab para conversão dos valores do tipo float para binário, para adequar-se aos tipos de dados do programa em hardware e posteriormente de binário para float, para interpretação dos resultados.

Foi definido na simulação um clock de 100MHz (período de 10 ns), similar ao clock comum da placa Basys 3. Na figura 28, é apresentado o resultado da simulação do testbench com todos os sinais relevantes para análise do comportamento do bloco. Para interpretação dos sinais temos:

- Verde: são os sinais de *clock(s_clk)*, de *reset(s_reset)* e *start(s_start)* do sistema;
- Azul Claro: são os sinais intermediários de *ready* e saída de cada componente de multiplicação;
- Roxo: são os sinais intermediários de *ready* e saída de cada componente de soma.
- Amarelo: são os sinais de *ready* da função RELU e da saída do sistema, indicando que o houve alteração no resultado;
- Azul escuro: sinal com o resultado da saída do bloco.

Cada sinal de *ready* indica que o valor de saída do respectivo bloco já foi alterado e está apto para ser lido, dessa forma o sinal de *ready* de um bloco multiplicador ou somador, está conectado ao sinal de *start* do próximo bloco a ser executado, ou seja, na parte sequencial os blocos só realizam a operação após os blocos do estágio anterior finalizarem suas respectivas operações e retornarem valores na saída.

Foi comparado o resultado de saída simulado no testbench pelo Vivado com o resultado gerado na função similar escrita no MatLab. Para essa simulação inicial, o resultado mostrou-se correto, com um erro de apenas 4.7e-07.



Figura 28 – Simulação comportamental do bloco do loop interno da convolução: *conv7samples*. FONTE: Autor.

5.2.2 Testbench do módulo principal

O *testbench* descrito para o módulo principal seguiu os mesmos parâmetros de clock e inicialização de sinais que o *testbench* do loop interno da camada. Na figura 29 é apresentado os resultados da simulação comportamental do módulo principal do momento em que é iniciado a convolução com *s_start_conv* indo para nível lógico alto até o fim da convolução com a flag *done* indo para nível lógico alto.

Na simulação da figura 29 pode ser observado também que o código registra corretamente os resultados da camada convolucional na matriz de saída 60x32, representado pelo sinal *s_outConv(32,60,27)*.

Observa-se pela figura 30 que os contadores matriciais *i* e *j* decrementam corretamente até 6 e 0, respectivamente. Os estados, por sua vez, realizam a transição conforme a FSM proposta no capítulo anterior, onde ocorre a transição entre os estados *atualiza_entradas* e *reg_saida* de acordo com a variação de *i*, *j* e o registro dos resultados da convolução na matriz de saída até a transição para o estado *fim* após os contadores chegarem ao último índice da matriz e a flag *done* ir para nível lógico alto.

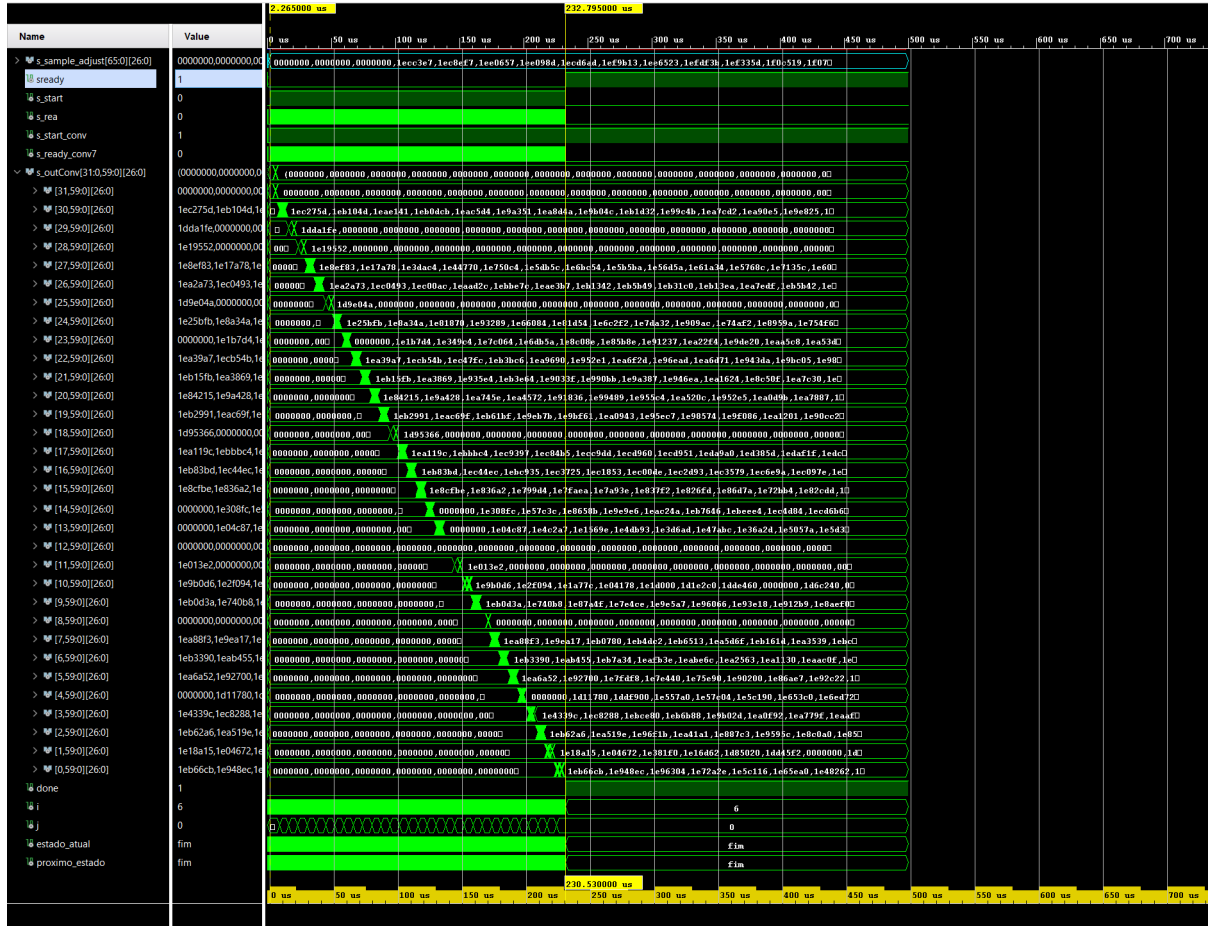


Figura 29 – Simulação comportamental do módulo principal:registro dos resultados da camada convolucional na matriz de saída 60x32. FONTE: Autor.

5.3 Testes de implementação em hardware com o ILA Core

Para validação do funcionamento correto do circuito da primeira camada convolucional implementado em hardware no FPGA Artix7 da Basys3, foi utilizado o ILA Core, onde foram adicionados as pontas de provas para análise dos sinais internos e de saída que corroboram o devido comportamento da função.

Conforme já indicado na figura 26 do capítulo anterior, os filtros e bias foram armazenados como constantes, enquanto as 66 amostras de entradas da camada convolucional foram armazenadas em memória BROM, onde estes são utilizados para testar o funcionamento com o ILA. A figura 31 mostra o comportamento do circuito simulado dentro do módulo principal, onde é possível observar que após o primeiro *start* de entrada, todos os valores das amostras são armazenados em registradores e então dá-se o acionamento para a primeira camada convolucional.

Para análise dos resultados da implementação em hardware, no ILA core foram definidos 8 *probes*, ou pontas de prova, sendo estas:

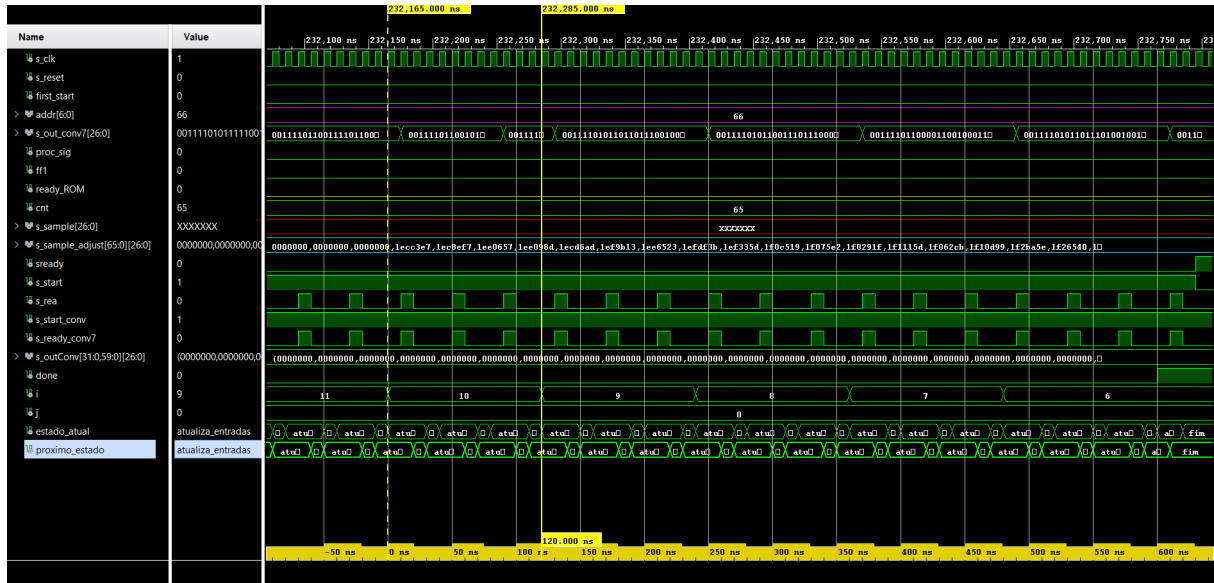


Figura 30 – Simulação comportamental do módulo principal: transição dos estados e contadores de índices matriciais i e j . FONTE: Autor.

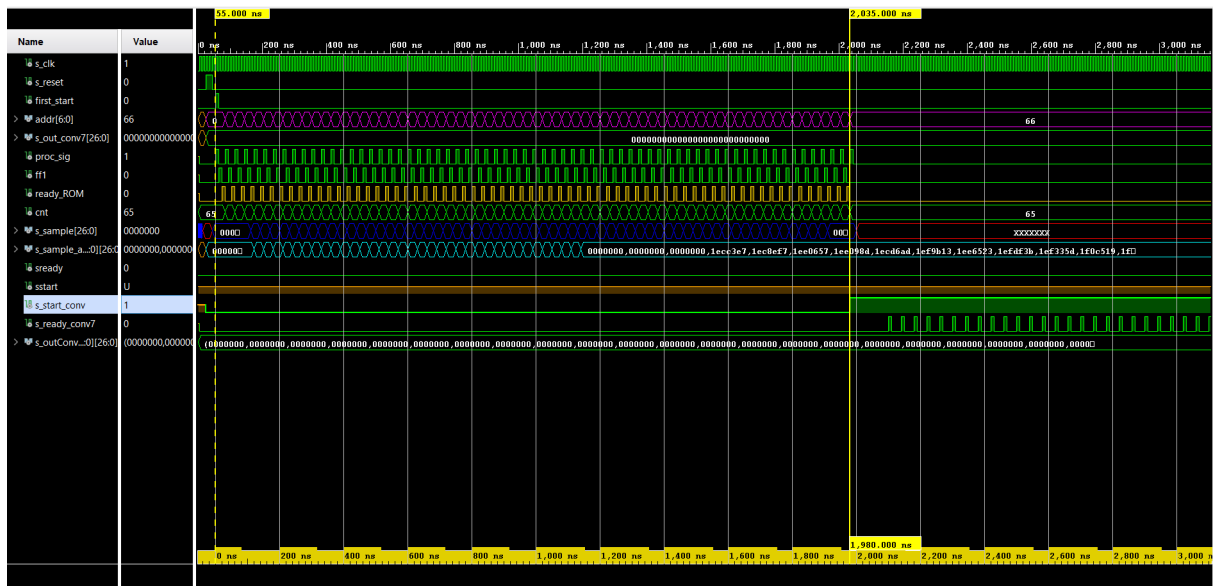


Figura 31 – Funcionamento da leitura e registro temporário das 66 amostras de entrada da primeira camada convolucional. FONTE: Autor.

- s_start_conv : sinal que lê o *start* de execução da primeira camada convolucional, após a leitura e registro das amostras vindas da BROM.
- s_ready_conv7 : sinal que lê o *ready* da função do loop interno da convolução, que mostra quando há um resultado válido na saída.
- s_ready : sinal que lê o *ready* da camada convolucional, este indica que foi finalizada a execução completa da camada.

- *s_led_reg_saida*: sinal utilizado apenas para representar no ILA, o estado que registra a saída e incrementa os contadores counter, i e j.
- *s_out_conv7*: saída de resultado da camada convolucional, sendo atribuída quando o counter é igual a 2, onde há um atraso de 12 ciclos de clock para obter-se um resultado válido a partir das amostras de entrada da convolução.
- *s_counter*: sinal que indica o contador de atraso para leitura da saída da convolução.
- *s_i*: contador que varre as linhas das matrizes de amostras, filtros e saída. Decrementa de 65 a 6.
- *s_j*: contador que varre as colunas das matrizes do filtros, bias e saída. Decrementa de 31 a 0.

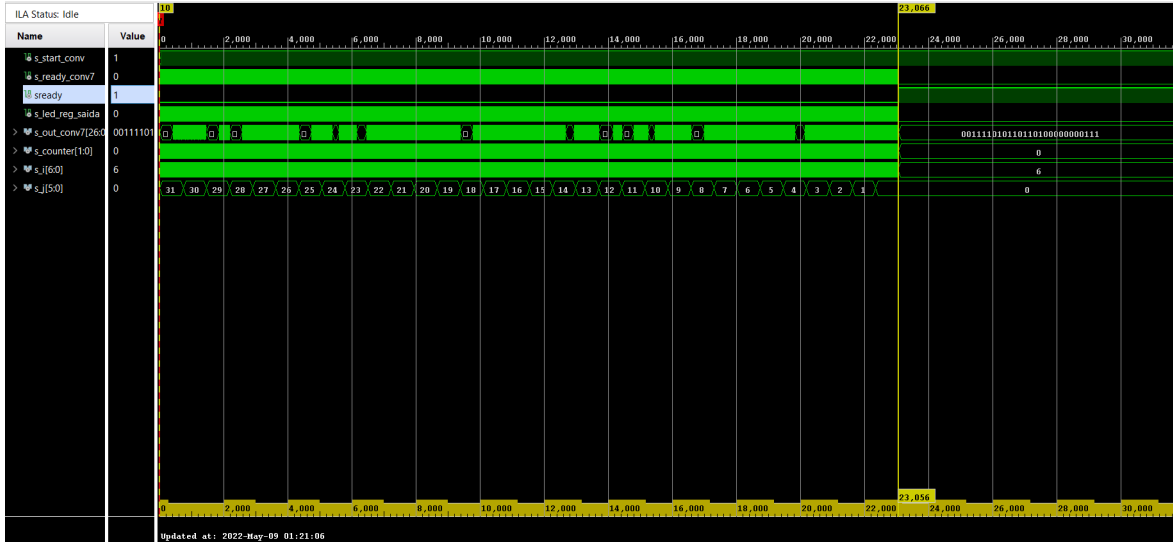


Figura 32 – Implementação da primeira camada convolucional com testes pelo ILA Core. Tempo geral da convolução. FONTE: Autor.

O *trigger* para leitura das amostras pelo ILA core foi definido como sendo a detecção de borda de subida do sinal *s_start_conv*, ou seja, quando se dá o início de execução da primeira camada convolucional. O comportamento do circuito pelo teste com o ILA é apresentado nas figuras 32 e 33, onde é evidenciado que o *trigger* foi acionado corretamente, a camada convolucional está entregando resultados no tempo esperado e os contadores estão variando de acordo com o registro de cada saída válida da convolução na matriz de resultados 60x32.

Os testes com o ILA validam o funcionamento da primeira camada convolucional descrita em hardware, onde foram comparados os 1920 valores dos resultados registrados na matriz de saída (60x32) da primeira camada convolucional implementada em hardware, com os 1920 respectivos valores encontrados no modelo de referência em software.

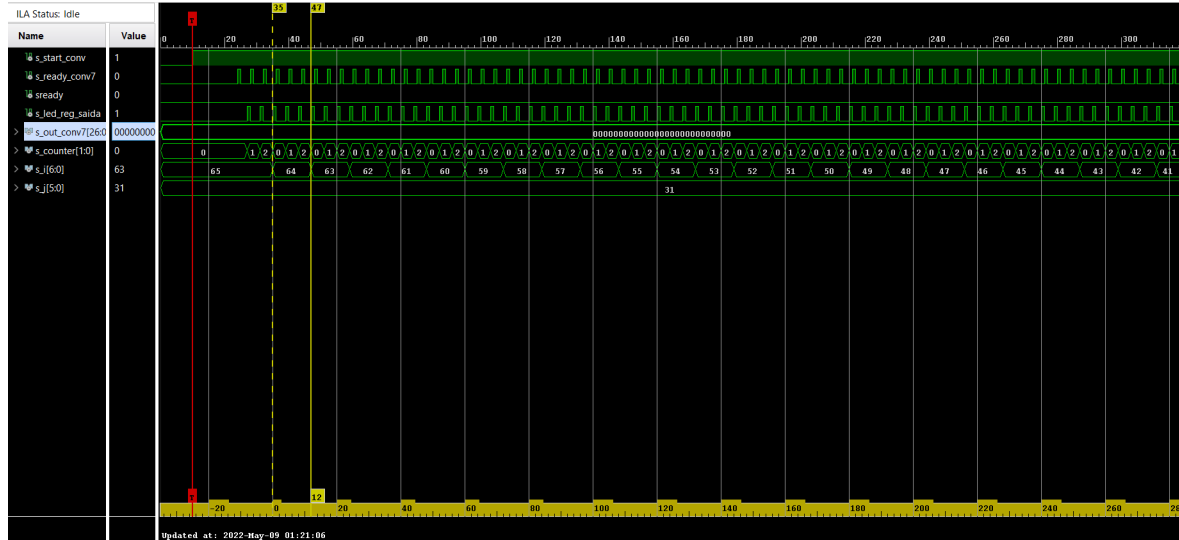


Figura 33 – Implementação da primeira camada convolucional com testes pelo ILA Core. Tempo de saída de resultado da convolução. FONTE: Autor.

Os valores dos resultados em hardware foram exportados do ILA em formato .CSV e convertidos de binário (27 bits) para ponto flutuante no MatLab. Para cada um dos 1920 valores em hardware foi calculado o erro em relação ao software e obtido o gráfico de erros presente na figura 34. O MSE calculado entre hardware e software foi de 0.000123, que é considerado satisfatório uma vez que o erro apresentado pode ser explicado devido conversão de dados ou erros intrínsecos ao cálculo em ponto flutuante de 27 bits, enquanto que os resultados em software foram realizado em computador que utiliza um processador Intel Core i5 de 64 bits.

5.4 Análise de recursos, energia e *timing*

Após implementação da primeira camada convolucional em hardware, foram elaborados reports de *timing* e de recursos, que trazem compreensão das características do circuito desenvolvido. A figura 35 apresenta a representação gráfica do uso de recursos do FPGA da Basys3 pelo circuito descrito. Observa-se que destacado em vermelho é a representação dos recursos utilizados para implementação do ILA apenas, em verde tem-se o utilizado para o circuito descrito e em magenta tem-se a utilização da BROM.

Na figura 36, é apresentado o report de utilização dos recursos pelo circuito descrito para a primeira camada convolucional, onde utiliza-se 7 DSPs para as operações da convolução, 1046 *Flip-flops* e 5604 LUTs. O ILA propriamente requer muitos recursos da FPGA para sua implementação, porém seu consumo não deve ser considerado no projeto, uma vez que ele é usado apenas para testes de validação do circuito. O mesmo se aplica para o bloco de memória BROM que foi utilizado apenas para armazenamento das amostras de entrada de teste para uso do ILA.

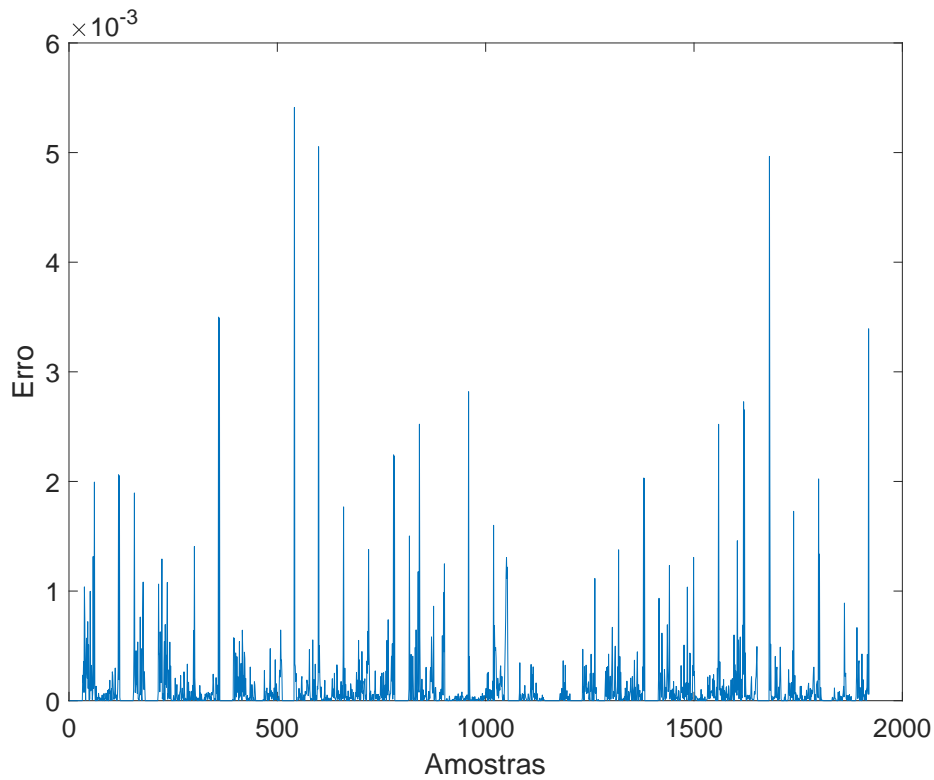


Figura 34 – Distribuição de erro numérico para os resultados da primeira camada convolucional implementada em hardware, utilizando FPU de 27 bits. FONTE: Autor.

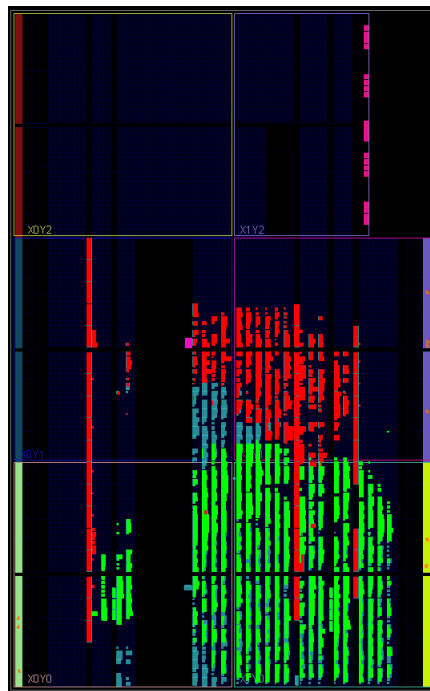


Figura 35 – Representação gráfica dos recursos utilizados no FPGA da Basys3 para implementação do circuito + ILA. FONTE: Autor.

utt (cnn_conv1_top	LUT	FF	DSP	BRAM
Nº	5605	1046	7	0
%	26.95%	2.51%	7.78%	0%

Figura 36 – Report de utilização de recursos pelo circuito. FONTE: Autor.

Em relação ao consumo de energia, vide figura 37, o circuito implementado junto com o ILA, consome 0,137W de energia, sendo 54% do consumo referente a parte estática, apenas para manter o dispositivo funcionando. Os outros 46% referentes a parte dinâmica são distribuídos entre 29% para os *clocks*, 26% para os sinais, 22% para a lógica do circuito implementado, 10% para os DSPs e 7% para os blocos BRAM. Desconsiderando fatores externos como blocos de comunicação, periféricos, sensores, memória, etc., pode-se estimar que para uma alimentação do circuito de 5 V o consumo do circuito é de 27,4 mA. Portanto para uma autonomia de 2 horas, seria necessário uma bateria de pelo menos cerca de 60 mAh.

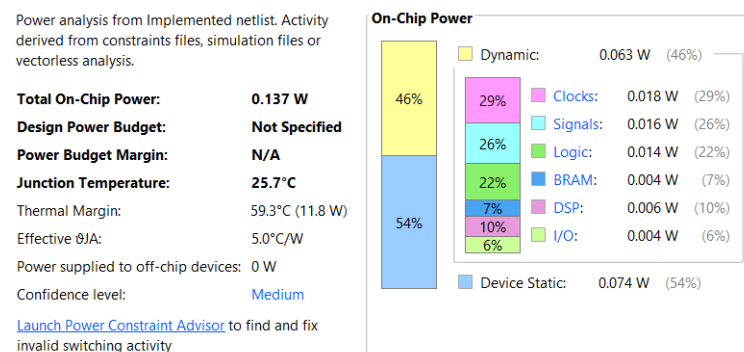


Figura 37 – Report de consumo de energia da Basys3 pelo circuito + ILA. FONTE: Autor.

Por fim, na figura 38 tem-se o report de timing do circuito implementado junto ao ILA, onde pode ser observado que não há problemas de *timing* no circuito, não apresentando folgas negativas de tempo.

Design Timing Summary			
Setup	Hold	Pulse Width	
Worst Negative Slack (WNS): 0.706 ns	Worst Hold Slack (WHS): 0.029 ns	Worst Pulse Width Slack (WPWS): 3.750 ns	
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns	
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	
Total Number of Endpoints: 12053	Total Number of Endpoints: 12037	Total Number of Endpoints: 5550	
All user specified timing constraints are met.			

Figura 38 – Report de *timing* do circuito implementado no FPGA + ILA. FONTE: Autor.

5.5 Hardware vs Software

Para a componente do *loop* mais interno da função, tem-se que os blocos de multiplicação são executados em paralelo e realizam as operações com entrega de resultado em 2 ciclos de *clock*, ou seja, em 20 ns. Os blocos somadores, por sua vez, também operam e entregam resultados em 2 ciclos de *clock*. Dessa forma, desde o *start* do sistema até o primeiro resultado na saída do bloco da função, tem-se 11 ciclos de *clock*, o que representa uma latência de 110 ns. Como as multiplicações e somas entregam ambos em 2 ciclos de clock (20 ns), o *throughput* para este bloco é de 50 MFLOPS (50 Milhões de Operações em Ponto Flutuante por Segundo).

Como o componente citado acima, entrega um resultado válido a cada 11 ciclos de *clock* após atualização das entradas, logo o módulo principal que o instancia, está sincronizado para registrar cada valor resultante da convolução na matriz de saída da camada a cada 12 ciclos de *clock*, sendo 11 para o *ready* da componente e 1 para o registro na matriz. Sendo assim, conforme testado no ILA, desde o *start* da camada convolucional até o seu *ready* são necessários 230.56 us, que é o tempo total de execução da camada convolucional em hardware.

Dessa forma, comparando o desempenho em tempo de execução entre hardware e software, tem-se que a primeira camada convolucional descrita no FPGA Artix7 (230,56 us) é aproximadamente 5,85 vezes mais rápida do que a função da primeira camada convolucional executada no ARM-Zynq (1.350 ms).

6 Conclusão

Neste trabalho foi apresentado a implementação em hardware, no FPGA Artix7 da placa de desenvolvimento Basys 3, da primeira camada convolucional de uma Rede Neural Convolucional proposta para extração de pico de fECG em um módulo estimador da frequência cardíaca fetal. Por meio do desenvolvimento do circuito desta camada, podem ser compreendido o conceito de aceleração em hardware das camadas convolucionais da rede.

O projeto de hardware implementado mostrou resultados satisfatórios no quesito tempo de execução, onde a camada convolucional em hardware performa 5.85 vezes mais rápido do que o modelo em software executado no microprocessador ARM-Zynq. Também é notório a eficiência de cálculo da convolução, com um erro médio quadrático calculado de apenas 0.000123 para os resultados entre hardware e software. O projeto do circuito no FPGA foi validado com uma utilização de cerca de 29% dos LUTs, 8% dos DSPs, 1% de BRAM e 13% dos registradores disponíveis no FPGA da Basys 3.

Por meio da comprovação da maior eficiência em termos de tempo de execução da primeira camada convolucional implementada em hardware comparado ao software, considerando que a camada convolucional é a camada de maior custo computacional nas arquiteturas de Redes Neurais Convolucionais, evidencia-se no presente trabalho a demonstração da aceleração em 5.85 vezes da primeira camada convolucional da arquitetura da CNN proposta.

No entanto, é necessário apontar que não foi cumprido no presente trabalho a implementação do projeto co-software que realizaria a comunicação entre o software e o módulo da camada convolucional em hardware no FPGA, o que demonstraria com maior clareza a diferença de tempo de execução do algoritmo da rede neural completo sem e com a aceleração em hardware da camada.

6.1 Trabalhos futuros

Podem ser propostos, portanto, trabalhos futuros que desenvolvam o projeto co-software completo para a aceleração da Rede Neural Convolucional, por meio da implementação de todas as camadas convolucionais em hardware.

Primeiramente, pode ser realizado a descrição das demais camadas convolucionais da rede em VHDL, aproveitando a parametrização que foi realizada no circuito da primeira camada convolucional para o tamanho das entradas, filtros e iterações da camada convolucional. Os tamanhos dos filtros e dos conjuntos de amostras das próximas camadas

convolucionais são menores do que os da primeira camada convolucional, estes parâmetros podem ser alterados dentro da biblioteca *fpupack*. Destaca-se que as amostras por sua vez, não necessariamente precisam entrar na SIPO do *loop* interno de 7 em 7, e sim pode ser modificado em relação ao tamanho dos filtros de cada camada convolucional. A entrada das próximas camadas convolucionais são matrizes, portanto, deve ser acrescentado no bloco do modulo principal de cada camada convolucional, um *loop* ou contador que realize convolução para cada linha da matriz de entrada. Cada bloco de camada convolucional deve ser testado, sintetizado e implementado em hardware separadamente para melhor organização do projeto.

Após implementação e validação no FPGA de cada bloco das camadas convolucionais, estes devem ser encapsulados em IPs e exportados em hardware, onde através do uso da ferramenta XSDK, pode ser realizado a comunicação dos IP em hardware com o software implementado em C no ARM-Zynq7000 da placa de desenvolvimento Zybo ou outra similar. É sugerido que a comunicação entre o software e o hardware seja realizada por meio do uso do protocolo AXISStream ou do protocolo AXI4Lite, porém uma análise profunda deve ser realizada pelo projetista a fim de definir qual o mais adequado.

Com a consolidação do projeto co-software implementado no SoC, deve ser realizada uma análise profile que estime a aceleração total do projeto em relação a arquitetura apenas em software. Deve ser considerado os atrasos de *overhead* oriundos da comunicação entre os dados e o co-processador.

Sugere-se também que seja realizado o uso de técnicas de comunicação com memória, sensores ou outros periféricos que podem estar presentes entre o bloco de processamento e comunicação ou o bloco de aquisição, e assim obtenha-se uma estimativa de custo energético completo do bloco de processamento em uma possível futura fabricação de um dispositivo independente.

Referências

- AREIBI, M. S. S. *ITutorial: Building an Embedded Processor System on a Xilinx Zynq FPGA (Profiling)*. 2020. PG172. Disponível em: <http://islab.soe.uoguelph.ca/sareibi/TEACHING_dr/XILINX_VIVADO_dr/HwSw_dr/VivadoEmbeddedZyncTutorial.pdf>. Acesso em: 16 fevereiro de 2022. Citado na página 48.
- BAJAJ, R.; FAHMY, S. Mapping for maximum performance on fpga dsp blocks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 35, p. 1–1, 01 2015. Citado 2 vezes nas páginas 11 e 33.
- BARBOSA, I. J. T. Aceleração de algoritmos para estimativa da frequência cardíaca fetal utilizando fpga. trabalho de conclusão de curso (bacharelado em engenharia eletrônica). Faculdade do Gama, Universidade de Brasília, Brasília, 2016. Citado na página 22.
- BEAR, M. F. *Neurociências : desvendando o sistema nervoso*. [S.l.]: Artmed, 2008. v. 3 ed. Citado na página 28.
- BERBERT, W.; BERTINI, L.; COPETTI, A. Aceleração de hardware em sistemas embarcados para aprendizado de máquina utilizando knn em fpga. SC, Brasil. XII Computer on the Beach., 2021. Citado 2 vezes nas páginas 33 e 34.
- BRAGA, A. de P. *Redes Neurais Artificiais: Teorias e Aplicações*. [S.l.]: LTC – Livros Técnicos e Científicos Editora S.A., 2000. Citado na página 29.
- BRITO, K. *MT Ciências: Você sabia que os neurônios são responsáveis pela condução do impulso nervoso*. 2019. Disponível em: <<http://mtciencias.com.br/voce/biologia-cerebral/>>. Acesso em: 04 setembro de 2021. Citado 2 vezes nas páginas 11 e 28.
- CUNHA Éverton Soares da. Desenvolvimento de sistemas embarcados utilizando plataformas fpga com dispositivos arm. Pontifícia Universidade Católica do Rio Grande do Sul., 2014. Citado na página 34.
- DIGILENT. *Basys 3 FPGA Board Reference Manual. DOC: 502-183*. 2016. Disponível em: <https://br.mouser.com/datasheet/2/690/basys3_rm-845168.pdf>. Acesso em: 03 setembro de 2021. Citado 3 vezes nas páginas 11, 42 e 43.
- FEIST, T. White paper: Vivado design suite. XILINX, WP416 (v1.1), 2012. Citado 3 vezes nas páginas 11, 40 e 41.
- FILHO, M. M. S. *Redes neurais artificiais: Do neurônio artificial À convolução*. Universidade Federal Fluminense, 2018. Citado na página 28.
- HASAN, M. A.; IBRAHIMY, M. I.; REAZ, M. B. I. Fetal ecg extraction from maternal abdominal ecg using neural network. Published Online December 2009 (<http://www.SciRP.org/journal/jsea>). Scientific Research, 2009. Citado na página 36.
- HASAN, M. A. et al. Vhdl modeling of fecg extraction from the composite abdominal ecg using artificial intelligence. International Islamic University Malaysia, 53100 Gombak, Malaysia, 2009. Citado na página 36.

- HASAN, M. A.; REAZ, M. B. I. Hardware prototyping of neural network based fetal electrocardiogram extraction. *MEASUREMENT SCIENCE REVIEW*, Volume 12, No. 2, 2012. Citado na página 36.
- HAYKIN, S. *Redes Neurais: Princípios e Prática*. [S.l.]: Bookman, 2001. v. 2 ed. Citado 5 vezes nas páginas 11, 28, 29, 31 e 32.
- JAGANNATH, D. J.; SELVAKUMAR, A. I. Issues and research on foetal electrocardiogram signal elicitation. *biomedical signal processing and control*[s. l.], v. 10, p. 224-244. Elsevier Ltd., 2014. Citado 3 vezes nas páginas 11, 21 e 22.
- JUNIOR, H. D. de C. Proposta de uma arquitetura de redes neurais para um módulo estimador da frequência cardíaca fetal baseado em fpga. Universidade de Brasília - UnB, 2018. Citado 11 vezes nas páginas 11, 12, 22, 23, 37, 39, 45, 46, 47, 49 e 50.
- KAHANKOVA, R. et al. A review of signal processing techniques for non-invasive fetal electrocardiography. *IEEE REVIEWS IN BIOMEDICAL ENGINEERING*, VOL. 13, 2020. Citado 4 vezes nas páginas 11, 21, 27 e 28.
- KHAN, S. et al. A guide to convolutional neural networks for computer vision. [S.l.]: Morgan & Claypool, ISBN 9781681730226, 2018. Citado na página 30.
- KIM, S. E.; SEO, I. W. Artificial neural network ensemble modeling with conjunctive data clustering for water quality prediction in rivers. *journal of hydro-environment research*. Department of Civil and Environmental Engineering, Seoul National University, 1 Gwanak-ro, Gwanak-gu, Seoul, 151-744, South Korea., 2015. Citado 2 vezes nas páginas 11 e 32.
- LECUN, Y.; KAVUKCUOGLU, K.; FARABET, C. Convolutional networks and applications in vision. *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*. [S.l.]p. 253–256, 2010. Citado na página 30.
- MAIA, A. V. A. *Um gerador de arquiteturas SoC para execução de redes neurais convolucionais em FPGAs*. 2020. Disponível em: <https://repositorio.ufc.br/bitstream/riufc/55720/1/2020_tcc_avamaia.pdf>. Acesso em: 04 fevereiro de 2022. Citado na página 36.
- MATHWORKS, I. Matlab: the language of technical computing. desktop tools and development environment, version 7. [S.l.]: MathWorks. v. 9., 2005. Citado na página 40.
- MUNOZ D. F. SANCHEZ, C. H. L. M. A.-R. D. M. Fpga based floating-point library for cordic algorithms. *IEEE Southern Programmable Logic Conference (SPL)*, p. 55–60, 2010. Citado na página 50.
- MUNOZ 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*, p. 42–52, 2010. Citado na página 50.
- ORDONEZ, E. D. M. *Projeto, Desempenho e Aplicações de Sistemas Digitais em Circuitos Programáveis (FPGAs)*. [S.l.]: BLESS Gráfica e Editora Ltda, 2003. Citado na página 32.

RODRIGUES, J. A. Implementação da comunicação sem fio de um módulo estimador da frequência cardíaca fetal baseado em fpga. trabalho de conclusão de curso (bacharelado em engenharia eletrônica). Faculdade do Gama, Universidade de Brasília, Brasília, 2016. Citado na página 22.

SAKURAI, H. P. E.; LIMA, R. B.; FARIA, M. Frequência cardíaca fetal durante o primeiro trimestre da gestação. Revista Brasileira de Ginecologia e Obstetrícia, 2001. Citado na página 21.

SAMENI, R.; CLIFFORD, G. D. A review of fetal ecg signal processing; issues and promising directions. Boston, USA, 2011. 30 p., 2011. Citado 4 vezes nas páginas 11, 25, 26 e 27.

SILVA, C. L. da; BAILÃO, L. A. Avaliação da frequência cardíaca embriofetal no primeiro trimestre da gestação por meio da ultra-sonografia transvaginal com doppler colorido e pulsátil. Departamento de Ginecologia e Obstetrícia do Hospital das Clínicas da Faculdade de Medicina de Ribeirão Preto – USP., 2000. Citado na página 27.

TOMASI, H. F. Acelerador de hardware em fpga para aplicações em aprendizado de máquina. Universidade de Brasília - UnB, 2020. Citado na página 36.

TUTIDA, H. I. C. Implementação de um módulo de aquisição de ecg abdominal em gestantes para estimativa da frequência cardíaca fetal usando fpga. trabalho de conclusão de curso (bacharelado em engenharia eletrônica). Faculdade do Gama, Universidade de Brasília, Brasília, 2016. Citado na página 22.

VARGAS, A. C. G.; CARVALHO, A. M. P.; VASCONCELOS, C. N. Um estudo sobre redes neurais convolucionais e sua aplicação em detecção de pedestres. Universidade Federal Fluminense, 2016. Citado 2 vezes nas páginas 11 e 30.

VIEIRA, N. J. *Introdução aos fundamentos da computação: Linguagens e máquinas*. [S.l.]: Cengage Learning, 2006. v. 1 ed. Citado na página 52.

XILINX. *Integrated Logic Analyzer v6.2 - LogiCORE IP Product Guide*. Vivado Design Suite. 2016. PG172. Disponível em: <<https://docs.xilinx.com/v/u/en-US/pg172-ila>>. Acesso em: 25 abril de 2022. Citado 2 vezes nas páginas 11 e 41.

XILINX. *Xilinx Software Development Kit (SDK) User Guide: System Performance Analysis*. 2019. Disponível em: <https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2019_1/ug1145-sdk-system-performance.pdf>. Acesso em: 08 novembro de 2021. Citado 2 vezes nas páginas 11 e 42.

XILINX. *Zynq-7000 SoC Product Advantages*. 2021. Disponível em: <<https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.htm>>. Acesso em: 01 novembro de 2021. Citado 2 vezes nas páginas 11 e 35.

Apêndices

APÊNDICE A – Código VHDL da biblioteca *fpupack*.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

package fpupack is

    constant FRAC_WIDTH : integer := 18;
    constant EXP_WIDTH  : integer := 8;
    constant FP_WIDTH   : integer:= FRAC_WIDTH+EXP_WIDTH+1;

    constant bias : std_logic_vector(EXP_WIDTH-1 downto 0) := "01111111";
    constant int_bias : integer := 127;
    constant int_alin : integer := 255;
    constant EXP_DF : std_logic_vector(EXP_WIDTH-1 downto 0) := "10000010";
    constant bias_MAX : std_logic_vector(EXP_WIDTH-1 downto 0) := "10000110";
    constant bias_MIN : std_logic_vector(EXP_WIDTH-1 downto 0) := "01110101";
    constant EXP_ONE: std_logic_vector(EXP_WIDTH-1 downto 0):= (others => '1');
    constant EXP_INF : std_logic_vector(EXP_WIDTH-1 downto 0) := "11111111";
    constant ZERO_V : std_logic_vector(FP_WIDTH-1 downto 0) := (others => '0');

    constant s_one : std_logic_vector(FP_WIDTH-1 downto 0) := "001111111000000000000000";
    constant s_ten : std_logic_vector(FP_WIDTH-1 downto 0) := "010000010010000000000000";
    constant s_twn : std_logic_vector(FP_WIDTH-1 downto 0) := "010000011010000000000000";
    constant s_hundred : std_logic_vector(FP_WIDTH-1 downto 0) := "01000010110010000000";
    constant s_pi2 : std_logic_vector(FP_WIDTH-1 downto 0) := "001111111100100100001111";
    constant s_pi : std_logic_vector(FP_WIDTH-1 downto 0) := "010000000100100100001111";
    constant s_3pi2 : std_logic_vector(FP_WIDTH-1 downto 0) := "010000001001011011001010";
    constant s_2pi : std_logic_vector(FP_WIDTH-1 downto 0) := "010000001100100100001111";

    constant Phyp          : std_logic_vector(FP_WIDTH-1 downto 0) := "001111111001101001";
    constant log2e          : std_logic_vector(FP_WIDTH-1 downto 0) := "00111111101110001";
    constant ilog2e         : std_logic_vector(FP_WIDTH-1 downto 0) := "00111111100110001";
    constant d_043         : std_logic_vector(FP_WIDTH-1 downto 0) := "0011110100110000";

```

```

("00111101000101011111110101", "101111011100110000010001000", "0
("101111010000001100010111000", "101110110010000011000000110", "1
("001111011001100010001100001", "101111010001100110100110100", "1
("001111100001000011010001110", "001111100000000101010110110", "0
("101111000010101001011011001", "001111001000100000100000111", "0
("101111100101100011011001000", "101111010100001110100101101", "0
("101111001000011001100100000", "101111010011101100010111111", "0
("101111010110100001110000110", "001111011001101010111011010", "1
("001111100000010111101001000", "001111010100110101000000000", "0
("001111000101111100100001101", "101111100001110000110000011", "1
("101111011111111010111101100", "0011110110111010000000000001", "0
("101111011000000001001100011", "101111011010011001110111001", "0
("001111100001111110011111100", "101111011010101110000100001", "1
("101111010001100001111001000", "001111100000101111101101000", "0
("101111100001111101011010001", "101111100000010110110111010", "1
("101111100001101101101011101", "001111011100010011010110000", "1
("101111011101000110010011011", "101111010000001100101101011", "0
("101111011001110111010101100", "101111011011110100100101010", "1

constant firstConvBias : biasConv := ("001110111001101011101101110", "00111101001111

end fpupack;

package body fpupack is
end fpupack;

```


APÊNDICE B – Código VHDL que descreve o loop interno da primeira camada convolucional.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.fpack.all;

entity cnn_conv1 is
    Port ( reset          : in STD_LOGIC;
          clk             : in STD_LOGIC;
          start           : in std_logic;
          sample_adjusted0 : in std_logic_vector (FP_WIDTH-1 downto 0);
          sample_adjusted1 : in std_logic_vector (FP_WIDTH-1 downto 0);
          sample_adjusted2 : in std_logic_vector (FP_WIDTH-1 downto 0);
          sample_adjusted3 : in std_logic_vector (FP_WIDTH-1 downto 0);
          sample_adjusted4 : in std_logic_vector (FP_WIDTH-1 downto 0);
          sample_adjusted5 : in std_logic_vector (FP_WIDTH-1 downto 0);
          sample_adjusted6 : in std_logic_vector (FP_WIDTH-1 downto 0);
          filter0          : in std_logic_vector (FP_WIDTH-1 downto 0);
          filter1          : in std_logic_vector (FP_WIDTH-1 downto 0);
          filter2          : in std_logic_vector (FP_WIDTH-1 downto 0);
          filter3          : in std_logic_vector (FP_WIDTH-1 downto 0);
          filter4          : in std_logic_vector (FP_WIDTH-1 downto 0);
          filter5          : in std_logic_vector (FP_WIDTH-1 downto 0);
          filter6          : in std_logic_vector (FP_WIDTH-1 downto 0);
          bias             : in std_logic_vector (FP_WIDTH-1 downto 0);
          ready            : out std_logic;
          output           : out std_logic_vector (FP_WIDTH-1 downto 0));
end cnn_conv1;

architecture Behavioral of cnn_conv1 is

    component multiplier_fsm_v2 is

```



```
        reset          => reset,
        clk             => clk,
        op_a            => sample_adjusted1,
        op_b            => filter1,
        start_i         => start,
        mul_out          => s_mul_out1,
        ready_mul        => ready_mul1);

add0: addsubfsm_v6 port map(
    reset          => reset,
    clk            => clk,
    op              => '0',
    op_a            => s_mul_out0,
    op_b            => s_mul_out1,
    start_i         => ready_mul1,
    addsub_out       => s_add_out0,
    ready_as         => ready_add0);

mult2: multiplierfsm_v2 port map(
    reset          => reset,
    clk            => clk,
    op_a            => sample_adjusted2,
    op_b            => filter2,
    start_i         => start,
    mul_out          => s_mul_out2,
    ready_mul        => ready_mul2);

mult3: multiplierfsm_v2 port map(
    reset          => reset,
    clk            => clk,
    op_a            => sample_adjusted3,
    op_b            => filter3,
    start_i         => start,
    mul_out          => s_mul_out3,
    ready_mul        => ready_mul3);

add1: addsubfsm_v6 port map(
    reset          => reset,
    clk            => clk,
```

```

    op                => '0',
    op_a              => s_mul_out2,
    op_b              => s_mul_out3,
    start_i           => ready_mul2,
    addsub_out         => s_add_out1,
    ready_as           => ready_add1);

mult4: multiplierfsm_v2 port map(
    reset              => reset,
    clk                => clk,
    op_a               => sample_adjusted4,
    op_b               => filter4,
    start_i            => start,
    mul_out             => s_mul_out4,
    ready_mul           => ready_mul4);

mult5: multiplierfsm_v2 port map(
    reset              => reset,
    clk                => clk,
    op_a               => sample_adjusted5,
    op_b               => filter5,
    start_i            => start,
    mul_out             => s_mul_out5,
    ready_mul           => ready_mul5);

add2: addsubfsm_v6 port map(
    reset              => reset,
    clk                => clk,
    op                 => '0',
    op_a               => s_mul_out4,
    op_b               => s_mul_out5,
    start_i            => ready_mul4,
    addsub_out          => s_add_out2,
    ready_as            => ready_add2);

mult6: multiplierfsm_v2 port map(
    reset              => reset,
    clk                => clk,

```

```
        op_a            => sample_adjusted6,
        op_b            => filter6,
        start_i         => start,
        mul_out         => s_mul_out6,
        ready_mul       => ready_mul6);

add3: addsubfsm_v6 port map(
    reset              => reset,
    clk                => clk,
    op                 => '0',
    op_a               => s_add_out0,
    op_b               => s_add_out1,
    start_i            => ready_add0,
    addsub_out         => s_add_out3,
    ready_as           => ready_add3);

add4: addsubfsm_v6 port map(
    reset              => reset,
    clk                => clk,
    op                 => '0',
    op_a               => s_add_out2,
    op_b               => s_mul_out6,
    start_i            => ready_add2,
    addsub_out         => s_add_out4,
    ready_as           => ready_add4);

add5: addsubfsm_v6 port map(
    reset              => reset,
    clk                => clk,
    op                 => '0',
    op_a               => s_add_out3,
    op_b               => s_add_out4,
    start_i            => ready_add3,
    addsub_out         => s_add_out5,
    ready_as           => ready_add5);

add6: addsubfsm_v6 port map(
    reset              => reset,
    clk                => clk,
```

```

        op                                => '0',
        op_a                             => s_add_out5,
        op_b                             => bias,
        start_i      => ready_add5,
        addsub_out => s_add_out6,
        ready_as    => ready_add6);

-- processo para resetar o bloco e tornar finalizar a função RELU
process (clk, reset, ready_add6, s_add_out6)
begin
    if reset='1' then
        s_RELU <= (others => '0');
        ready_RELU <= '0';
    elsif rising_edge (clk) then
        if (ready_add6 = '1') then
            ready_RELU <= '1';
            if (s_add_out6(FP_WIDTH-1) = '0') then
                s_RELU <= s_add_out6;
            else
                s_RELU <= (others => '0');
            end if;
        else
            ready_RELU <= '0';
        end if;
    end if;
end process;

ready <= ready_RELU;
output <= s_RELU;

end Behavioral;
```

APÊNDICE C – Código VHDL do módulo principal da Camada Convolucional com FSM.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use IEEE.NUMERIC_STD.ALL;
use work.fpupack.all;

entity cnn_conv1_top is
    Port ( reset_conv      : in STD_LOGIC;
          clk              : in STD_LOGIC;
          start_conv       : in std_logic;
          sample_adjust    : in samples_adjust;    -- (1x60x27bits)
          -- first_filter   : in filter;          -- (32x7x27bits)
          -- first_bias     : in biasConv;        -- (1x32x27bits)
          out_conv7         : out std_logic_vector(FP_WIDTH-1 downto 0);--teste no
          ready_conv7       : out std_logic;--teste no ILA
          output_conv       : out outConv;        -- (32x60x27bits)
          ready_conv        : out std_logic;--teste no ILA
          led_inicio        : out std_logic;--teste no ILA
          led_atualiza_entradas : out std_logic;--teste no ILA
          led_reg_saida     : out std_logic;--teste no ILA
          counter           : out std_logic_vector(1 downto 0);--teste no ILA
          count_i           : out std_logic_vector(6 downto 0);--teste no ILA
          count_j           : out std_logic_vector(5 downto 0)--teste no ILA
        );
end cnn_conv1_top;

architecture Behavioral of cnn_conv1_top is

    component cnn_conv1 is
        Port ( reset      : in STD_LOGIC;
              clk          : in STD_LOGIC;
              start        : in std_logic;
              sample_adjusted0 : in std_logic_vector (FP_WIDTH-1 downto 0);

```

```

    sample_adjusted1 : in std_logic_vector (FP_WIDTH-1 downto 0);
    sample_adjusted2 : in std_logic_vector (FP_WIDTH-1 downto 0);
    sample_adjusted3 : in std_logic_vector (FP_WIDTH-1 downto 0);
    sample_adjusted4 : in std_logic_vector (FP_WIDTH-1 downto 0);
    sample_adjusted5 : in std_logic_vector (FP_WIDTH-1 downto 0);
    sample_adjusted6 : in std_logic_vector (FP_WIDTH-1 downto 0);
    filter0           : in std_logic_vector (FP_WIDTH-1 downto 0);
    filter1           : in std_logic_vector (FP_WIDTH-1 downto 0);
    filter2           : in std_logic_vector (FP_WIDTH-1 downto 0);
    filter3           : in std_logic_vector (FP_WIDTH-1 downto 0);
    filter4           : in std_logic_vector (FP_WIDTH-1 downto 0);
    filter5           : in std_logic_vector (FP_WIDTH-1 downto 0);
    filter6           : in std_logic_vector (FP_WIDTH-1 downto 0);
    bias              : in std_logic_vector (FP_WIDTH-1 downto 0);
    ready             : out std_logic;
    output            : out std_logic_vector (FP_WIDTH-1 downto 0));
end component;

-- sinais primários do top module que se comunicam com o SOFTWARE
--signal s_first_filter : filter;    -- (32x7x27bits)
--signal s_first_bias : biasConv;    -- (1x32x27bits)
signal s_output_conv : outConv;    -- (32x60x27bits)

-- sinais intermediários de comunicação com o componente

signal s_sample_adjusted0, s_sample_adjusted1, s_sample_adjusted2, s_sample_adjusted3, s_sample_adjusted4, s_sample_adjusted5, s_sample_adjusted6 : std_logic_vector (FP_WIDTH-1 downto 0);
signal s_filter0, s_filter1, s_filter2, s_filter3, s_filter4, s_filter5, s_filter6 : std_logic_vector (FP_WIDTH-1 downto 0);
signal s_bias, s_output : std_logic_vector (FP_WIDTH-1 downto 0):=(others => '0');
signal s_start, s_ready, done : std_logic :='0';

signal i : integer range 6 to (sampleLength-1) := sampleLength-1;
signal j : integer range 0 to (numberOfFilters-1):= numberOfFilters-1;

TYPE estados is (inicio, atualiza_entradas, reg_saida, fim);
signal estado_atual, proximo_estado : estados;

begin

    conv7samples: cnn_conv1 port map (

```

```

reset          => reset_conv,
clk            => clk,
start          => s_start,
sample_adjusted0 => s_sample_adjusted0,
sample_adjusted1 => s_sample_adjusted1,
sample_adjusted2 => s_sample_adjusted2,
sample_adjusted3 => s_sample_adjusted3,
sample_adjusted4 => s_sample_adjusted4,
sample_adjusted5 => s_sample_adjusted5,
sample_adjusted6 => s_sample_adjusted6,
filter0        => s_filter0,
filter1        => s_filter1,
filter2        => s_filter2,
filter3        => s_filter3,
filter4        => s_filter4,
filter5        => s_filter5,
filter6        => s_filter6,
bias           => s_bias,
ready          => s_ready,
output         => s_output);

```

-- PROCESSO PARA ATRIBUIÇÃO DO ESTADO ATUAL E RESET

```
state_reg: process(clk, reset_conv)
```

```
begin
```

```
    if reset_conv = '1' then
```

```
        estado_atual <= inicio;
```

```
    elsif rising_edge (clk) then
```

```
        estado_atual <= proximo_estado;
```

```
    end if;
```

```
end process;
```

-- PROCESSO PARA TRANSIÇÃO DOS ESTADOS

```
next_state_logic: process(clk, start_conv, estado_atual, s_ready, done)
```

```
begin
```

```
    if rising_edge(clk) then
```

```
        case estado_atual is
```

```
            when inicio =>
```

```
                if start_conv = '1' then
```

```

        proximo_estado <= atualiza_entradas;
    else
        proximo_estado <= inicio;
    end if;
when atualiza_entradas =>
    if done = '1' then
        proximo_estado <= fim;
    elsif s_ready = '1' then
        proximo_estado <= reg_saida;
    else
        proximo_estado <= atualiza_entradas;
    end if;
when reg_saida =>
    if done = '1' then
        proximo_estado <= fim;
    else
        proximo_estado <= atualiza_entradas;
        --proximo_estado <= reg_saida; -- apenas para testes
    end if;
when fim =>
    proximo_estado <= inicio;
when others =>
    proximo_estado <= inicio;
end case;
end if;
end process;

-- PROCESSO PARA DESCRIÇÃO DOS ESTADOS E ATRIBUIÇÃO PARA OS VALORES DE SAÍDA
output_logic: process(clk, estado_atual)

variable count : std_logic_vector(1 downto 0);

begin

count_i <= std_logic_vector(to_unsigned(i,7));
count_j <= std_logic_vector(to_unsigned(j,6));

if rising_edge(clk) then
    case estado_atual is

```

```

when inicio => -- ESTADO INICIAL QUE ZERA OS VALORES DE ENTRADA DA CONVOLUÇÃO
    s_start <= '0';
    s_sample_adjusted0 <= (others => '0'); -- (1x66x27bits)
    s_sample_adjusted1 <= (others => '0');
    s_sample_adjusted2 <= (others => '0');
    s_sample_adjusted3 <= (others => '0');
    s_sample_adjusted4 <= (others => '0');
    s_sample_adjusted5 <= (others => '0');
    s_sample_adjusted6 <= (others => '0');

    s_filter0 <= (others => '0');
    s_filter1 <= (others => '0');
    s_filter2 <= (others => '0');
    s_filter3 <= (others => '0');
    s_filter4 <= (others => '0');
    s_filter5 <= (others => '0');
    s_filter6 <= (others => '0');

    s_bias <= (others => '0');
    led_inicio <= '1';
    led_atualiza_entradas <= '0';
    led_reg_saida <= '0';
    --s_output_conv <=(others =>(others =>(others => '0')));
    output_conv <=(others =>(others =>(others => '0')));
    count := "00";
    i <= 65;
    j <= 31;
    ready_conv <= '0';
when atualiza_entradas => -- ESTADO QUE ATUALIZA AS ENTRADAS DA CONVOLUÇÃO
    s_start <= '1';

    s_sample_adjusted0 <= sample_adjust(i); -- (1x66x27bits)
    s_sample_adjusted1 <= sample_adjust(i-1);
    s_sample_adjusted2 <= sample_adjust(i-2);
    s_sample_adjusted3 <= sample_adjust(i-3);
    s_sample_adjusted4 <= sample_adjust(i-4);
    s_sample_adjusted5 <= sample_adjust(i-5);
    s_sample_adjusted6 <= sample_adjust(i-6);

```

```

--      s_filter0 <= first_filter(j,6);
--      s_filter1 <= first_filter(j,5);
--      s_filter2 <= first_filter(j,4);
--      s_filter3 <= first_filter(j,3);
--      s_filter4 <= first_filter(j,2);
--      s_filter5 <= first_filter(j,1);
--      s_filter6 <= first_filter(j,0);

--      s_bias <= first_bias(j);

s_filter0 <= firstConvFilter(j,6);
s_filter1 <= firstConvFilter(j,5);
s_filter2 <= firstConvFilter(j,4);
s_filter3 <= firstConvFilter(j,3);
s_filter4 <= firstConvFilter(j,2);
s_filter5 <= firstConvFilter(j,1);
s_filter6 <= firstConvFilter(j,0);

s_bias <= firstConvBias(j);

ready_conv <= '0';
i <= i;
j <= j;
led_inicio    <= '0';
led_atualiza_entradas <= '1';
led_reg_saida  <= '0';

when reg_saida => -- ESTADO QUE VARIA O i E j E REGISTRAS OS RESULTADOS DA CONV
    s_start <= '1';
    --s_output_conv(j,i-6) <= s_output;
    if count = "10" then
        count := "00";
        output_conv(j,i-6) <= s_output;
        if i<=6 and j>0 then
            i <= 65;
            j <= j-1;
            count := "00";

```

```

        elsif i>6 and j>=0 then
            i <= i-1;
            j <= j;
            count := "00";
        else
            i <= i;
            j <= j;
            done <= '1';
            count := "00";
        end if;
    else
        count := std_logic_vector(unsigned(count)+1);
    end if;
    ready_conv <= '0';

    led_atualiza_entradas <= '0';
    led_reg_saida <= '1'; -- apenas para teste
    led_inicio <= '0';

    --led_fim <= '0';
when fim => -- ESTADO FIM COM A CONVOLUÇÃO COMPLETA E REGISTRO COMPLETOS
    s_start <= '0';
    ready_conv <= '1';
    i <= i;
    j <= j;
    led_inicio <= '0';
    led_atualiza_entradas <= '0';
    led_reg_saida <= '0';

when others => -- outros estados
    s_start <= '0';
    ready_conv <= '0';
    i <= i;
    j <= j;
    led_inicio <= '0';
    led_atualiza_entradas <= '0';
    led_reg_saida <= '0';
end case;

```

```
counter <= count;
end if;
end process;

out_conv7    <= s_output;
--ready_conv7 <= s_ready;
ready_conv7 <= done; -- apenas para teste
```

APÊNDICE D – Código VHDL do módulo que instancia o ILA para testes em Hardware.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.fpupack.all;
use IEEE.NUMERIC_STD.ALL;

entity topmodule is
    Port ( clk : in STD_LOGIC;
          reset : in STD_LOGIC;
          first_start : in STD_LOGIC;
          ready_convolution : out STD_LOGIC;
          --output_convolution : out outConv
          init : out std_logic;
          update_in : out std_logic;
          reg_out : out std_logic;
          counter : out std_logic_vector(1 downto 0)
        );
end topmodule;

architecture Behavioral of topmodule is

    -- DECLARAÇÃO DE COMPONENTES
    component cnn_conv1_top is
        Port ( reset_conv : in STD_LOGIC;
              clk : in STD_LOGIC;
              start_conv : in std_logic;
              sample_adjust : in samples_adjust; -- (1x60x27bits)
              -- first_filter : in filter; -- (32x7x27bits)
              -- first_bias : in biasConv; -- (1x32x27bits)
              out_conv7 : out std_logic_vector(FP_WIDTH-1 downto 0);
              ready_conv7 : out std_logic;
              output_conv : out outConv; -- (32x60x27bits)
              ready_conv : out std_logic;
              led_inicio : out std_logic;
        );
    end component;

```

```

        led_atualiza_entradas      : out std_logic;
        led_reg_saida              : out std_logic;
        counter                    : out std_logic_vector(1 downto 0);
        count_i                    : out std_logic_vector(6 downto 0);
        count_j                    : out std_logic_vector(5 downto 0)
    );
end component;

COMPONENT ROM_sampleadjusted
PORT (
    clka : IN STD_LOGIC;
    ena  : IN STD_LOGIC;
    addra : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(26 DOWNTO 0)
);
END COMPONENT;

COMPONENT ila_0
PORT (
    clk : IN STD_LOGIC;

    probe0 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe1 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe2 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe3 : IN STD_LOGIC_VECTOR(26 DOWNTO 0);
    probe4 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe5 : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
    probe6 : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
    probe7 : IN STD_LOGIC_VECTOR(1 DOWNTO 0)
);
END COMPONENT ;

signal addr : std_logic_vector(6 downto 0);
signal s_sample, s_out_conv7 : std_logic_vector(FP_WIDTH-1 downto 0);
signal s_start_conv, s_ready, s_ready_conv7 : std_logic;
signal s_sample_adjust : samples_adjust;

```

```

signal s_outConv : outConv;

signal proc_sig,ready_ROM, ff1: std_logic;
signal cnt : integer range 65 downto 0;


signal s_led_inicio : std_logic;
signal s_led_atualiza_entradas : std_logic;
signal s_led_reg_saida : std_logic;


signal s_i : std_logic_vector(6 DOWNTO 0);
signal s_j : std_logic_vector(5 DOWNTO 0);
signal s_counter: std_logic_vector(1 downto 0);

begin

    myROM: ROM_sampleadjusted port map(
        clka    => clk,
        ena     => '1',
        addra   => addr,
        douta   => s_sample
    );

    uut: cnn_conv1_top port map(
        reset_conv    => reset,
        clk           => clk,
        start_conv    => s_start_conv,
        sample_adjust  => s_sample_adjust,
        out_conv7     => s_out_conv7,
        ready_conv7   => s_ready_conv7,
        output_conv   => s_outConv,
        ready_conv    => sready,
        led_inicio    => s_led_inicio,
        led_atualiza_entradas => s_led_atualiza_entradas,
        led_reg_saida => s_led_reg_saida,
        counter       => s_counter,
        count_i       => s_i,
        count_j       => s_j
    );

```

```

myILA : ila_0
PORT MAP (
    clk => clk,
    probe0(0) => s_start_conv,
    probe1(0) => s_ready_conv7,
    probe2(0) => sready,
    probe3 => s_out_conv7,
    probe4(0) => s_led_reg_saida,
    probe5 => s_i,
    probe6 => s_j,
    probe7 => s_counter
);

process(clk, reset)
--variable cnt : integer range 65 downto 0;
variable first_start_press : std_logic := '0';
begin
    if reset='1' then
        cnt <= 65;
        proc_sig <= '0';
        addr <= (others => '0');
        s_sample_adjust <= (others => (others => '0'));
        s_start_conv <= '0';
        first_start_press := '0';
    elsif rising_edge(clk) then
        proc_sig <= '0';
        if first_start='1' and first_start_press='0' then
            addr <= (others => '0');
            proc_sig <= '1';
            first_start_press := '1';
        elsif ready_ROM='1' then
            addr <= std_logic_vector(unsigned(addr) + 1);
            s_sample_adjust(cnt) <= s_sample;
            proc_sig <= '1';
            cnt <= cnt-1;
            if cnt=0 then

```

```

        cnt <= 65;
        s_start_conv <= '1';
    end if;
end if;
end if;
end process;

process(clk, reset)
begin
    if reset='1' then
        ff1 <= '0';
        ready_ROM <= '0';
    elsif rising_edge(clk) then
        if s_start_conv='0' then
            ff1 <= proc_sig;
            if ff1='1' then
                ready_ROM <= '1';
            else
                ready_ROM <= '0';
            end if;
        else
            ready_ROM <= '0';
            ff1 <= '0';
        end if;
    end if;
end process;

ready_convolution <= sready;
--output_convolution <= s_outConv;

init      <= s_led_inicio;
update_in <= s_led_atualiza_entradas;
reg_out   <= s_led_reg_saida;
counter <= s_counter;

end Behavioral;
```