UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA MECÂNICA
COMISSÃO DE PÓS-GRADUAÇÃO EM ENGENHARIA MECÂNICA

Análise de sinais de ECG com o uso de Wavelets e Redes Neurais em FPGA

Autor: Klaus Raizer
Orientador: Prof. Dr. Eurípedes Guilherme de Oliveira Nóbrega

12/2010

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA MECÂNICA
COMISSÃO DE PÓS-GRADUAÇÃO EM ENGENHARIA MECÂNICA
DEPARTAMENTO DE MECÂNICA COMPUTACIONAL

Análise de sinais de ECG com o uso de Wavelets e Redes Neurais em FPGA

Autor: Klaus Raizer
Orientador: Prof. Dr. Eurípedes Guilherme de Oliveira Nóbrega

Curso: Engenharia Mecânica
Área de Concentração: Mecânica dos Sólidos e Projeto Mecânico

Dissertação de mestrado acadêmico apresentada à comissão de Pós Graduação da Faculdade de Engenharia Mecânica, como requisito para a obtenção do título de Mestre em Engenharia Mecânica.

Campinas, 2010
S.P . - Brasil

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA E ARQUITETURA - BAE - UNICAMP

Raizer, Klaus
R137a Análise de sinais de ECG com o uso de wavelets e redes neurais em FPGA / Klaus Raizer. --Campinas, SP: [s.n.], 2010.

Orientador: Eurípedes Guilherme de Oliveira Nóbrega.
Dissertação de Mestrado - Universidade Estadual de Campinas, Faculdade de Engenharia Mecânica.

1. Eletrocardiografia. 2. Redes neurais (Computação). 3. Wavelets (Matemática). 4. VHDL (Linguagem descritiva de hardware). I. Nóbrega, Eurípedes Guilherme de Oliveira . II. Universidade Estadual de Campinas. Faculdade de Engenharia Mecânica. III. Título.

Título em Inglês: ECG signal analysis with wavelets and neural networks in FPGA
Palavras-chave em Inglês: Electrocardiography, Neural Networks (Computer science), Wavelets (Mathematics), VHDL (Hardware description language)
Área de concentração: Mecânica dos Sólidos e Projeto Mecânico
Titulação: Mestre em Engenharia Mecânica
Banca examinadora: Luiz Otávio Saraiva Ferreira, Eduardo Tavares Costa
Data da defesa: 03/02/2010
Programa de Pós Graduação: Engenharia Mecânica

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA MECÂNICA
COMISSÃO DE PÓS-GRADUAÇÃO EM ENGENHARIA MECÂNICA
DEPARTAMENTO DE MECÂNICA COMPUTACIONAL

DISSERTAÇÃO DE MESTRADO ACADÊMICO

Análise de sinais de ECG com o uso de Wavelets e Redes Neurais em FPGA

Autor: **Klaus Raizer**
Orientador: **Prof. Dr. Eurípedes Guilherme de Oliveira Nóbrega**

A Banca Examinadora composta pelos membros abaixo aprovou esta Dissertação:

Prof. Dr. Eurípedes Guilherme de Oliveira Nóbrega
Instituição: Unicamp - FEM

Prof. Dr. Luiz Otávio Saraiva Ferreira
Instituição: Unicamp – FEM

Prof. Dr. Eduardo Tavares Costa
Instituição: Unicamp - FEEC

Campinas, 3 de Fevereiro de 2010.

À minha família:
pela compreensão e apoio.

Agradecimentos

Há um grande número de pessoas às quais eu gostaria de agradecer por terem contribuído de maneira direta ou indireta para a realização deste trabalho. Agradeço em primeiro lugar à minha família e amigos pelo apoio ao longo de toda minha vida.

Gostaria de agradecer ao meu orientador Prof. Dr. Eurípedes pela oportunidade de desenvolver o presente trabalho. Agradeço também ao Professor Carlos Suzuki e à Emiko Sensei por terem me apoiado tanto durante minha vida acadêmica e com o intercâmbio para o Japão. Gostaria também de agradecer em particular aos colegas de trabalho do LNLS e a meu chefe Sérgio Rodrigo Marques que durante a curta duração de meu estágio de graduação muito me ensinaram.

Sou grato aos colegas de laboratório do DMC, não só pela inestimável ajuda com o projeto como também pelas esporádicas conversas.

Agradeço a CAPES pelo apoio com a bolsa de estudos que permitiu que eu me dedicasse ao mestrado e contribuísse, mesmo que um pouco, com o processo de desenvolvimento científico.

"Never can we sit back and wait for
miracles to save us.
Miracles don't happen; sweat happens,
effort happens, thought happens."

Isaac Asimov

Resumo

RAIZER, Klaus; *Análise de sinais de ECG com o uso de Wavelets e Redes Neurais em FPGA*. 2010. 110p. Dissertação (Mestrado) - Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, Campinas.

Este trabalho apresenta a implementação de um sistema de análise de sinais de ECGs (eletrocardiogramas) embarcado em FPGA (*field programmable gate array*), capaz de classificar cardiopatias. A análise de ECGs é de grande importância devido a sua natureza potencialmente não-invasiva, baixo custo e alta eficiência na identificação de patologias cardíacas. Visto que um sinal de ECG pode ser composto por horas de gravação da atividade cardíaca, uma abordagem computacional para a sua análise torna-se um instrumento valioso para a redução do tempo e dos erros de diagnóstico. No presente trabalho uma série de características são extraídas dos pulsos de ECG, que foram obtidos a partir dos sinais do banco de dados MIT-BIH, através da decomposição por transformada *wavelet* discreta. Essas características foram então utilizadas para treinar uma Rede Neural do tipo *feedforward* para discernir pulsos normais de pulsos anômalos. Uma versão da rede neural foi então programada em VHDL e em seguida implementada em um *Kit* da Xilinx modelo Spartan 3E para a classificação pulso a pulso dos sinais de ECG. As implicações dessa arquitetura são discutidas e os resultados são apresentados.

Palavras-chave: ECG, FPGA, redes neurais artificiais, wavelets.

Abstract

RAIZER, Klaus; *ECG Signal Analysis with Wavelets and Neural Networks in FPGA*. 2010. 110p. Dissertation (Master Degree in Mechanical Engineering): Faculty of Mechanical Engineering, State University of Campinas, Campinas.

This work presents an implementation of an embedded ECG (electrocardiogram) signal analysis system using FPGA (field programmable gate array), capable of classifying cardiopathies. The importance of ECG analysis is mainly due to its potentially non-invasive nature, low cost and high efficiency to identify heart pathologies. Since a single ECG signal can be the record of hours of heart activity, a computational approach to its analysis is invaluable to reduce diagnostic errors and the time taken by the process. In the present work, features are extracted from ECG pulses, obtained from the MIT-BIH database, by decomposing them with the Discrete Wavelet Transform. These features are then used to train a Backpropagation Neural Network in order to discriminate normal ECG pulses from anomalous ones. The neural network is programmed in VHDL and uploaded into a Spartan 3E Xilinx development kit, which performs a pulse-by-pulse classification. The implications of such architecture are discussed and its results are presented.

Keywords: ECG, FPGA, artificial neural networks, wavelets.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

1.1 Introdução

A análise de ECGs é uma técnica de grande importância devido a sua eficiência em identificar anomalias e também ao fato de ser potencialmente não-invasiva e de baixo custo. Ela é capaz de verificar várias doenças ou anormalidades como arritmias, isquemias e alterações mecânicas na estrutura do coração.

A detecção precoce de doenças ou anormalidades podem prolongar e melhorar a qualidade de vida através de tratamento mais cedo e apropriado (Silipo and Marchesi, 1998). Em certos casos, para que a análise do ECG seja efetiva é necessário que a mesma seja feita ao longo de um período de tempo que pode chegar a horas de gravação (Güler and Übeyll, 2005), como no caso do monitoramento Holter. Como o volume de informação é muito grande, sua análise leva muito tempo e se torna tediosa, o que aumenta as chances de que o responsável pela análise cometa algum erro vital. Isso sugere que uma análise utilizando recursos computacionais seja o mais indicado.

1.2 Histórico

As primeiras publicações acerca do uso de sistemas automatizados na análise de ECGs foram escritos por volta da década de 50 e 60 sendo que a partir de então vários sistemas de análise automatizada de ECGs foram desenvolvidos de forma independente por vários grupos, tanto para o sistema de 12 terminações quanto para o sistema de terminações ortogonais de Krank (Zywietz and Schneider, 1973).

Estes sistemas iniciais, no entanto, não apresentavam um desempenho satisfatório o sufici-

ente para uso em situações clínicas reais. De fato, apenas em 1959, durante uma conferência sobre métodos para processamento de dados de ECGs, um conversor analógico-digital foi demonstrado agindo em conjunto com um programa de computador voltado para o processamento destes sinais.

O primeiro programa de computador para o reconhecimento automático de ondas foi desenvolvido no início da década de 60 pelo grupo de Pipberger e se tornou referência para os desenvolvedores seguintes (Neufeld et al., 1971). Várias versões comerciais foram surgindo com o passar do tempo, muitas vezes de maneira desordenada e com pouco critério, levando a uma baixa aceitação inicial destes sistemas por parte do profissional médico. Importante ressaltar no entanto que estes sistemas iniciais sofriam de uma série de problemas vinculados justamente ao desenvolvimento pouco planejado. Na maioria dos casos a detecção das ondas não era feita de maneira sistemática, e os resultados obtidos eram baseados em escolhas geralmente arbitrárias de métodos de detecção. Uma das partes mais importantes do sistema de análise de ECGs, como será visto ao longo deste trabalho, é justamente o pré-processamento adequado do sinal. Também vital é a extração de características que formem um grupo de padrões que permitam ao sistema diferenciar as diversas características presentes no sinal. No início, estas decisões eram frequentemente feitas com base em tentativa e erro e raramente se fazia uma análise consistente dos efeitos do ruído e da tolerância ao mesmo (Neufeld et al., 1971).

No entanto, em trabalhos mais rigorosos nessa época já era comum o uso de uma variedade de filtros, em especial para a detecção do complexo QRS, parte mais central do pulso de ECG como pode ser vista na Figura 1.1.

Como será visto em mais detalhes na Seção 2.1, o pulso de ECG é composto por uma série de ondas denominadas P, Q, R, S e T, sendo que cada uma reflete uma atividade específica do coração. A sequência de ondas Q, R e S é referida como sendo o complexo QRS.

Esses filtros, projetados para a detecção do complexo QRS, já tinham a função de eliminar ruídos como os provenientes da alimentação elétrica, ou mesmo filtros passa-banda que focavam a análise do sinal nas frequências que compõem o ECG (Neufeld et al., 1971).

Técnicas comuns na época, e que ainda hoje são muito empregadas, são a análise em frequência, possibilitada pela transformada de Fourier, e a análise da derivada do sinal, para identificar alterações abruptas que indiquem as posições das ondas R.

1.3 Identificação e Classificação de Anomalias

Um diagrama que sintetiza o processo de identificação e classificação de anomalias em ECGs pode ser visto na Figura 1.2.

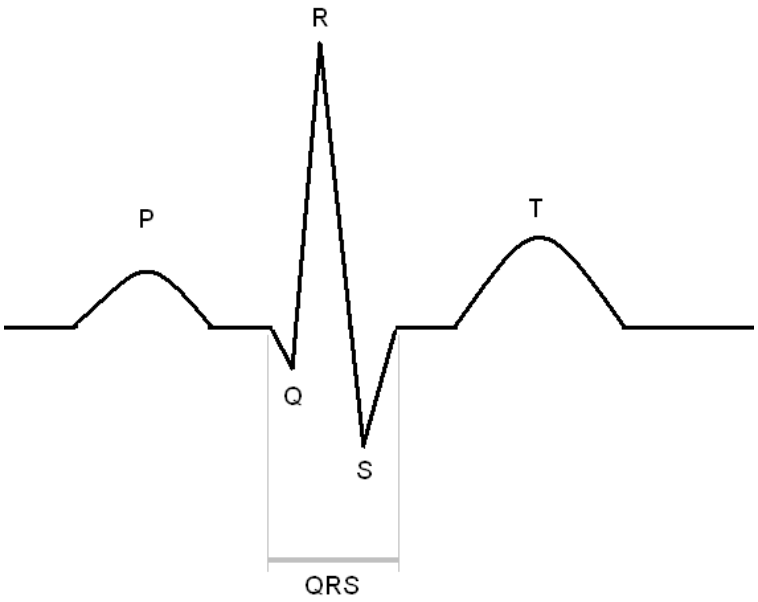

Figura 1.1: Diagrama de um pulso de ECG com destaque para o complexo QRS, como visto na derivação MLII

Cada bloco funcional de um sistema identificador de anomalias encapsula uma série de técnicas influenciadas em maior ou menor grau pelos desenvolvimentos previamente citados. Como visto na Figura 1.2, o primeiro passo do pré-processamento é a detecção do pulso de ECG seguido da extração de características. Na fase de detecção, procura-se identificar o complexo QRS focalizando na busca da onda R, devido à sua maior amplitude. De posse desta informação, é possível fazer uma série de análises tanto da morfologia das ondas quanto de suas durações.

Como ilustra a Figura 1.2, o passo seguinte é a interpretação dos dados obtidos, o que leva à fase de classificação do pulso e que pode ser feita de diversas maneiras.

O passo seguinte, ilustrado na Figura 1.2 como a classificação do pulso, é a interpretação dos dados obtidos que pode ser feita de diversas maneiras. Ao longo dos anos estas técnicas, que constituem um sistema computacional analisador de ECGs, foram aprimoradas e abordagens diferentes foram utilizadas para a implementação destas funções.

Atualmente, técnicas para a detecção dos complexos QRS apresentam desempenhos muito próximos de 100%. Como cada onda do pulso cardíaco fornece informações valiosas, muitas vezes é de grande interesse determinar os momentos em que cada onda começa e termina e, tanto a já citada técnica da análise da derivada quanto outras mais recentes, chegam a alcançar capacidades de detecção de até 99,3%, em particular graças ao uso de filtros bem projetados para o pré-processamento do sinal (Pan and Tompkins, 1985).

No final da década de 80, o uso da transformada *wavelet* discreta (DWT) tomou ímpeto

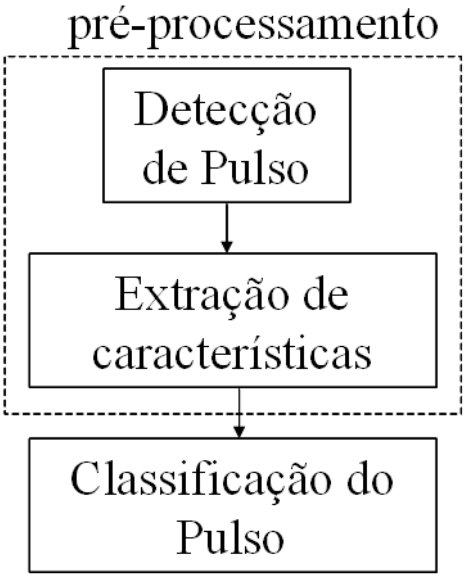

Figura 1.2: Diagrama para um sistema classificador de ECGs

tanto na detecção do complexo QRS (Vijaya et al., 1997) como na filtragem do sinal de maneira geral, e também na extração de características pertinentes para a detecção de anomalias cardíacas específicas (Saxena et al., 2002).

A escolha das técnicas para a extração de características dos sinais deve ser feita de maneira justificada. Resultados da literatura mostram que o uso da transformada *wavelet* tem produzido resultados geralmente superiores aos obtidos por métodos mais tradicionais como a análise por Fourier (Dokur et al., 1999). O uso de *wavelets* para a extração de características dos sinais de ECG se mostrou uma das técnicas mais efetivas para o processamento dos sinais.

Há várias técnicas que produzem resultados satisfatórios, como *clustering*, lógica nebulosa, sistemas lineares, redes neurais ou mesmo técnicas híbridas que utilizam várias abordagens simultaneamente, visando obter um resultado superior àquele obtido pelo uso de apenas uma.

1.4 Desenvolvimento em *Hardware*

Aliada à evolução das técnicas de tratamento e análise dos sinais ocorreu também um desenvolvimento tecnológico impressionante com o passar das décadas. O nível de miniaturização dos componentes eletrônicos permitiu a construção de sistemas completos muito menores do que os desenvolvidos há algumas décadas, viabilizando assim o desenvolvimento de sistemas complexos, porém portáteis.

Essa evolução no *hardware* influenciou também os sistemas de análise de ECGs permitindo uma maior robustez, capacidade e portabilidade dos dispositivos desenvolvidos.

1.5 Objetivos do Trabalho

O presente trabalho faz parte de um projeto maior sendo desenvolvido no Departamento de Mecânica Computacional da Faculdade de Engenharia Mecânica, que almeja a construção de um sistema de análise e interpretação de ECG usando FPGA.

Os sinais de ECG serão lidos por um conversor AD e pré-processados pelo microcontrolador MicroBlaze™que também gerencia os acessos à memória e aos componentes codificados em VHDL no *hardware*. O MicroBlaze™é uma propriedade intelectual da Xilinx, pode ser codificado no próprio *hardware* do FPGA e sua programação é feita em C. Um diagrama simplificado do sistema, implementado no *hardware* da Xilinx, está representado na Figura 1.3.

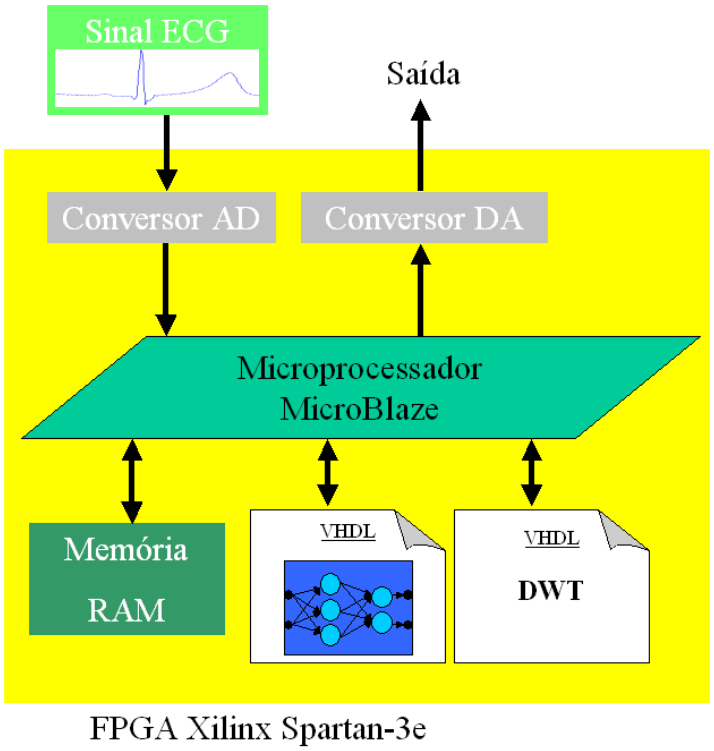

Figura 1.3: Diagrama do projeto final: em desenvolvimento no Departamento de Mecânica Computacional da Unicamp

A detecção de pulsos utiliza a transformada *wavelet* discreta, como visto no diagrama, para a detecção dos complexos QRS em *hardware*.

O foco do presente trabalho é então o desenvolvimento do sistema de classificação dos pulsos, que será capaz de identificar anomalias específicas. Devido ao sucesso das redes neurais na classificação de sinais cardíacos observado na literatura (Yu and Chen, 2007), decidiu-se adotar uma rede neural do tipo MLP (*Multilayer Perceptron*), que possui como entradas características extraídas dos pulsos de ECG através da transformada *wavelet* discreta (Güler and Übeyll, 2005).

De modo a integrar o presente trabalho no projeto final, buscou-se manter a similaridade da estrutura de fluxo de dados do projeto, como pode ser visto na Figura 1.4. O pré-processamento é feito em uma CPU externa e enviado para o dispositivo através da camada de comunicação, que pode ser uma conexão USB ou serial. Os dados pré-processados são então avaliados pela rede neural implementada em *hardware*, que por sua vez identifica o pulso analisado como sendo anômalo ou normal, atribuindo a cada pulso valores de saida que o classificam dentro de um dos grupos de anomalias para as quais a rede foi treinada.

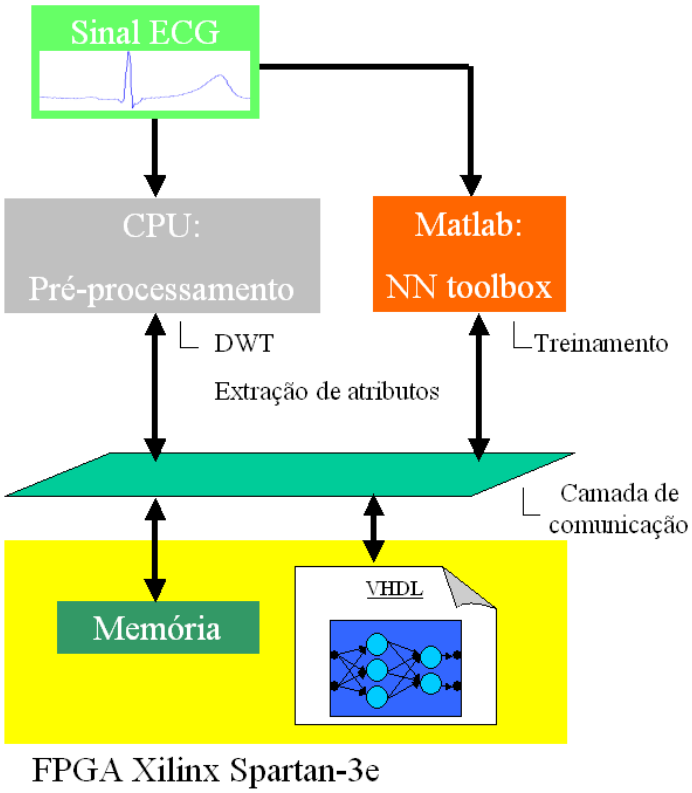

Figura 1.4: Diagrama do trabalho desenvolvido

Conforme representado na Figura 1.4, o treinamento da rede neural artificial foi feito utilizando-se o *toolbox* de redes neurais do Matlab, como será descrito em mais detalhes na Seção 3.2. Um aspecto significativo do presente trabalho é o uso de um metaprograma

desenvolvido em Matlab, que facilita implementar a rede neural em VHDL. Isto viabiliza testes e simulações em *hardware* de novas topologias de rede ou técnicas de treinamento de maneira imediata, como será visto na Seção 3.3.

1.6 Corpo do Texto

A estrutura do texto está organizada da seguinte maneira:

- **Introdução:** Apresenta a motivação que resultou nos objetivos e na arquitetura adotada no trabalho e a organização do texto..
- **Fundamentos técnicos:**
  - **O eletrocardiograma:** Disserta-se sobre as características biológicas do batimento cardíaco, como ele funciona e como tal funcionamento se traduz em impulsos elétricos mensuráveis que são então registrados na forma do ECG.
  - **Redes Neurais:** É feita uma introdução resumida sobre o funcionamento e aplicações de redes neurais artificiais. Disserta-se sobre seu treinamento e sobre a importância de um pré-processamento adequado dos dados, com um breve comentário sobre os efeitos do sobre-treinamento.
  - **Wavelets:** É feita uma explicação breve sobre a transformada wavelet discreta e de seu papel na análise de sinais transientes, como é o caso do ECG.
  - **FPGA:** Explicação sobre o funcionamento dos FPGAs (*field programmable gate arrays*) com ênfase no *hardware* da Xilinx, modelo Spartan 3E, utilizado neste trabalho.
- **Desenvolvimento dos Métodos e Implementação:** Descrição detalhada do método de identificação de pulsos, extração de características relevantes, identificação e classificação de anomalias com redes neurais e a implementação do sistema em FPGA.
- **Resultados:** Apresenta os resultados obtidos com a implementação do método de identificação de anomalias descrito na seção anterior. Relata também o produto da implementação em FPGA do sistema de análise de ECG, os problemas encontrados e as soluções desenvolvidas.
- **Considerações Finais:** Conclusões finais sobre o desenvolvimento do projeto e perspectivas futuras.

Os programas dos componentes VHDL estão disponíveis nos apêndices.

Capítulo 2

Fundamentos Técnicos e Científicos

2.1 O eletrocardiograma

O eletrocardiograma foi inventado há mais de 100 anos pelo fisiologista Willem Einthoven e, mesmo com o advento de inúmeras outras técnicas de diagnóstico, continua sendo de grande importância na avaliação de anomalias cardíacas e de arritmias.

2.1.1 Técnica

De maneira geral, o ECG é um registro da atividade elétrica do coração e é apresentado na forma de um gráfico cujas variações representam as correntes elétricas percorrendo os tecidos do coração ao longo do tempo.

As medições são feitas através de eletrodos fixados em pontos específicos do corpo com o intuito de detectar a diferença de potencial elétrico entre os mesmos, como exemplificado na Figura 2.1.

São utilizados até 10 eletrodos, ou derivações, posicionados de acordo com a Figura 2.1 para se obter 12 vistas da atividade elétrica do coração (Khan, 2007).

Como visto na figura, há dois grupos de terminações: eletrodos peitorais e eletrodos periféricos. Os seis eletrodos peitorais são nomeados de V1 até V6 e são posicionados em locais específicos ao longo da caixa torácica.

As quatro terminações periféricas são RA (fixa no braço direito), LA (fixa no braço esquerdo), LL (fixa na perna esquerda) e RL (fixa na perna direita).

Devido ao transporte ativo dos íons Na+ e K+ através da membrana semipermeável, uma célula cardíaca em repouso possui diferentes concentrações destes íons em seu interior.

Os íons de sódio são conduzidos para fora do corpo da célula enquanto que os íons de potássio são levados para dentro. Entretanto, como a membrana celular é permeável ao K+,

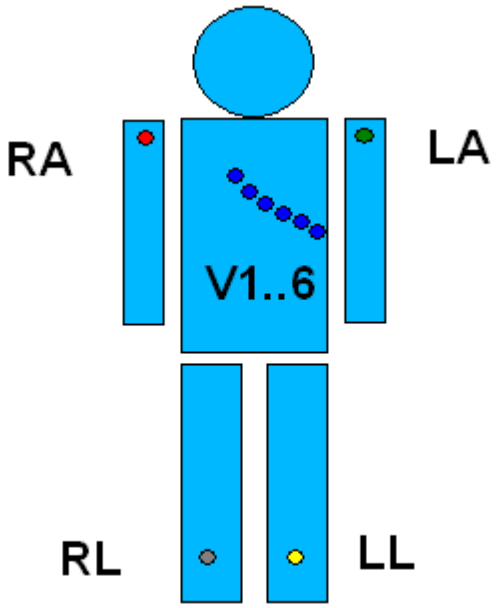

Figura 2.1: Posicionamento dos eletrodos, para a técnica *12-leads*

este tende a sair da célula deixando o interior da mesma eletricamente negativo. Isto cria uma diferença de potencial entre o interior e exterior da célula, da ordem de -90mV, ao que se chama *potencial de repouso*, estado no qual a célula é considerada polarizada.

Ao receber um estímulo elétrico, as células cardíacas passam ao estado despolarizado, o que leva à contração do coração. Um corte que mostra a estrutura interna do coração humano pode ser visto na Figura 2.2.

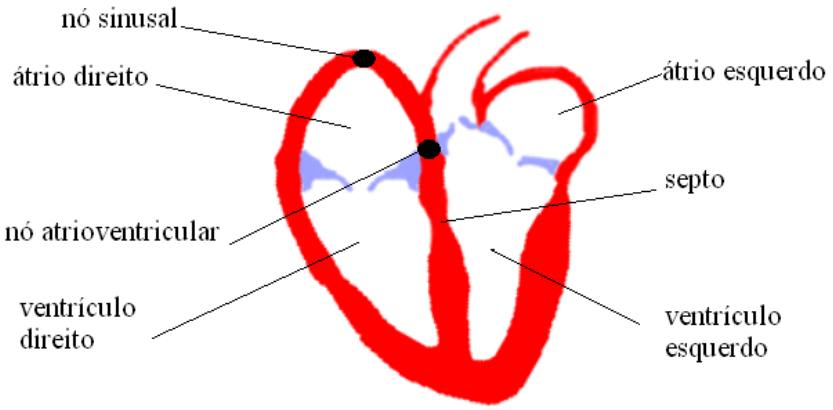

Figura 2.2: Esquema do coração humano com suas áreas mais importantes

O estímulo elétrico parte do nó sinusal que se localiza na região superior do átrio direito

e se propaga para o átrio esquerdo logo em seguida. É essa atividade que dá origem à onda P do ECG que pode ser vista na Figura 2.3.

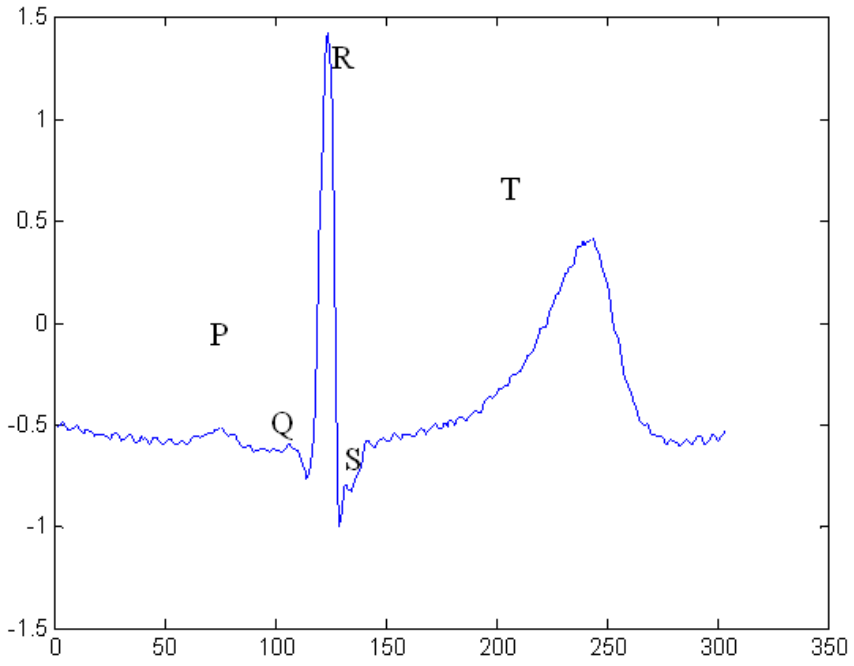

Figura 2.3: Pulso completo de um ECG

O impulso ultrapassa então o nó atrioventricular estimulando ambos os ventrículos. A primeira parte a ser estimulada é o septo, o que dá origem à onda negativa Q do ECG. Ocorre logo em seguida o estímulo das paredes dos ventrículos dando origem à onda R, sendo a influência do ventrículo esquerdo preponderante devido a sua maior massa em relação ao ventrículo direito.

A onda S é finalmente produzida pela ativação da parte mais inferior do coração visto na Figura 2.2. Em seguida ocorre a repolarização dos ventrículos, que é registrada no ECG na forma da onda T.

O pulso completo do ECG é composto então pela onda P (fruto da despolarização dos átrios), complexo QRS (que registra a despolarização dos ventrículos) e finalmente a onda T (resultado da repolarização ventricular).

Entretanto, mesmo para indivíduos normais, o ECG apresenta variações devido a inúmeros fatores como idade, variações de exame para exame, biótipos diferentes e etc. Determinadas variações, no entanto, indicam anomalias como sobrecargas das câmaras cardíacas, bloqueios de ramo, infarto, arritmias e outras cardiopatias (Khan, 2007).

2.1.2 Cardiopatias

Para este trabalho foram escolhidas 3 cardiopatias, com amostras obtidas no banco de dados MIT Physionet (Goldberger et al., e 13), as quais, junto ao caso normal, formam um problema de quatro classes distintas.

As cardiopatias adotadas foram o bloqueio de ramo esquerdo, bloqueio de ramo direito e contração ventricular prematura. A escolha das cardiopatias foi baseada nos trabalhos vistos na literatura (Ilic, 2007)(Song and Lee, 2005). A presença de um pulso normal ou com determinada cardiopatia é identificada por siglas como se segue:

- N - pulso normal
- V - contração ventricular prematura (PVC)
- R - bloqueio de ramo direito (RBBB)
- L - bloqueio de ramo esquerdo (LBBB)

Em batimentos cardíacos normais, como os vistos na Figura 2.4 nas quais o caso normal é identificado por pontos, ambos os ventrículos devem se contrair ao mesmo tempo, a não ocorrência desse processo é um forte indício de bloqueio de ramo esquerdo ou de ramo direito.

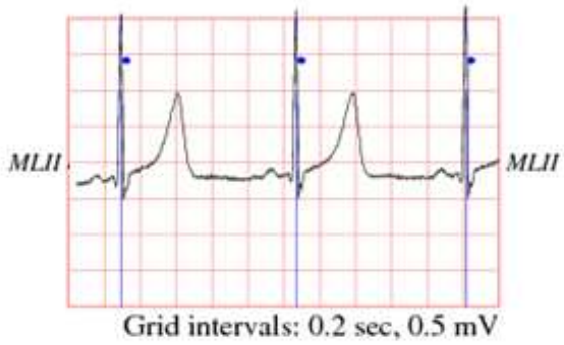

Figura 2.4: Exemplos de pulsos normais de ECG

O bloqueio de ramo esquerdo (*left bundle branch block*) é uma cardiopatia caracterizada pela ativação atrasada do ventrículo esquerdo. Um sinal exemplificando este tipo de cardiopatia pode ser visto na Figura 2.5. Identificar corretamente o bloqueio de ramo esquerdo é muito importante pois a sua presença dificulta a interpretação do segmento S-T.

O bloqueio de ramo direito é caracterizado pela não ativação do ventrículo direito de maneira direta, ou seja, os impulsos do ramo direito não ativam o ventrículo direito ao

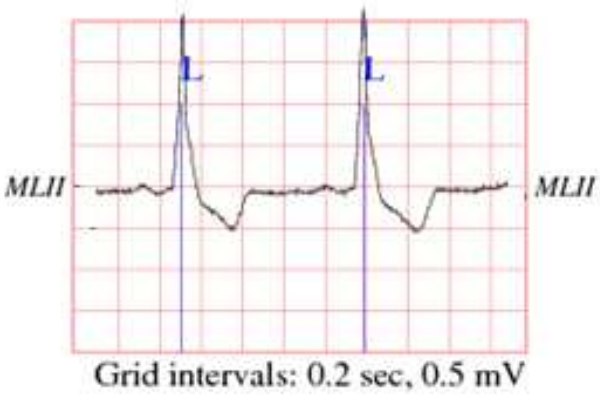

Figura 2.5: Exemplos de pulsos de ECG afetados por bloqueio de ramo esquerdo

mesmo tempo em que o ventrículo esquerdo é ativado pelo ramo esquerdo. O reflexo desse tipo de anomalia pode ser visto na Figura 2.6.

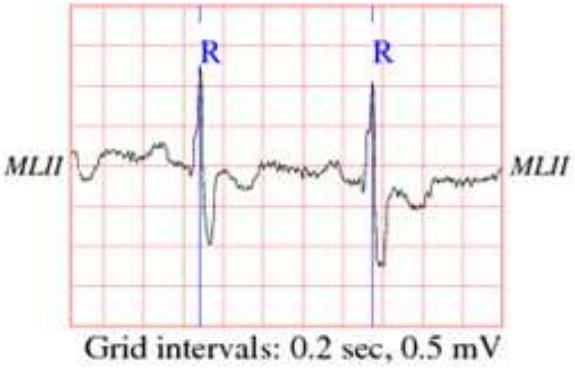

Figura 2.6: Exemplos de pulsos de ECG afetados por bloqueio de ramo direito

Como o ventrículo esquerdo continua sendo ativado normalmente, o impulso que por ele passa atravessa o miocárdio e chega no ventrículo direito gerando uma ativação atrasada do mesmo.

Uma contração ventricular prematura (*premature ventricular contraction*) tem natureza diferente das anomalias de bloqueio de ramo. No caso de uma PVC, os ventrículos são ambos ativados prematuramente gerando uma ineficiência no processo de bombeamento do sangue ao longo do corpo. Alguns pulsos acusando PVC podem ser vistos na Figura 2.7.

O paciente geralmente sente o fenômeno como batimentos faltantes, dores no peito, fadiga e pode ocorrer devido a vários motivos. Pode indicar casos de isquemia, cardiomiopatia, hipercapnia entre outros.

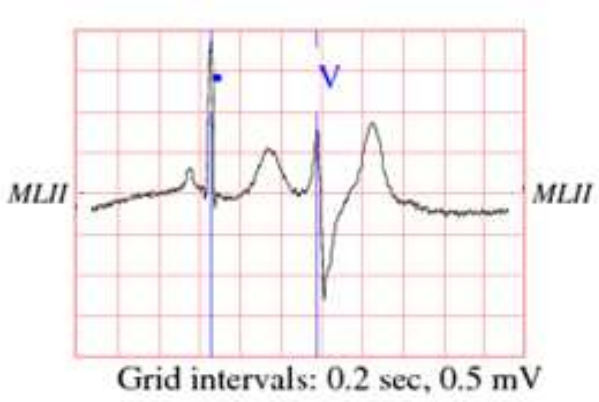

Figura 2.7: Exemplos de pulsos de ECG afetados por contração ventricular prematura

2.2 Redes Neurais

2.2.1 Introdução

A motivação inicial para o estudo das redes neurais veio da constatação de que o cérebro humano processa informação de uma maneira totalmente diferente da que um computador o faz. O cérebro é capaz de resolver problemas similares aos que podem ser resolvidos por um computador extremamente complexo, não-linear e de processamento paralelo, sendo capaz de realizar determinados processamentos várias ordens de grandeza mais rápido do que os mais rápidos computadores digitais (Haykin, 1998).

A finalidade de se usar redes neurais artificiais então é explorar essa considerável capacidade de processamento quando tarefas como reconhecimento de padrões ou mapeamentos complexos precisem ser realizadas.

O funcionamento das redes neurais artificiais e seus métodos de aplicação serão explicados nas seções seguintes.

2.2.2 Redes Neurais Artificiais

Redes neurais artificiais são compostas por elementos relativamente simples que, trabalhando em paralelo como uma rede, são capazes de realizar aproximações, cálculos, reconhecimento de padrões, controle de sistemas e classificações complexas. Assim, o objetivo desta seção não é apresentar sistemas neurais com rigor neurocientífico, mas sim comentar a inspiração obtida no campo da pesquisa neurológica e explanar sobre a aplicação destes conceitos na área de classificação de padrões.

Cada elemento da rede tem o nome de neurônio, e sua estrutura é baseada no funciona-

mento de um neurônio biológico como o da Figura 2.8.

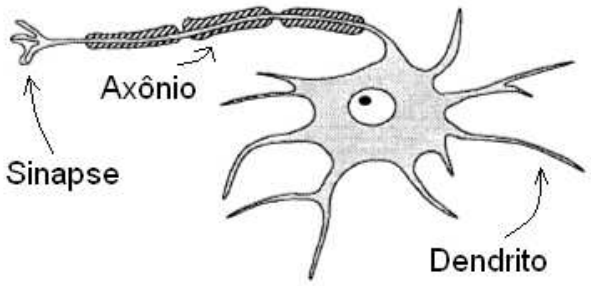

Figura 2.8: Neurônio biológico

Em um neurônio biológico, as conexões entre um neurônio e outro ocorrem em vários pontos da célula, em estruturas chamadas sinapses. Impulsos neurais através destas conexões podem resultar em mudanças locais no potencial elétrico do corpo da célula receptora. Estes potenciais de entrada podem percorrer o corpo principal da célula, podendo ser tanto excitatórios (reduzindo a polarização da célula) quanto inibitórios (aumentando a polarização da célula). Os potenciais de entrada são somados na conexão do corpo da célula com o axônio (*axon hillock*). Se o total de despolarização for suficiente é gerado um potencial de ativação, que caminha ao longo do axônio (Freeman and Skapura, 1991).

A analogia do neurônio artificial a partir do neurônio biológico vai além da estrutura básica de um único neurônio. O sistema nervoso animal é composto por milhões de células nervosas interconectadas. Embora a resposta de cada célula seja muito mais lenta que aquela obtida com componentes eletrônicos, o cérebro é capaz de resolver problemas de complexidade muito superior que os atuais computadores. Isso se deve ao processamento paralelo efetuado por cada neurônio quando o mesmo faz parte de uma rede. De maneira análoga é possível então determinar um modelo matemático que emule o comportamento do neurônio biológico e que, estruturado em camadas, consiga mimetizar o processamento paralelo observado nas redes neurais biológicas. O modelo de neurônio mais utilizado pode ser visto na Figura 2.9.

O neurônio artificial, como visto na Figura 2.9, é constituido então por uma soma ponderada de suas entradas com a aplicação de uma função de ativação antes de produzir uma saída. Essa ponderação é feita pelos pesos em cada conexão, chamados de matriz de ponderação, quando vários neurônios são organizados em uma camada. O bias é uma entrada fixa em 1 com seu próprio peso. Seu objetivo é permitir a translação da função de ativação, vista a seguir, conferindo ao neurônio um grau de liberdade extra que independa das entradas.

Após o somatório das entradas ponderadas, o resultado passa pela função de ativação.

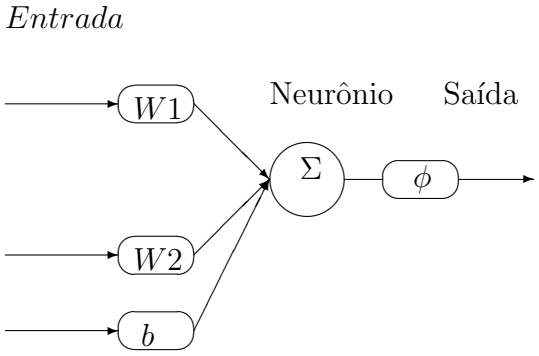

Figura 2.9: Modelo de um neurônio artificial

Essa função de ativação pode ser de vários tipos, como visto na Figura 2.10.

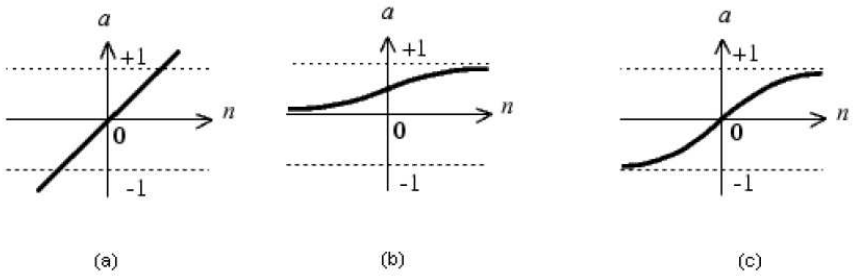

Figura 2.10: Funções de ativação Linear (a), Sigmóide (b) e Tangente Hiperbólica (c)

As funções mais utilizadas são a sigmoide, ou a tangente hiperbólica, sendo a escolha geralmente determinada pelo problema a ser resolvido.

Fica então clara a notação matricial que liga as saídas da primeira camada às entradas da seguinte. A matriz de pesos pode ser escrita como na Equação 2.1.

$$W = \begin{bmatrix} W_{1,1} & W_{1,2} & \cdots & W_{1,nInputs} \\ W_{2,1} & W_{2,2} & \cdots & W_{2,nInputs} \\ \vdots & \vdots & \ddots & \vdots \\ W_{nNeurons,1} & W_{nNeurons,2} & \cdots & W_{nNeurons,nInputs} \end{bmatrix} \tag{2.1}$$

A saída de uma camada pode então ser escrita como na Equação 2.2.

$$s = \phi(W \cdot i + b) \tag{2.2}$$

Onde ϕ é a função de ativação, i o vetor de entrada da camada, b o bias e W a matriz

de pesos.

Os índices das linhas da matriz de ponderação indicam o neurônio de destino para aquele peso específico, enquanto que o índice da coluna indica o neurônio de origem. Quando várias camadas de neurônios são ligadas em série, como na Figura 2.11, tem-se então uma rede chamada perceptron de múltiplas camadas, ou MLP (*Multilayer Perceptron*). Nesta rede, todas as saídas de uma camada estão ligadas a todas as entradas da camada seguinte.

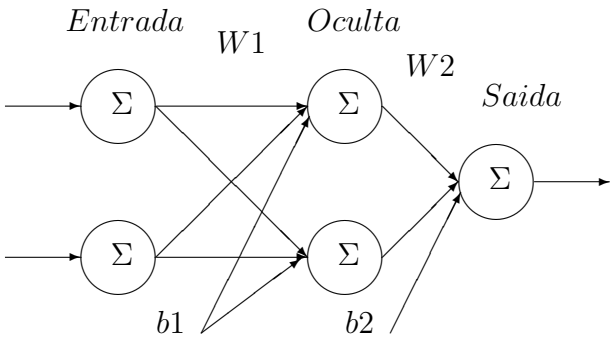

Figura 2.11: MLP - neurônios organizados em camadas

Treinamento

O objetivo do processo de treinamento é minimizar uma função de erro, definida pelos pesos da rede e suas entradas. Um exemplo para o caso com apenas dois pesos pode ser visto na Figura 2.12.

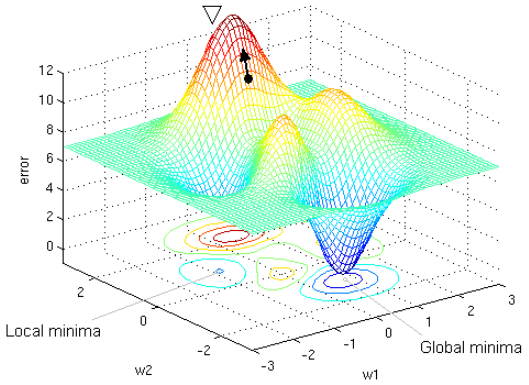

Figura 2.12: Função de erro para dois pesos.

O erro a ser reduzido pode ser obtido a partir das Equações 2.3 e 2.4, sendo N o número total de neurônios de saída (Haykin, 1998).

$$erro = SaidaEsperada - SaidaProduzida \tag{2.3}$$

$$E(n) = \frac{1}{2} \cdot \sum_{k=1}^{N} erro_k(n)^2 \tag{2.4}$$

Os pesos que formam a rede precisam ser corrigidos para que o erro da Equação 2.4, que representa a soma instantânea dos quadrados dos erros das saídas, se torne o menor possível. Há várias maneiras de se alcançar tal objetivo, indo do tradicional método do gradiente, que faz uso do algoritmo de retropropagação (*backpropagation*) e segue o caminho inverso apontado pelo gradiente da função de erro no ponto, até algoritmos evolutivos. De maneira geral, a correção dos pesos é feita de acordo com a Equação 2.5 (Haykin, 1998).

$$W(n+1) = W(n) + \eta \cdot \Delta W(n) \tag{2.5}$$

Sendo η a taxa de aprendizado e $\Delta W(n)$ a variação nos pesos atuais.

Pré-processamento

Para que a rede neural seja capaz de tratar os problemas reais que lhe são apresentados, os dados de entrada precisam ser pré-processados para garantir que se encontram em um formato adequado.

Neste trabalho utilizou-se a normalização dos dados de entrada de acordo com a Equação 2.6, sendo X o vetor de entradas, μ a média e σ o desvio padrão do vetor.

$$X_{norm} = \frac{X - \mu}{\sigma} \tag{2.6}$$

A normalização dos dados é especialmente importante para valores que variam em escalas diferentes. De maneira geral, sem a normalização a própria rede neural tende a compensar pela discrepância entre os valores de entrada, o que acaba gerando pesos grandes e também levando a uma saturação dos neurônios.

Outro ponto importante durante a fase de pré-processamento é a separação dos valores de entrada e saída em dois grupos: grupo de treino e grupo de validação.

O grupo de validação não participa do processo de treinamento, sendo utilizado apenas para checar a capacidade de generalização da rede. Ele é então o mais indicado para servir como critério de parada e sua aplicação ficará clara nas seções seguintes.

Efeitos do sobre-treinamento em redes MLP

O sobre-treinamento é resultado de uma adaptação excessiva da rede aos dados de treinamento. Ocorre então o chamado *overfitting*, sendo que a rede em questão passa a modelar não apenas os dados mas até mesmo o ruído presente neles.

Do ponto de vista estatístico, isto ocorre quando o modelo possui graus de liberdade em excesso em comparação aos dados sendo aproximados. Isto resulta em uma capacidade de generalização reduzida e a rede se torna incapaz de ter um desempenho satisfatório em casos reais.

2.3 Wavelets

2.3.1 A importância do transitório

Fenômenos no mundo real são frequentemente de natureza transitória. A análise de sinais reais precisa levar em consideração que em muitos casos o fenômeno, cuja representação por ventura esteja encerrada dentro do sinal, pode estar representado de maneira efêmera e extremamente localizada no tempo. Das oscilações do mercado de ações às batidas do coração humano medidas em um eletrocardiograma, a detecção dos fenômenos transitórios constitui um desafio às tradicionais técnicas de análise de sinais estacionários.

Um exemplo de técnica de análise de sinais muito utilizada tanto no meio científico quanto na engenharia é a transformada de Fourier. A transformada de Fourier baseia-se na composição espectral de um determinado sinal a partir da soma de ondas senoidais de formato $e^{i\omega t}$. A transformada de Fourier é uma ferramenta extremamente útil para a análise de fenômenos periódicos e estacionários. Entretanto, para casos em que o fenômeno a ser analisado esteja localizado em um intervalo curto do sinal (como a queda abrupta do valor de uma ação ou o aumento localizado no consumo de energia elétrica medida por uma empresa fornecedora), a transformada de Fourier tradicional deixa de fornecer resultados tão satisfatórios.

2.3.2 Transformada de Fourier janelada

Uma maneira de se contornar essa limitação foi trabalhada por Gabor em 1946(Gabor, 1946). Em seu trabalho, Gabor utilizou uma janela g transladada no tempo e na frequência como na equação 2.7 para aplicar a metodologia da transformada de Fourier.

$$g_{u,\Psi} = e^{i\Psi t} \cdot g(t - \tau) \tag{2.7}$$

Essa técnica é também conhecida como *windowed Fourier transform*, ou *short time Fourier transform*. O nome *short time Fourier transform* (STFT) vem do fato de que a multiplicação por g(t-τ) localiza a integral de Fourier nas vizinhanças de t=τ. Gabor adotou uma janela gaussiana em seu artigo original, porém várias outras funções de janelamento podem ser utilizadas, como a retangular, de Haar, etc. A STFT é especialmente eficaz em discernir variações de frequência ao longo do tempo tendo um desempenho muito bom em aplicações como reconhecimento de voz (Mallat, 1999).

2.3.3 Transformada *wavelet* contínua

Da mesma forma que a transformada de Fourier opera através da composição do sinal com senóides, a transformada *wavelet* é o resultado da composição do sinal com uma *wavelet*, também conhecida como *wavelet*-mãe. Uma *wavelet* é uma função de média zero, que obedece à Equação 2.8.

$$\int_{-\infty}^{\infty} \Psi(t)dt = 0 \tag{2.8}$$

Alguns dos exemplos de funções *wavelets* mais utilizados podem ser vistos na Figura 2.13. A *wavelet* do tipo Haar é a mais simples e também é conhecida como daubechie 1 (db1) (Higham and Higham, 2005). A família das Daubechies é de extrema importância na análise de sinais discretos.

A *wavelet* é então dilatada por um fator s e transladada de τ de acordo com a Equação 2.9.

$$\Psi_{u,s}(t) = \frac{1}{\sqrt{s}}\Psi\left(\frac{t-\tau}{s}\right)dt \tag{2.9}$$

De fato, a transformada *wavelet* consegue medir variações tempo-frequência como a já mencionada técnica da transformada de Fourier janelada. Entretanto, a técnica da janela móvel sofre de uma limitação séria; uma vez escolhido o tamanho da janela, esse tamanho é fixo para todas as frequências (Higham and Higham, 2005), o que determina a sua evolução tanto no tempo quanto na frequência.

A vantagem da transformada *wavelet* é então a sua capacidade de análise em múltiplas escalas, que permite avaliar os sinais com resoluções adequadas para diferentes frequências ao longo do tempo.

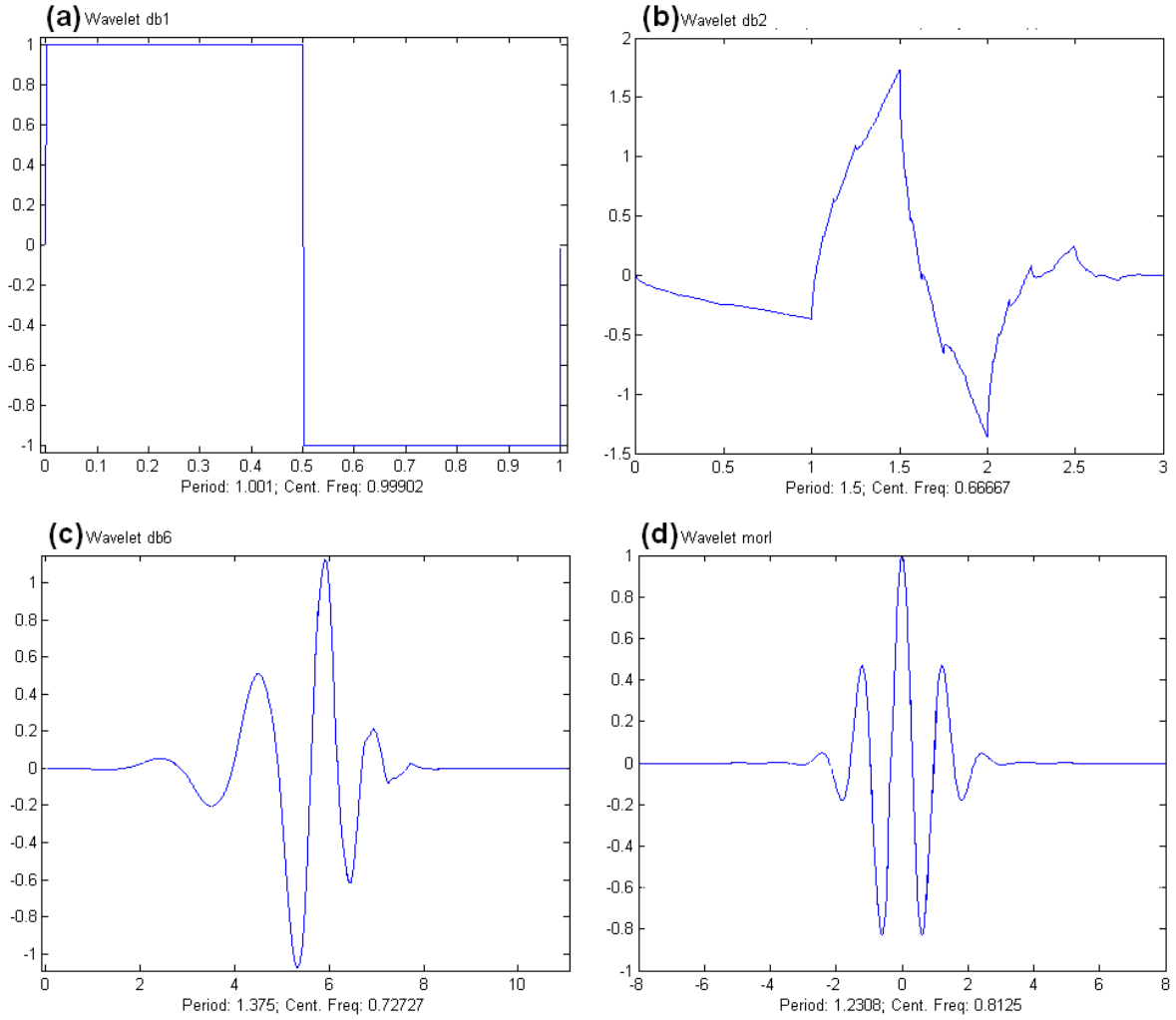

Figura 2.13: Exemplos de funções *wavelet*: (a)*wavelet* tipo Haar,(b)família Daubechie tipo 2, (c) Daubechie 6, (d)*wavelet* Morlet

2.3.4 Transformada *wavelet* discreta

Embora a transformada *wavelet* tenha amplas aplicações para o caso contínuo ela normalmente é usada em aplicações em tempo discreto, no processamento digital de sinais. Um método muito eficiente para implementar a transformada *wavelet* discreta(DWT) foi proposto por Mallat em 1988 (Mallat et al., 1989). De fato, de maneira análoga à transformada rápida de Fourier , é possível implementar uma transformada rápida *wavelet* (FWT) por meio de um banco de filtros. Uma ilustração dessa aplicação pode ser vista na Figura 2.14. O sinal original **X** é submetido a dois filtros de naturezas diferentes; o filtro **H** é do tipo passa alta enquanto que o filtro **L** é do tipo passa baixa. Os sinais filtrados são sub-amostrados, mantendo-se apenas uma amostra a cada duas. O resultado da filtragem com **H** é chamado

de sinal de *detalhamento* enquanto que o que se produz com a filtragem por **L** é chamado de *aproximação*.

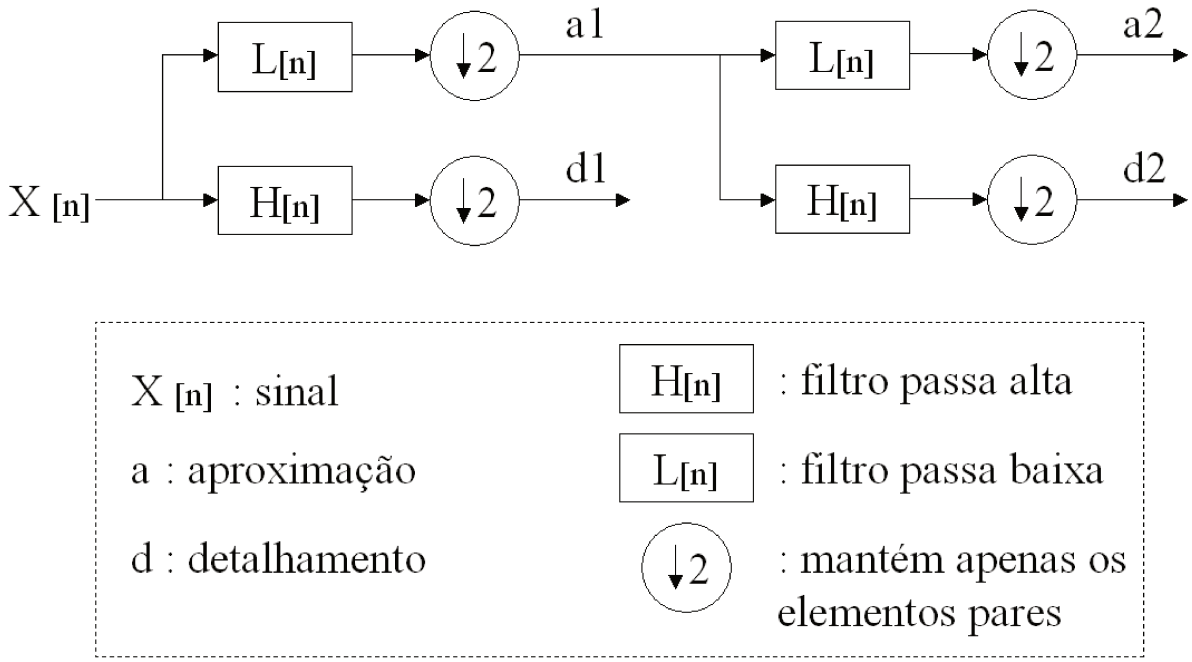

Figura 2.14: Decomposição *wavelet* em detalhes e aproximações

Encadeando-se os filtros em sequência, com nova decomposição do sinal de aproximação em dois outros sinais, são produzidos sinais com maior resolução e menor número de elementos. Essa metodologia permite então a aplicação direta da análise *wavelet* com o uso de filtros digitais.

2.4 FPGA

2.4.1 Introdução

O FPGA (*field programmable gate array*) é um Dispositivo Lógico Programável, ou PLD (*Programmable Logic Device*), que contém um *array* bidimensional de células lógicas genéricas e chaves programáveis com estrutura similar à que pode ser vista na Figura 2.15.

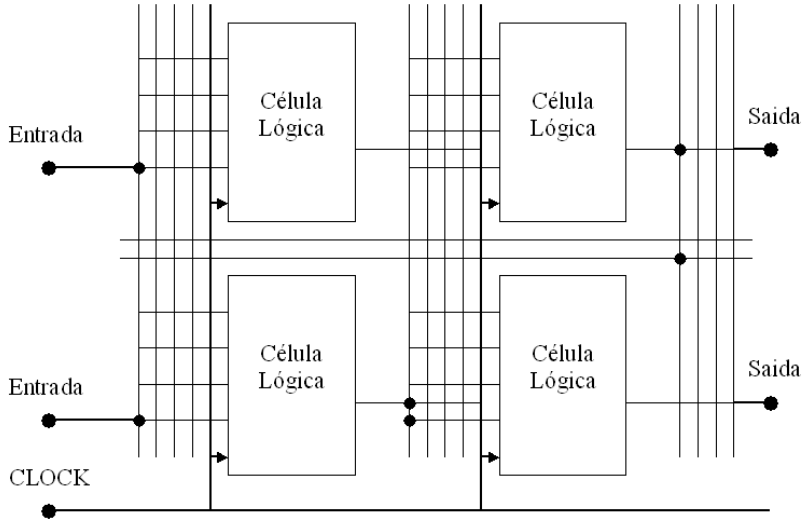

Figura 2.15: Array de células programáveis

Em meados da década de 70 a idéia de se produzir circuitos lógicos que fossem programáveis deu origem aos PLDs. O objetivo não era o processamento de um programa em *software*, como no caso de microcontroladores, mas sim desenvolver a capacidade de se programar no nível de *hardware*.

Os primeiros PLDs eram compostos apenas por portas lógicas. Desprovidos de flip-flops, podiam implementar circuitos combinacionais, mas eram incapazes de implementar circuitos sequenciais.

Com o passar dos anos houve uma série de desenvolvimentos que aprimoraram a tecnologia dos PLDs, culminando na criação dos CPLDs (*Complex PLDs*), que possuíam alta densidade, alta performance e baixo custo, considerando a tecnologia da época) (Pedroni, 2004).

Na década de 80 os dispositivos FPGA foram introduzidos pela Xilinx. Eles diferiam dos CPLDs em arquitetura, tecnologia de armazenamento, número de características incorporadas e custo, sendo voltados à implementação de circuitos grandes e de alto desempenho (Pedroni, 2004).

Existem dois tipos de FPGAs sendo que o mais comum é reprogramável e baseado em SRAM (do tipo volátil) enquanto que o outro é do tipo OTP (*One Time Programmable*), baseado em tecnologia *antifuse* (Xilinx and Mehta, 2008). O utilizado neste trabalho, é o baseado na tecnologia SRAM, e portanto reprogramável, e a estrutura de suas células lógicas é essencialmente como vista na Figura 2.16.

A lógica desejada para um projeto pode ser definida como uma série de modificações nas ligações entre essas células individuais, de modo que uma vez que a síntese do circuito esteja pronta, basta carregar o FPGA com o projeto.

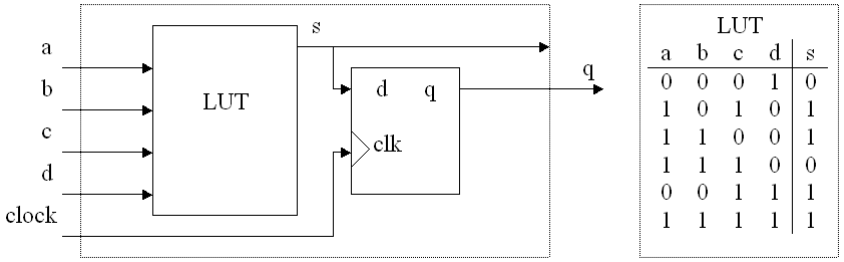

Figura 2.16: Célula Lógica programável com LUT de 4 entradas

Na Figura 2.16 pode ser visto que a célula é formada basicamente por uma LUT (*lookup table*) e um flip-flop tipo D. A LUT é utilizada para implementar qualquer função combinacional e o seu resultado pode ser utilizado diretamente ou fornecido ao flip-flop tipo D. O objetivo do flip-flop é assim tornar possível a programação de circuitos sequenciais.

Importante ressaltar que, para sistemas desenvolvidos em *hardware*, há mais de um tipo de velocidade que deve ser levado em consideração: taxa de transferência (*throughput*), latência e tempo sequencial (*delay*).

A taxa de transferência se refere à quantidade de dados processados por unidade de tempo. A latência é uma medida comparativa entre o tempo que leva para os dados de entrada serem totalmente processados e fornecidos como saída. Já o *delay* se refere ao tempo que a lógica leva para ser concluída em cada elemento do circuito.

Tendo isso em mente, pode-se concluir que quebrar operações em sub-operações que possam ser processadas em paralelo leva a um aumento da velocidade total de processamento, pois o tempo total que o circuito requer passa a ser o máximo *delay* dos sub-componentes produzidos, ao invés da soma dos *delays* de um circuito sequencial.

Em outras palavras, se uma determinada lógica é composta por três componentes: A, B e C. E cada componente possui um tempo mínimo para ser processado (seus *delays*) os quais serão chamados de dA, dB e dC. Os diagramas que diferenciam a implementação sequencial da paralela podem ser vistos na Figura 2.17, na qual os blocos A, B e C simbolizam os

elementos de mesmo nome, e as setas simbolizam o tempo que cada elemento leva para ser processado.

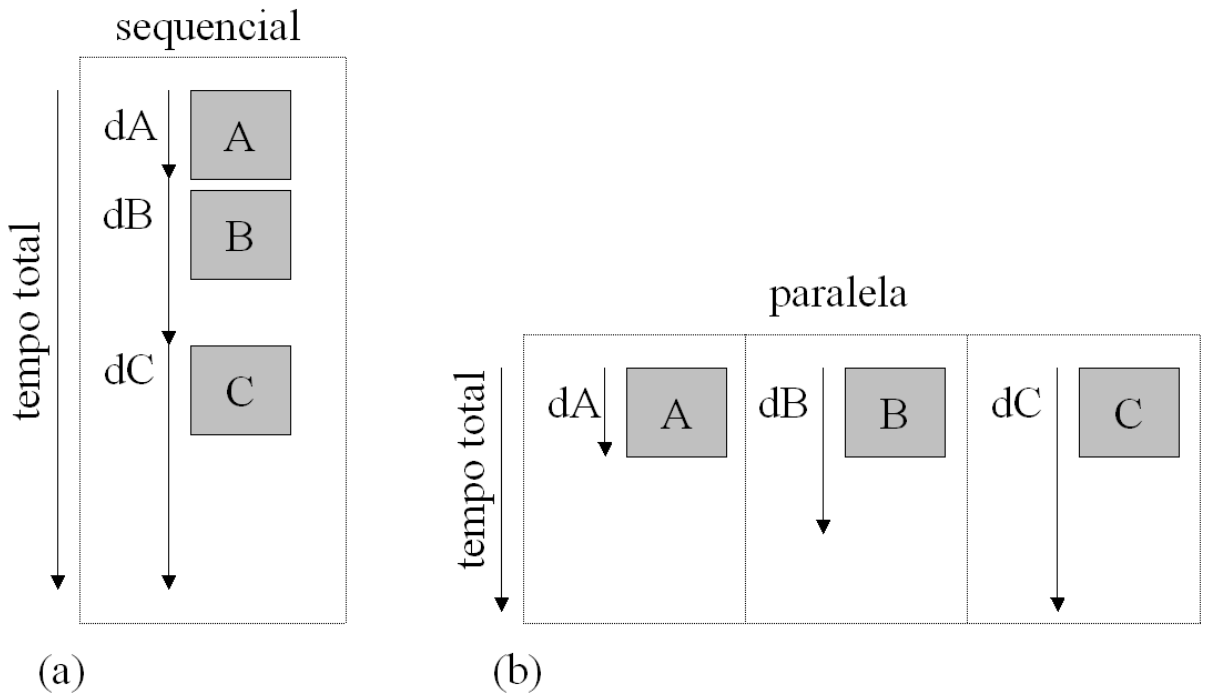

Figura 2.17: Elementos de um circuito implementados de forma sequencial (a) e de forma paralela (b)

Supondo o caso hipotético em que dA = 1 ms, dB = 2 ms e dC = 3 ms, uma implementação serial implica que o tempo total que o circuito precisará para que a lógica seja processada é de dA + dB + dC = 6 ms. Enquanto que uma versão na qual todos os processos ocorram em paralelo precisará de dC = 3 ms apenas.

Em filtros digitais e redes neurais, por exemplo, é muito comum a ocorrência de operações de multiplicação de amostras de um sinal por coeficientes ou pesos, a subsequente soma dos resultados dos produtos. A esse tipo de processo se dá o nome de MAC (*multiply and accumulate*), e é bastante beneficiado por essa característica (Kilts, 2007).

2.4.2 Implementação

A programação de um determinado sistema no *hardware* da Xilinx pode ser feita tanto em linguagem Verilog quanto VHDL. O presente trabalho foi implementado em VHDL e fez uso do ambiente de programação da própria Xilinx, o Xilinx ISE (*Integrated Software Environment*), cuja interface pode ser visualizada na Figura 2.18.

O desenvolvimento do projeto consiste de 8 etapas que são em grande parte transparentes

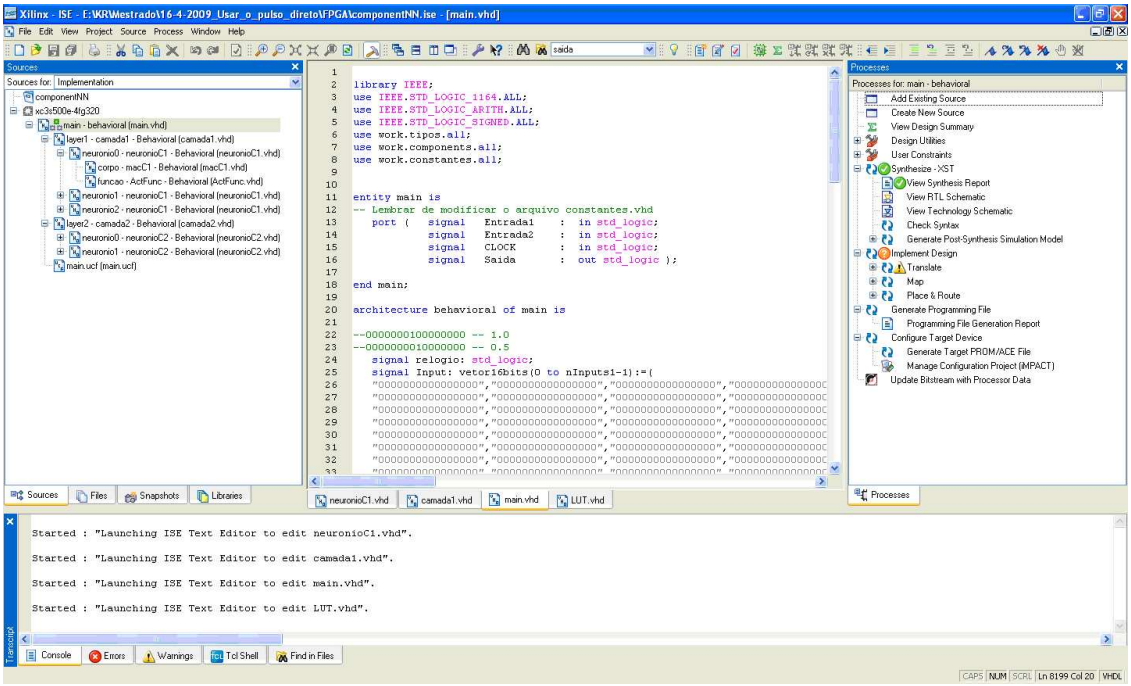

Figura 2.18: Ambiente de programação Xilinx ISE

para o desenvolvedor visto que a maior parte delas, em especial os últimos passos, são automáticas:

1. *Design Entry*
2. *Simulation*
3. *Synthesis*
4. *Implementation*
5. *Map*
6. *Place and Route*
7. *Timing Simulate*
8. *Program*

Em *Design Entry*, o projeto é definido pelo usuário na forma de código ou esquemas, em *Synthesis* o projeto é sintetizado pelo XST (*Xilinx Synthesis Technology*) da Xilinx. O passo de simulação é opcional, e é útil para analisar o comportamento do projeto antes de carregá-lo no *hardware*.

Após a simulação, o estágio de sintetização converte o código desenvolvido no estágio de *Design* para um arquivo *netlist* (NGC), que contém a descrição em baixo nível do circuito a ser implementado.

O passo de implementação traduz o projeto sintetizado para que possa ser gravado no FPGA utilizando os arquivos de *netlist* e *constraints*. *Constraints* são parâmetros de funcionamento e implementação do *hardware* que podem ser definidos pelo usuário ou automaticamente pelo *software*. Alguns exemplos são estabelecer limites de tempo, ligar saídas e entradas dos componentes às saídas e entradas do dispositivo, limitações de área e etc.

A fase de mapeamento distribui o projeto em função dos recursos disponíveis no FPGA. Em *Place and Route* é feito o trabalho de decisão quanto à alocação das células lógicas. E finalmente em *Program* o resultado da fase anterior é utilizado por um programa chamado *bitgen* para gerar um *bitstream* que será enviado ao FPGA.

2.4.3 MicroBlaze™

O Microblaze é um microprocessador de 32 bits e arquitetura RISC que pode ser implementado diretamente no FPGA da Xilinx (Xilinx, 2007). A ferramenta de programação respectiva da Xilinx é o EDK (*Embedded Development Kit*), formado por dois outros componentes, o XPS e o SDK. O XPS (*Xilinx Platform Studio*) é o responsável por gerar os arquivos de *netlist* que serão utilizados para se implementar a descrição do projeto em *hardware*. Já o SDK é responsável por lidar com o *software* que será executado sendo que a programação é feita em C.

A vantagem no uso do MicroBlaze é a facilidade de acesso aos recursos disponíveis no *kit*. Através dele, é possível utilizar a memória SDRAM de 64 Mb, já inclusa no corpo do kit, conversor A/D e dispositivos de comunicação como USB, ethernet e serial.

Mais detalhes sobre seu uso para o estabelecimento da comunicação entre o *kit* e o PC externo podem ser vistos no Apêndice A.2.

Capítulo 3

Desenvolvimento dos Métodos e Implementação

3.1 Detecção de Pulsos

3.1.1 Introdução

Antes de pensar em um software ou dispositivo capaz de discernir um pulso anômalo de um saudável, é necessária no entanto a capacidade de se detectar os pulsos nos longos sinais de ECG para que os mesmos possam ser analisados.

Com a análise da derivada do sinal, suas amplitudes e extensões no eixo temporal alguns trabalhos alcançaram uma capacidade de detecção do complexo QRS de até 99,3% (Pan and Tompkins, 1985), após o uso de filtros que permitem a aplicação de uma linha limite bem baixa.

É importante frisar que nem sempre os sinais de ECG estão em um formato adequado para a detecção automática dos pulsos. Devido à frequente ocorrência de ruídos e interferências, oriundas de eventos díspares como a própria rede elétrica ou do movimento de corpo do paciente, é necessário antes de tudo um tratamento adequado do sinal.

Cada sinal de ECG é no entanto distinto. Como a sua gravação pode ser feita ao longo de um intervalo de tempo considerável, chegando a horas em alguns casos, as chances de interferências graves ocorrerem são grandes. Tal desafio acaba exigindo uma abordagem adaptativa por parte do algoritmo de detecção, para evitar que oscilações na linha de base, ruídos ou presença de artefatos no sinal invibializem o processo de detecção.

Há um número considerável de trabalhos que utilizam redes neurais para a detecção do complexo QRS. O reconhecimento auto-organizável aliado à adaptabilidade da rede neural chegou a produzir erros de reconhecimento inferiores a 1 ms (Suzuki, 1995).

De fato, o uso de redes neurais para a identificação dos pulsos dá espaço a soluções criativas como o treinamento da rede para reconhecer tudo o que não é QRS, e através de uma análise por janelamento, utilizar o erro crescente da rede ao se deparar com um complexo QRS para identificá-lo (Vijaya et al., 1997). Neste caso as entradas fornecidas à rede neural para o seu treinamento são compostas por amostras extraídas do sinal de ECG apenas nos intervalos entre os pulsos. Ao ser treinada para reconhecer intervalos entre pulsos a rede acaba produzindo uma saída com erro alto quando lhe é fornecida uma série de dados que contém um pulso de ECG. Essa série de dados é uma janela móvel que percorre o sinal de ECG e é fornecida à rede.

Mesmo nos trabalhos mais antigos, nas décadas de oitenta e noventa, já se notava uma atenção especial para o uso de *wavelets* no processo de detecção de pulsos (Vijaya et al., 1997). De fato um grande número de trabalhos mais recentes utilizam transformada *wavelet* para a detecção das ondas que compõem o pulso do eletrocardiograma. A transformada *wavelet* é especialmente eficaz no processo de detecção dos complexos QRS por permitir a eliminação da linha de base, interferência devido a artefatos e ruídos de maneira direta, não requerendo o uso de outros filtros. Seu desempenho é tão notável que chega a permitir taxas de detecção muito próximas dos 100%, especialmente quando soluções envolvem *wavelets* de vários tipos para casos específicos (Saxena et al., 2002).

Com efeito, a escolha da *wavelet*-mãe adequada para a solução de um problema específico é de vital importância. *Wavelets*-mãe do tipo *haar* são simples de se computar e seu filtro possui poucos coeficientes, o que torna sua implementação mais ágil (Mahmoodabadi et al., 2005). As outras *daubechies* no entanto apresentam um desempenho geralmente superior, em especial para aplicações envolvendo a análise de sinais de ECG, como será visto a seguir, mas apresentam o revés de que seus filtros são maiores, o que vem a elevar o custo computacional.

Tendo isso em mente, pode-se desenvolver um algoritmo que permita a identificação dos pulsos para que os mesmos sejam posteriormente analisados pelo método desenvolvido neste trabalho.

3.1.2 Escolha das ferramentas e métodos

O uso da transformada *wavelet* para a detecção não apenas dos complexos QRS como também de ondas distintas, como ondas P, R ou T, tem se tornado cada vez mais comum nos trabalhos atuais.

Com base nesse sucesso, buscou-se implementar um sistema de detecção do QRS no presente trabalho que permitisse usar a informação da posição do QRS para colocar de maneira estratégica uma janela no tempo de análise sobre o pulso de ECG. É essa janela que

fornecerá à rede neural as informações necessárias para a detecção de anomalias e análise do ECG como um todo, englobando um pulso básico completo. É vital portanto que o desempenho da detecção do complexo QRS, ou mais diretamente da busca pelo pico da onda R, tenha um desempenho satisfatório.

Este desempenho é função direta de vários fatores tais como a seleção da *wavelet*-mãe apropriada, a escolha correta dos níveis de decomposição e o tratamento que se faz sobre esses novos dados obtidos.

De maneira geral, uma primeira tentativa válida para abordar a questão da escolha da *wavelet* mãe é a do tipo *haar*, como pode ser vista na Figura 3.1.

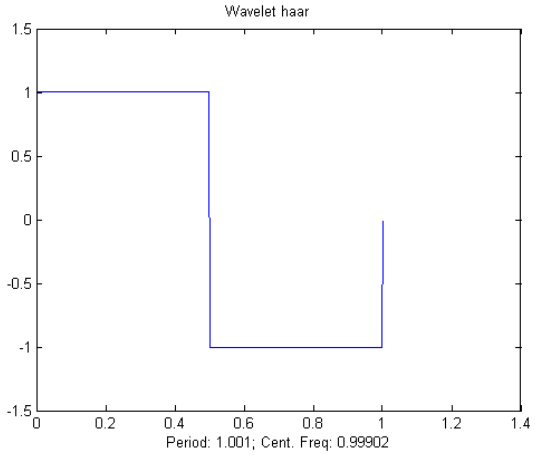

Figura 3.1: *Wavelet*-mãe tipo Haar

A principal vantagem no uso da *wavelet* do tipo *haar* em aplicações digitais é o tamanho reduzido de seu filtro e sua facilidade de implementação, como já mencionado. Como a análise *wavelet* é beneficiada pelo conceito de similaridade, é importante, em muitos casos, escolher uma *wavelet*-mãe que se assemelhe ao sinal analisado. A desvantagem mais aparente da *haar* é então que o seu formato não apresenta similaridade com o sinal de ECG.

Com base em resultados de trabalhos reunidos na literatura (Saxena et al., 2002), testes foram realizados com as *wavelets*-mãe da db1 a db6, sendo que os melhores resultados foram obtidos pelas db2 e db6. A db6 teve um desempenho ligeiramente superior ao da db2, o que levou à sua escolha final.

Na Figura 3.2, apresentam-se as decomposições em termos das aproximações e detalhamentos até o quarto nível, além do sinal original. Verificou-se que os níveis 3 e 4 de detalhamento foram os mais eficazes para ressaltar a posição do pico R, visto que os detalhamentos de nível 1 e 2 possuem um nível de ruído elevado. Notar por exemplo como as oscilações da linha de base do sinal original não estão presentes nos detalhamentos da decomposição.

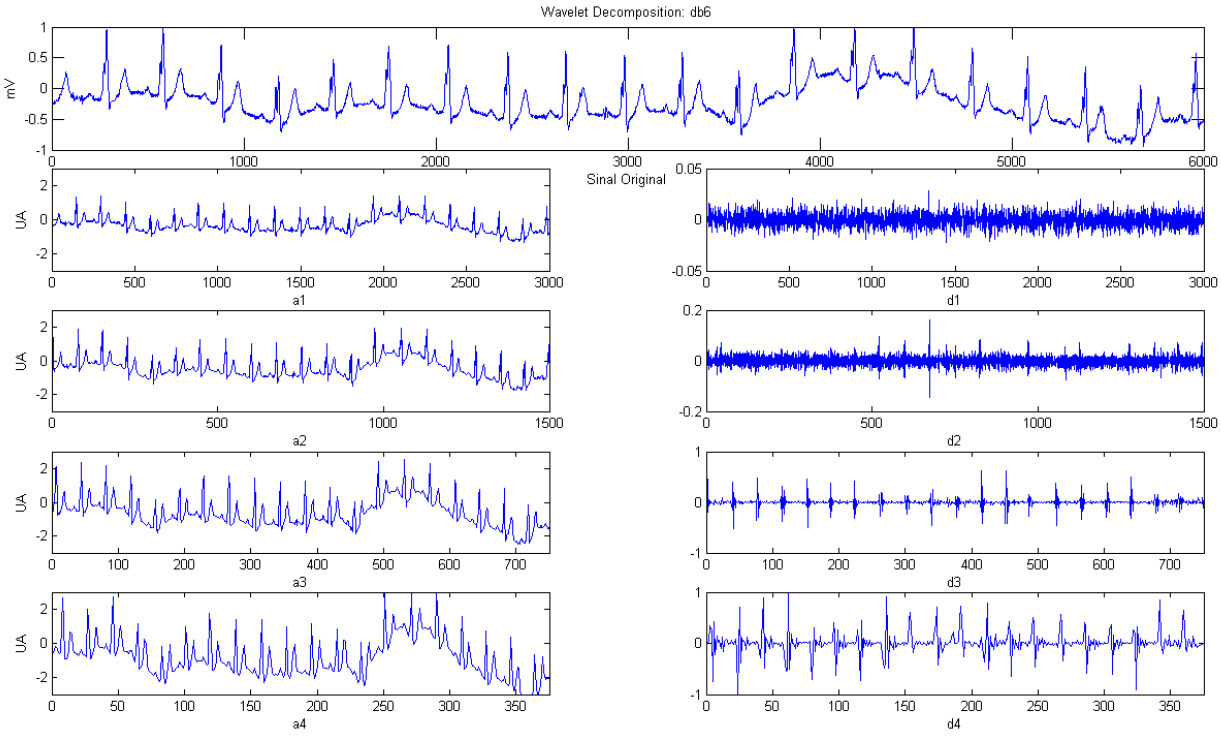

Figura 3.2: Decomposição com a DWT por db6 em 4 níveis

O quarto nível de decomposição foi escolhido, pois o terceiro apresenta amplitudes insuficientes em um número considerável de detecções. O sinal de decomposição nível 4 é então elevado ao quadrado, para garantir maiores amplitudes que venham a ajudar na diferenciação dos picos R daqueles picos de base que constituem o ruído do presente sinal.

Em seguida, é preciso localizar os picos referentes à onda R. O sinal de detalhamento pode ser visto como uma sucessão de pontos no espaço bidimensional sendo que alguns destes pontos são picos e outros não. Assim, o primeiro passo tomado foi o de discernir picos de não-picos de acordo com a Figura 3.4. De acordo com a Figura 3.4, um ponto b é considerado um pico local caso ambos os pontos adjacentes, pontos a e c, tenham amplitude inferior ao ponto b em questão. Isto ocorre no Caso 1, ilustrado na figura, mas não nos Casos 2 e 3.

É necessário então fazer a distinção entre picos R dos picos de base. Picos de base são aqueles que se encontram próximos da linha de base do sinal, podendo ser fruto de ruído ou mesmo de ondas que não sejam a onda R de interesse. Há vários métodos para se fazer tal distinção.

O método mais direto, e amplamente utilizado, é o estabelecimento de um valor limite que separe os picos desejados dos indesejados. O problema de uma abordagem tão simples é que existe uma variação grande entre as magnitudes dos picos R de sinais de pessoas diferentes, sendo que muitas vezes mesmo o sinal de uma mesma pessoa apresenta variação suficiente

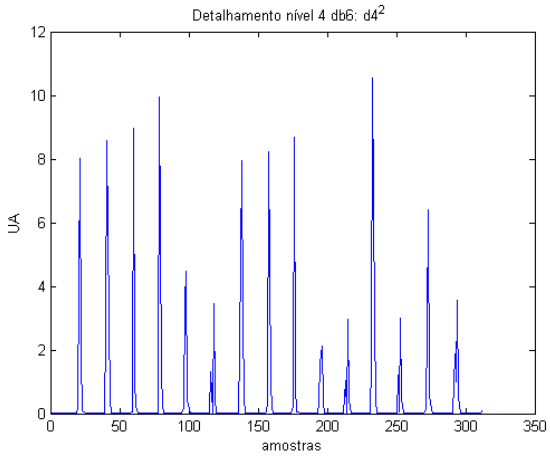

Figura 3.3: Detalhamento elevado ao quadrado

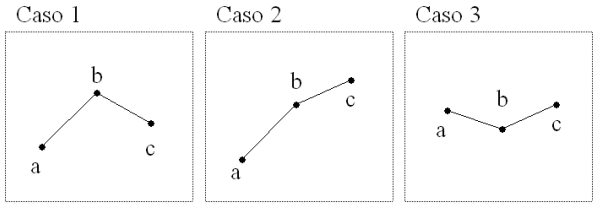

Figura 3.4: Diferenciação dos pontos locais em picos ou não picos. Os pontos a, b e c são três amostras em sequência do sinal sendo analisado: O ponto b é um pico local para o caso 1, mas não para os casos 2 e 3

para invalidar o processo.

Torna-se necessário então um método adaptativo que leve em consideração as mudanças ao longo do ECG. Para isto há várias estratégias diferentes na literatura, desde o uso de conjuntos nebulosos até técnicas de *clustering*.

A solução adotada neste trabalho foi o uso da média móvel do sinal como a linha limite de separação entre os picos R e os indesejados que se encontram na base do sinal. Uma ilustração do método empregado pode ser vista na Figura 3.5, para uma média móvel com janela de tamanho aproximado ao do período de três pulsos.

Para cada amostra do sinal pré-processado é calculada uma média móvel cuja extensão é o número de amostras equivalente ao período de três pulsos, e o resultado pode ser visto na Figura 3.5. Isso faz com que a linha delimitada pela média calculada aumente em amplitude quando um pico do tipo R, já amplificado pelo pré-processamento, entra na janela em questão. O resultado é que os picos que fazem parte da parte mais inferior do sinal ficam abaixo da linha da média móvel enquanto que aqueles referentes às ondas R ficam acima da mesma, permitindo assim a distinção dos dois tipos de picos. Com efeito, a média móvel

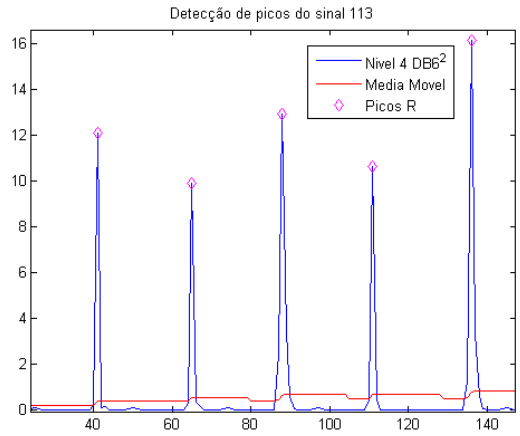

Figura 3.5: Média móvel para o discernimento entre picos: Pontos acima da média móvel são picos de onda R em potencial, enquanto que picos abaixo da média móvel são picos referentes ao ruído de base

permite que o limite se adapte às necessidades presentes em sinais provenientes de fontes diversas. Por fim, é preciso fazer o *upsampling* do sinal de detalhamento utilizado para que os picos detectados coincidam com as posições dos picos das ondas R no sinal original. O resultado produzido pode ser apreciado na Figura 3.6, onde as posições anotadas dos pulsos foram obtidas no banco de dados da Physionet, sendo que cada identificação foi feita por dois ou mais cardiologistas de maneira independente, como explicado em (Goldberger et al., e 13). Verificou-se uma diferença média de tempo de 10 ms entre a detecção feita pelo algoritmo e a anotação do banco de dados.

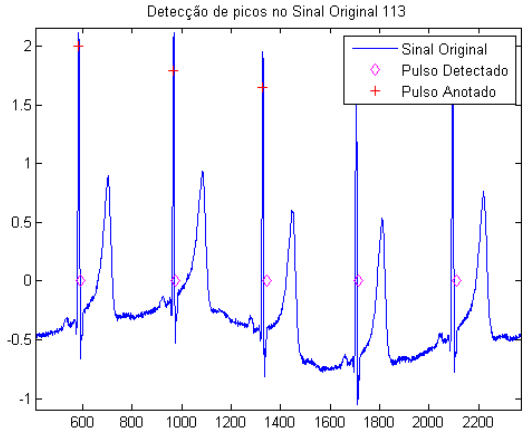

Figura 3.6: Comparação entre os pulsos detectados e os anotados

3.2 Identificação de anomalias

3.2.1 Introdução

Como visto na Seção 2.1, o sinal de ECG é composto por um longo registro dos impulsos elétricos emitidos durante o processo de funcionamento do coração. Visto que cada etapa do batimento cardíaco produz uma assinatura característica, é possível detectar anomalias no funcionamento do coração a partir de anomalias no sinal de ECG.

A identificação dessas anomalias e particularidades não é entretanto uma tarefa fácil. Aliado ao longo período de gravação dos ECGs, existe um número de características de difícil identificação para o analista humano. Uma abordagem computacional torna-se então de grande importância para auxiliar o analista humano, reduzindo a taxa de erro de diagnóstico.

O tratamento computacional do sinal de ECG apresenta seus próprios desafios. Por se tratar de um sinal biológico, adquirido a partir de instrumentos biomédicos, está sujeito a uma miríade de interferências, ruídos e variações. Tais interferências acabam exigindo, independentemente da técnica utilizada, um pré-tratamento para a eliminação de ruídos e de oscilações da linha de base.

A literatura é farta em técnicas para a detecção de anomalias em sinais de ECG, tais como: análise de autocorrelação dos sinais em busca de alterações (Yu and Chen, 2007), análise de características no domínio da frequência (Yu and Chou, 2008), análises tempo-frequência e transformada *wavelet* (Yu and Chen, 2007). A decisão das características que servirão de base para a análise do ECG pode ser tomada de diversas maneiras. Uma análise de componentes independentes pode, por exemplo, fornecer um grupo de estatísticas que sejam mutuamente independentes (Yu and Chou, 2008). As características de cada cardiopatia, por exemplo, podem ser analisadas de maneira individual e estatísticas apropriadas podem ser escolhidas. A escolha de um pré-processamento adequado deve possibilitar a extração ou ressaltar as características que se busca identificar.

O intuito final, no entanto é produzir um grupo de parâmetros que permita a diferenciação de cada doença cardíaca em grupos distintos. Como já mencionado, resultados atuais mostram que a transformada *wavelet* produz os resultados mais promissores na identificação de anomalias em ECGs (Güler and Übeyll, 2005), apresentando resultados superiores a outros métodos como o uso da transformada de Fourier (Dokur et al., 1999).

Uma vez em posse deste grupo de parâmetros, é necessário o uso de alguma técnica classificadora para a identificação de cada cardiopatia. Neste trabalho empregou-se redes neurais artificiais do tipo MLP. Tal escolha foi baseada no fato de que muitas das características, que definem os grupos aos quais cada cardiopatia pertence, não são linearmente separáveis. Outro ponto que pesou em favor da escolha do uso de redes MLP foi o seu sucesso e uso

extensivo em diversas aplicações que podem ser verificadas na literatura (Yu and Chen, 2007).

Para os procedimentos descritos neste capítulo foram utilizados os *toolboxes* de análise *wavelet* e de redes neurais do Matlab versão 7.1 R14 (Higham and Higham, 2005).

3.2.2 Materiais e métodos

Extração de Características

Os sinais foram obtidos no banco de dados Physionet (Goldberger et al., e 13) e foi escolhida a derivação do tipo MLII. A análise dos sinais buscou diferenciar os quatro tipos possíveis de pulsos (normal, contração ventricular prematura, bloqueio de ramo direito, bloqueio de ramo esquerdo). Como material de estudo utilizou-se os sinais da Tabela 3.1.

Tipo	Símbolo	Sinais
Normais	N	100, 113
Contração Ventricular Prematura	V	106, 119
Bloqueio de Ramo Direito	R	231, 118
Bloqueio de Ramo Esquerdo	L	111, 214

Tabela 3.1: Sinais de ECG analisados

É importante que haja mais de uma fonte de pulsos para cada tipo de anomalia para que não ocorra polarização do classificador em função do paciente que forneceu o sinal. De fato, quanto maior e mais variado for o número de amostras, mais genérico será o sistema identificador de anomalias.

Como já mencionado, este trabalho utilizou a transformada *wavelet* discreta (DWT) para realizar o pré-processamento de pulsos individuais de ECG. Como a transformada *wavelet* opera com base na similaridade da *wavelet*-mãe com o sinal analisado, é extremamente importante escolher uma *wavelet*-mãe que possa representar o sinal analisado de maneira satisfatória. As daubechies são conhecidas por apresentarem resultados excelentes no tratamento de sinais de ECG. Alguns trabalhos relataram o uso de métodos para a escolha de uma *wavelet*-mãe ótima (Castro et al., 2000), enquanto outros avaliaram os resultados para diferentes protótipos (Güler and Übeyll, 2005). Neste trabalho escolheu-se a *daubechie* 6 (DB6) devido a sua maior similaridade com o complexo QRS (Mahmoodabadi et al., 2005), como mencionado anteriormente, e aos bons resultados apresentados.

Utilizando os sinais da Tabela 3.1 foram extraídas janelas com 256 amostras, sendo que cada janela contém um único pulso. Esta premissa foi utilizada com sucesso para a classificação de cardiopatias com redes neurais em (Güler and Übeyll, 2005).

O sinal com o pulso extraído foi decomposto em quatro níveis de detalhamento e 4 níveis de aproximação. Cada nível de decomposição produz sinais com um número inferior de amostras que o do nível superior.

Em análise de sinais, é comum reduzir os dados obtidos a um conjunto de parâmetros que representem aquilo que se deseja identificar. A esses parâmetros também se dá o nome de *features* ou características.

O uso de parâmetros na identificação de certas qualidades presentes nos sinais é importante não apenas para facilitar a separação das classes presentes nos sinais, como também para reduzir a dimensionalidade do problema (Güler and Übeyll, 2005).

Tendo isso em vista, um vetor de características foi montado extraindo-se a variância e a média de cada nível de detalhe, o mesmo sendo feito para o quarto nível de aproximação. A escolha das estatísticas foi baseada em testes realizados com combinações de estatísticas sugeridas na literatura (Güler and Übeyll, 2005). O vetor resultante é como o que se segue:

$$V = \left[\overline{d1}\ \ \overline{d2}\ \ \overline{d3}\ \ \overline{d4}\ \ \overline{a4}\ \ var(d1)\ \ var(d2)\ \ var(d3)\ \ var(d4)\ \ var(a4)\right]$$

Sendo $\overline{d1}$, $\overline{d2}$, $\overline{d3}$, $\overline{d4}$ as médias dos detalhes de nível 1, 2, 3 e 4 respectivamente e $\overline{a4}$ a média da aproximação do quarto nível. Já var(d1), var(d2), var(d3) e var(d4) são as variâncias dos detalhes de nível 1, 2, 3 e 4 respectivamente e var(a4) a variância da aproximação de nível 4.

O vetor de dados é então normalizado de acordo com a Equação 3.1 para o cálculo do desvio padrão não enviesado segundo a Equação 3.2, sendo N o número de elementos do vetor.

$$V_{norm\ i} = \frac{\left(V_i - \overline{V}\right)}{\sigma} \tag{3.1}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}\left(V_i - \overline{V}\right)^2}{N-1}} \tag{3.2}$$

Como muitas das características do pulso de ECG podem ser discernidas a partir da análise de seu formato, escolheu-se adicionar ao vetor de classificação o quarto nível de aproximação, também normalizado. A adição dos elementos da aproximação de nível quatro aumenta a capacidade de reconhecimento de padrões pela rede neural, mas também exige uma rede com uma estrutura maior para lidar com o número extra de entradas.

O vetor final de características ficou então como o da Equação 3.3, formando um vetor

de dimensão 36. Um exemplo gráfico do vetor resultante pode ser visto na Figura 3.7 com exemplos das anomalias de bloqueio de ramo esquerdo (L111 e L214) e de bloqueio do ramo esquerdo (R231 R118).

$$V_{final} = [V_{norm} \quad a4_{norm}] \tag{3.3}$$

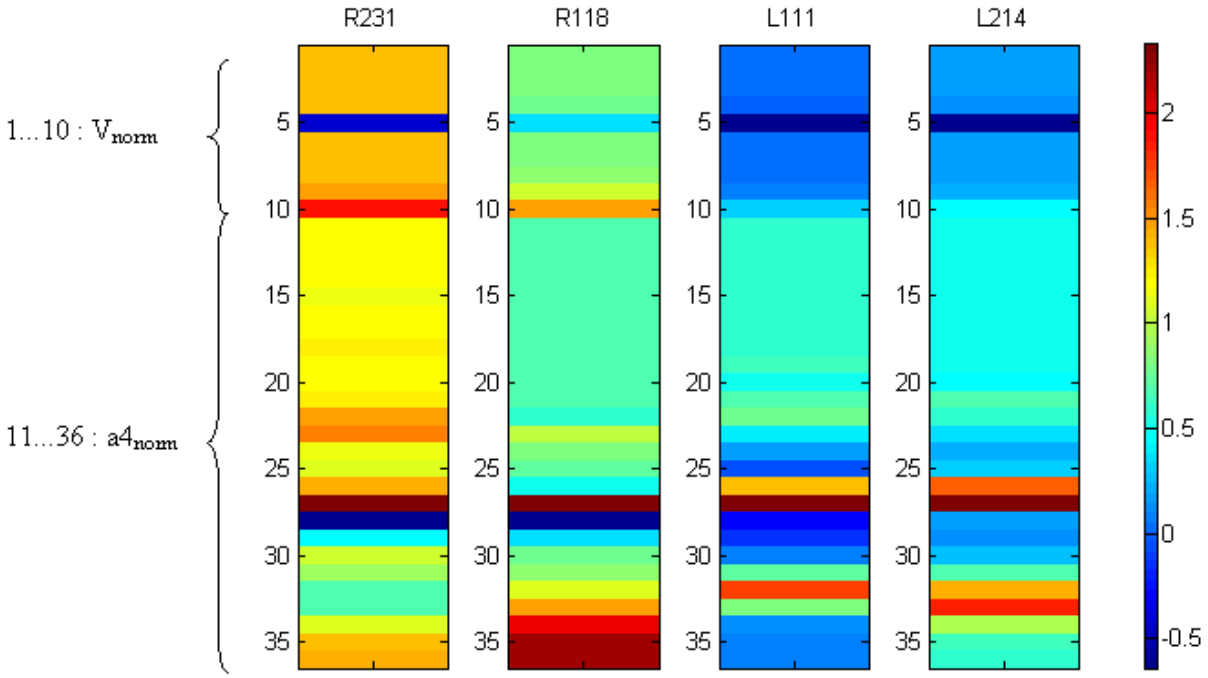

Figura 3.7: Exemplos de vetores de características para a classificação

As cores indicam as magnitudes de cada amostra dos vetores, as amostras de 1 a 10 equivalem aos parâmetros calculados e as amostras de 11 a 36 equivalem às aproximações.

3.2.3 Classificação com redes neurais

Tendo em vista que o objetivo final do desenvolvimento da rede neural é a sua implementação em *hardware*, foram feitos testes com diversas topologias de rede buscando sempre o melhor desempenho de classificação.

Após inúmeros testes, duas configurações distintas foram obtidas para a rede neural classificadora. Essas configurações foram denominadas como:

- *Modelo de rede completa*, que produziu os melhores resultados de classificação.

- *Modelo de rede reduzida*, que produziu resultados próximos aos alcançados pelo modelo completo mas com uma estrutura de rede reduzida.

Um comprometimento entre desempenho e tamanho de projeto pode ser necessário dependendo do *hardware* disponível. Importante lembrar no entanto que uma redução no número de neurônios da rede neural produz uma redução do número de graus de liberdade que ela é capaz de modelar, podendo comprometer sua capacidade de generalização.

O modelo de rede completa difere do modelo reduzido pelo número de parâmetros avaliados e pelo número de neurônios na camada oculta. Suas características serão detalhadas nas seções a seguir.

Rede Completa

A rede neural completa foi dimensionada tendo 36 entradas e 4 saídas. Cada saída foi atribuída a uma classe que representa uma anomalia específica, sendo que para cada pulso apresentado, apenas uma classe deve ter valor próximo de *um*, enquanto que as outras devem estar com valores próximos de *zero*. As regras de classificação são então como descrito na Tabela 3.2, sendo S1, S2, S3 e S4 as saídas da rede neural.

Saída	S1	S2	S3	S4
Normal	1	0	0	0
Contração Ventricular Prematura	0	1	0	0
Bloqueio de Ramo Direito	0	0	1	0
Bloqueio de Ramo Esquerdo	0	0	0	1

Tabela 3.2: Regras para a classificação dos pulsos de acordo com as saídas da rede neural

O treinamento utilizado foi o método do gradiente conjugado de Beale-Powell e tanto para a camada oculta quanto para a de saída foi utilizada a função de ativação do tipo sigmóide.

Os dados de entrada, já pré-processados como descrito previamente, foram agrupados em uma matriz na qual cada coluna equivale aos parâmetros extraídos de um determinado pulso. A ordem de montagem da matriz foi a seguinte:

$$[N100_1\ V106_1\ R231_1\ L111_1\ N113_1\ V119_1\ R118_1\ L214_1\ N100_2\ V106_2...]$$

Onde as letras denotam o tipo de anomalia como explicado no capítulo sobre cardiopatias e o número indica o sinal do qual tal amostra foi retirada. Ou seja, $N100_1$, por exemplo, é um vetor coluna com 36 valores.

A primeira coluna fica então com parâmetros de um pulso do tipo normal do sinal 100, a segunda coluna fica com parâmetros de um pulso com contração ventricular prematura do sinal 106 e assim sucessivamente. Quando todos os sinais são utilizados, volta-se a utilizar pulsos dos sinais anteriores, na mesma sequência, sendo que estes pulsos são diferentes dos anteriores.

Intercalar os tipos de anomalias durante o treinamento é importante para evitar que o aprendizado fique polarizado em direção a um grupo em particular durante o processo.

Foram separados 256 pulsos de cada sinal para o processo de treinamento e 128 para a validação. Tomou-se o cuidado de se retirar um número equivalente de pulsos de cada tipo, novamente para evitar a polarização da rede em direção a uma ou outra cardiopatia.

O treinamento foi feito com vários números diferentes de neurônios na camada oculta para identificar o melhor caso potencial. Os testes foram feitos com o número de neurônios variando de 5 até 30 de um em um. Embora alguns casos tenham produzido resultados similares, deu-se preferência a um número menor de neurônios para reduzir o custo computacional.

Os treinamentos foram feitos até 1000 épocas, sendo que os valores de peso que produziram os menores erros de validação foram guardados. Após a primeira centena de épocas o erro de validação começou a aumentar, embora o erro de treinamento continuasse a cair. Isso é um indicativo de que a partir daquele ponto a rede estava sofrendo um *overfitting* por sobre-treinamento. Isso é prejudicial para o uso da rede como ferramenta de classificação, pois diminui a sua capacidade de generalização. O melhor resultado obtido pode ser apreciado na Figura 3.8. A figura mostra o progresso do erro da rede ao longo das épocas de treinamento em azul e o progresso do erro na classificação do conjunto de validação em verde.

A rede adotada utilizou 15 neurônios na camada intermediária e apresentou maior sucesso ao ser treinada por 110 épocas.

Modelo de Rede Reduzida

A rede neural reduzida foi dimensionada tendo 10 entradas e 4 saídas. O padrão de saídas, a função de ativação utilizada nas duas camada, o algoritmo de treinamento utilizado e a organização dos dados de validação e treinamento foram os mesmos descritos na Seção 3.2.3.

Neste caso, os testes foram feitos com o número de neurônios variando de 3 até 15 de um em um. Embora neste caso alguns resultados similares também tenham sido obtidos, novamente deu-se preferência a um número menor de neurônios para reduzir o custo computacional.

Os treinamentos foram feitos até 3000 épocas, sendo que os valores de peso que produziram os menores erros de validação foram guardados. O melhor resultado obtido pode ser apreciado na Figura 3.9, sendo que a curva em azul representa o erro do treinamento e a

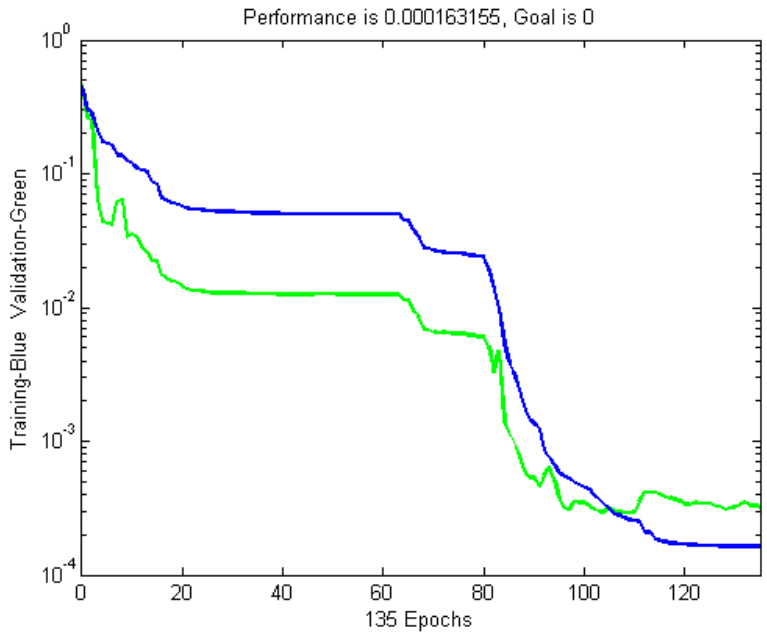

Figura 3.8: Treino da rede neural para a classificação dos pulsos de ECG

curva em verde representa o erro de validação.

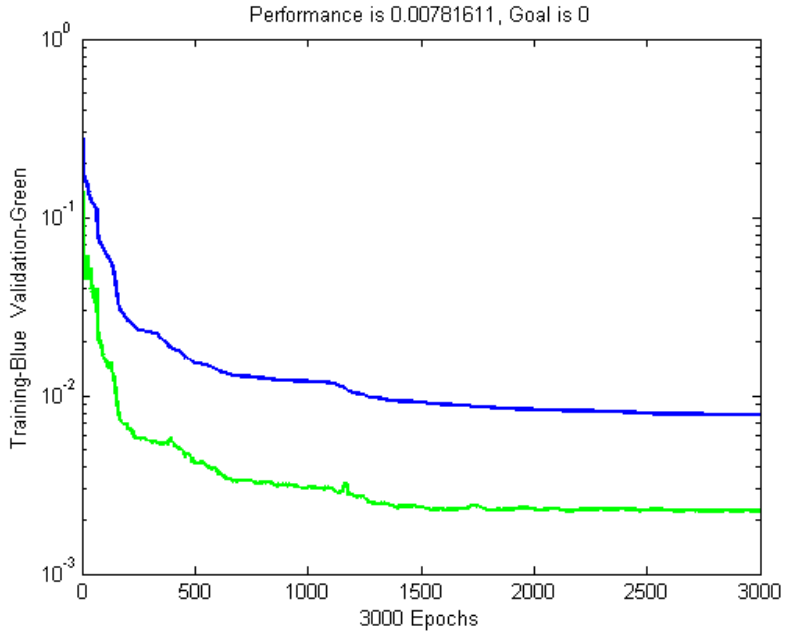

Figura 3.9: Treino da rede neural reduzida para a classificação dos pulsos de ECG

A rede adotada utilizou 5 neurônios na camada intermediária e apresentou maior sucesso ao ser treinada por 2890 épocas.

3.3 Implementação em FPGA

Há diversas maneiras de se implementar redes neurais em *hardware* programável como o FPGA. Devido à facilidade de se projetar os circuitos que compõem a lógica interna do componente com o uso de linguagens como o VHDL, estruturas complexas como uma rede neural podem ser implementadas tanto de forma integralmente paralela como de maneira sequencial, seguindo os pulsos do *CLOCK* disponível para o dispositivo.

Nesta seção serão apresentadas duas formas distintas de se implementar uma rede neural em VHDL:

- Modelo por Componentes
- Modelo Matricial

Chama-se de modelo por componentes o projeto da rede neural partindo de suas partes mais básicas: soma ponderada, função de ativação. Esses componentes são projetados separadamente e em seguida montados para se formar a rede final.

O modelo matricial segue a definição matemática da rede neural, de modo que a mesma realize as operações de multiplicação entre matrizes e vetores e aplicação da função de ativação diretamente como na Equação 2.2.

O funcionamento de ambos os modelos será explicado nas seções seguintes. Primeiramente são descritos o neurônio e seus sub-componentes: soma ponderada e função de ativação. Em seguida, descreve-se como esses componentes se encaixam nos modelos de redes neurais descritos.

3.3.1 Modelo por Componentes

Estrutura do Neurônio

A estrutura do neurônio em VHDL segue o visto na Figura 2.9 da Seção 2.2.1 sobre redes neurais. O componente do neurônio é composto por dois outros componentes: soma ponderada e função de ativação. Um diagrama da sua implementação pode ser visto na Figura 3.10.

As entradas do componente são duas:

- *Input*: O vetor de entradas da camada em questão
- W : O vetor de pesos que será multiplicado pelo vetor de entradas

As operações realizadas dentro do componente neurônio são, como já mencionado, a soma ponderada e a aplicação da função de ativação, como se segue.

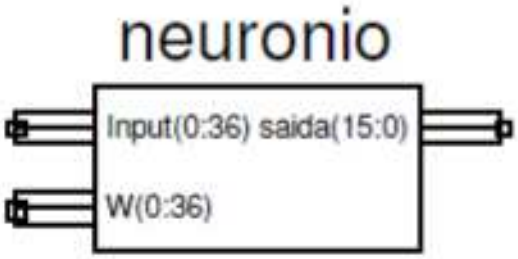

Figura 3.10: Implementação em VHDL do componente referente ao neurônio artificial

Soma Ponderada

A soma ponderada que ocorre no corpo do neurônio é feita a partir da multiplicação de cada entrada do neurônio pelo seu peso correto. No presente modelo, essa multiplicação é feita de maneira paralela, não havendo a necessidade de se acumular o resultado de cada multiplicação serialmente para só então se calcular a soma. Um exemplo esquemático daquilo que foi dito pode ser visto na Figura 3.11.

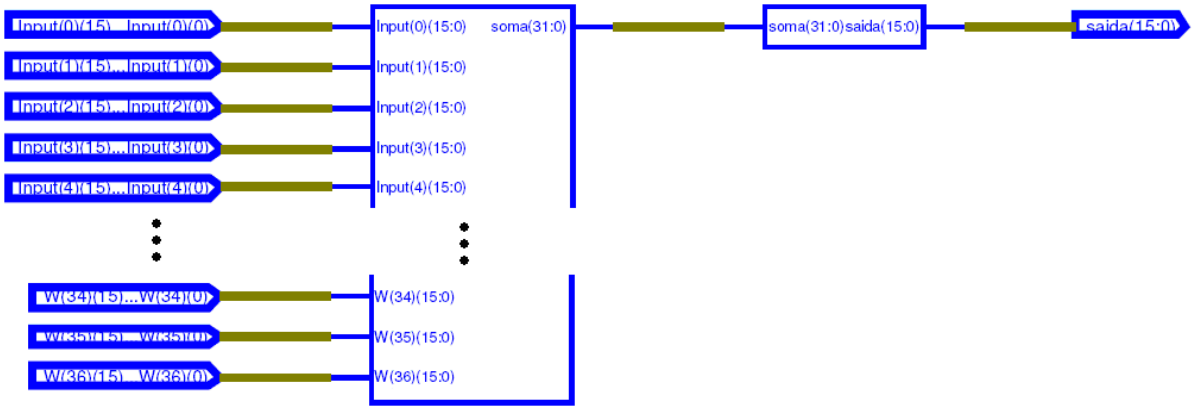

Figura 3.11: Esquema do componente resultante que realiza a soma ponderada das entradas de um neurônio

Notar na Figura 3.11 que as entradas são as mesmas descritas na Figura 3.10 com detalhes da estrutura das operações realizadas. O vetor *Input* tem cada elemento multiplicado pelo seu respectivo peso no vetor W mais abaixo. No mesmo bloco de instruções as multiplicações são todas acumuladas, produzindo uma soma total.

Como os números são binários (ver apêndice A.1 para mais detalhes), o produto dos dois números de 16 bits produz um terceiro número de 32 bits. O bloco mais à direita da figura extrai os 16 bits de interesse para produzir uma saída com o mesmo número de bits que a entrada do componente.

O passo seguinte é fornecer essa soma ponderada à função de ativação para que a mesma produza a saída do neurônio. Neste projeto a função de ativação utilizada foi a sigmóide, embora outras funções também possam ser implementadas de maneira similar.

Função de ativação em *Hardware*

Como visto nos capítulos anteriores, a função de ativação desempenha um papel crucial na capacidade da rede neural de desempenhar a função para a qual foi projetada. Embora a implementação das funções de ativação em software seja relativamente fácil, em especial para os casos mais clássicos como a tangente hiperbólica e a sigmóide, fazer com que elas funcionem em *hardware* é uma tarefa um pouco mais difícil. A dificuldade vem do fato de que essas funções não estão disponíveis de imediato, o que exige a sua implementação por parte do programador.

Há uma série de métodos para se obter uma função de ativação em *hardware*. Alguns utilizam aproximações por série de Taylor (Omondi and Rajapakse, 2006) para obterem um polinômio aproximado, outros fazem uso de algoritmos de aproximação iterativa como o CORDIC. O método utilizado neste trabalho foi a representação da função de ativação através de uma *lookup table*.

Lookup Table

A *lookup table* é uma estrutura de memória similar a um vetor de dados que podem ser acessados a partir de uma dada posição de memória. Com uma estrutura similar à vista na Tabela 3.3, ela pode ser implementada como um vetor de constantes em linguagem VHDL.

Tabela 3.3: Exemplo da estrutura de uma *Lookup Table*

Posição de memória	Valor de Saída
0	0.5
1	0.0
2	1.0
⋮	⋮

O ponto forte do emprego de *lookup tables* para a implementação de funções em FPGA é que o *hardware* já é especializado em lidar com este tipo de estrutura (Omondi and Rajapakse, 2006). Outro ponto positivo em relação às outras técnicas é a velocidade de processamento obtida com o seu emprego, visto que o cálculo do valor da função em si se resume a um acesso de memória. Entretanto, alguns problemas surgem com a adoção dessa solução.

Um que pode limitar o sucesso da rede é o fato de ser impossível, com uma *lookup table*, representar uma função contínua com perfeição. Isso tem duas implicações diretas:

1. É preciso restringir o espaço de atuação da função de ativação.
2. É necessário discretizar e quantizar os valores de entrada e saída da função.

O item (1) pode ser solucionado restringindo-se o intervalo de ação da função de ativação para que fique entre valores aceitáveis. No caso de uma função de ativação linear (para a qual Y = X) não há necessidade de se restringir este intervalo desde que se observe os limites da representação numérica adotada (restrito pelo número de bits). Na aplicação final deste trabalho foi utilizada uma rede com funções de ativação do tipo sigmóide, como a da Figura 3.12.

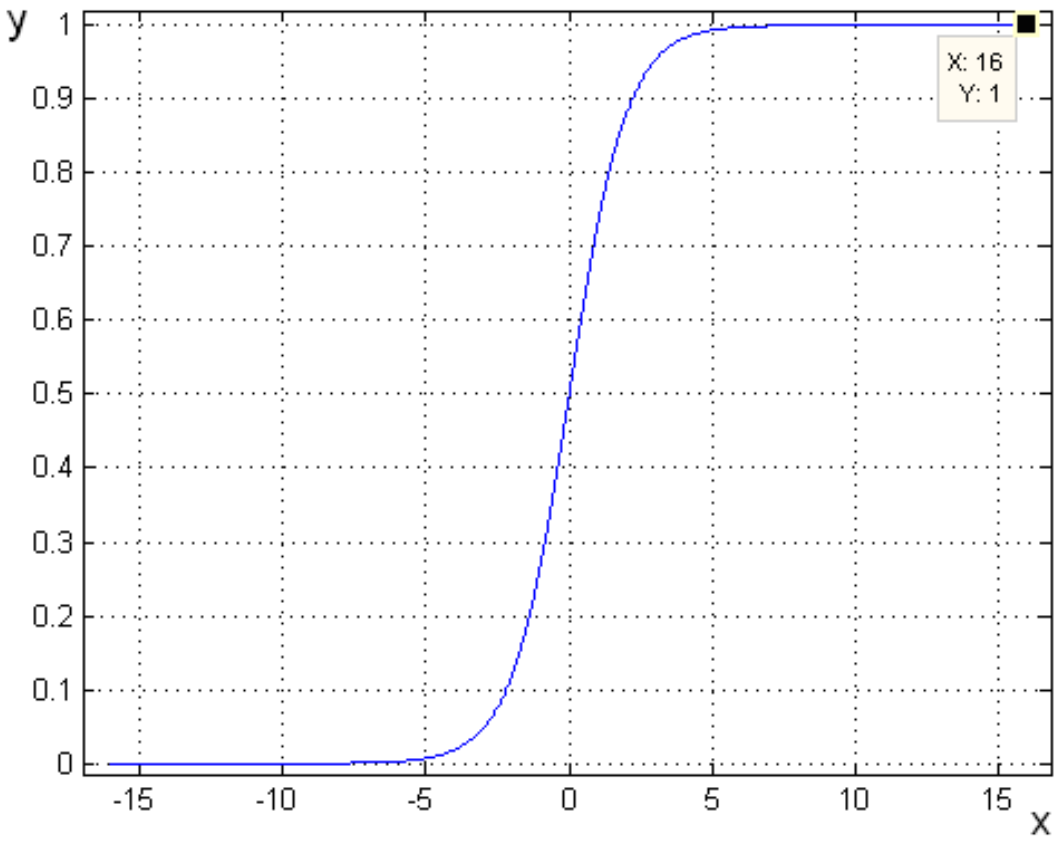

Figura 3.12: Função de ativação com intervalo reduzido

A sigmóide é caracterizada por um par de assíntotas horizontais sendo que $y \to 1$ para $x \to \infty$ e $y \to 0$ para $x \to -\infty$.

Inicialmente a atuação da função de ativação implementada em VHDL foi restrita ao intervalo de entrada -16 a +16 sendo que as saídas para entradas abaixo de -16 são assumidas como *zero* e saídas para entradas acima de +16 são assumidas como *um*.

Depara-se então com o problema de como representar esse intervalo como uma tabela. É preciso lembrar que essa representação deverá ser feita através de números binários, cuja resolução permita que a rede desempenhe seu papel de classificadora. De fato, para efeito de classificação, a resolução da *lookup table* não precisa ser muito alta. De acordo com a literatura (Savran and Unsal, 2003), 8 bits para a parte fracionária se mostraram suficientes para fornecer resultados satisfatórios.

Visto que o intervalo de entrada da função vai de -16 a +16, são necessários pelo menos 5 bits para representar a parte inteira da entrada (4 bits mais um para o sinal em complemento de dois, como pode ser visto em mais detalhes no Apêndice A.1 sobre representação em ponto

fixo). Isso resulta em um total de 13 bits para a entrada da *lookup table*. O resultado direto é que a tabela terá $2^{13} = 8192$ posições de memória, sendo cada endereço em si também um número sobre o qual será calculada a sigmóide.

Notar entretanto que surge um problema no processo de armazenamento. Na Figura 3.12 pode-se ver que X vai de -16 a +16, ou seja, vai do número mais negativo para o mais positivo. Como pode ser visto no apêndice sobre representação em ponto fixo, isso não ocorre para o caso da representação por complemento de dois. Na representação por complemento de dois as primeiras posições equivalem aos números positivos mais próximos de zero. Os valores vão aumentando sucessivamente até o máximo contemplado e abruptamente passam a representar o número mais negativo. Uma versão ilustrativa para 3 bits pode ser vista na Tabela 3.4.

Tabela 3.4: O problema do armazenamento em complemento de dois

Posição	Binário	Decimal	$Comp2_{10}$
0	000	0	0
1	001	1	1
2	010	2	2
3	011	3	3
4	100	4	-4
5	101	5	-3
6	110	6	-2
7	111	7	-1

Para contornar esse problema é preciso armazenar os dados na tabela na ordem em que a representação por complemento de dois funciona. O resultado pode ser visto na Figura 3.13.

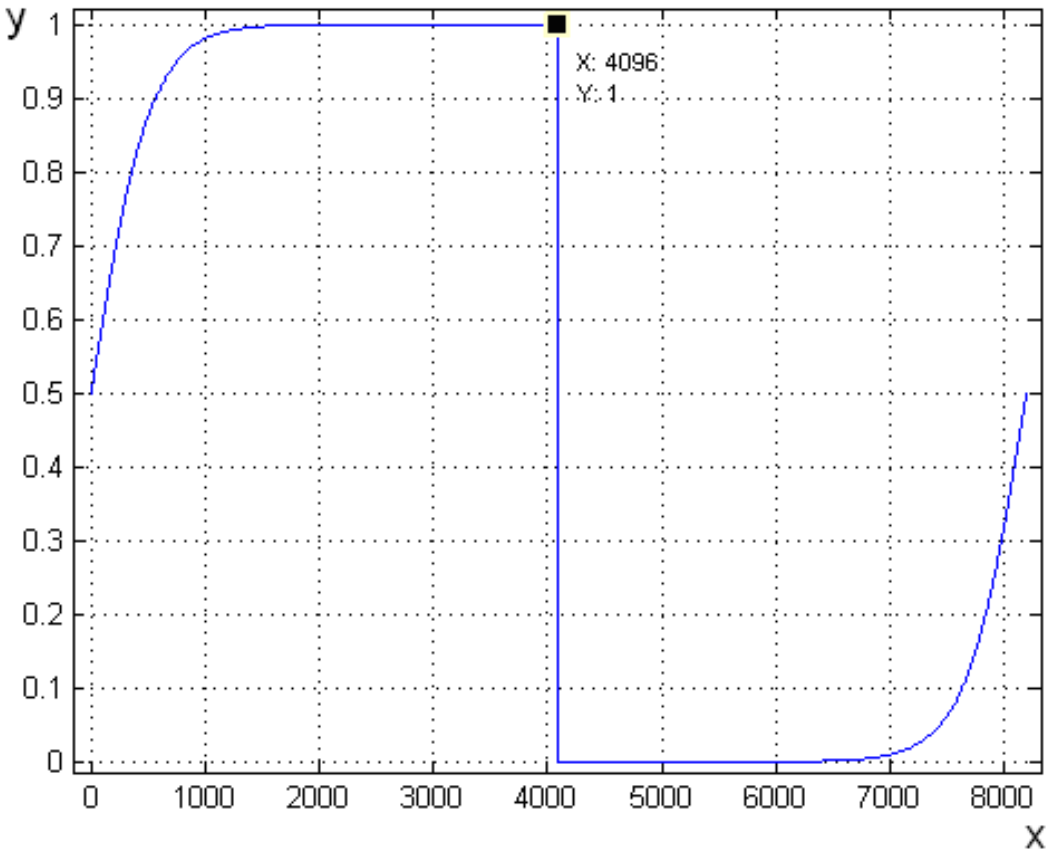

Figura 3.13: Armazenamento da função de ativação sigmóide na memória do FPGA

Como visto anteriormente, a *lookup table* é armazenada na memória na forma de um vetor de dados no qual a relação da posição do elemento e o valor do elemento é um mapeamento entre os valores de entrada e saída da função em questão, respectivamente. A implementação em VHDL da *lookup table* pode ser apreciada na Figura 3.14.

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.all;
3   use IEEE.STD_LOGIC_ARITH.ALL;
4   use IEEE.STD_LOGIC_SIGNED.ALL;
5   package LUT is
6   -- Declare constants
7
8   TYPE vector_array IS ARRAY (0 TO 8192-1) OF signed (15 DOWNTO 0);
9
10  CONSTANT LUTmemory: vector_array := (
11
12  "0000000010000000",
13  "0000000010000000",
14  "0000000010000000",
15  "0000000010000000",
16  "0000000010000001",
17  "0000000010000001",
18  "0000000010000001",
19  "0000000010000001",
                ●
                ●
                ●
8198  "0000000001111110",
8199  "0000000001111110",
8200  "0000000001111111",
8201  "0000000001111111",
8202  "0000000001111111",
8203  "0000000001111111"
8204   );
8205
8206  end LUT;
```

Figura 3.14: Estrutura da *lookup table* em VHDL

Assim, para cada X de entrada é seguido o seguinte algoritmo:

1. Se $X < -16$ então $Y = 0$
2. Se $X > +16$ então $Y = 1$
3. Senão Y = lookupTable(X)

Redução e otimização da LUT

Considerando os recursos limitados do equipamento, pode ser necessário sacrificar um pouco da precisão da função de ativação para que o consumo de *hardware* se adeque ao FPGA utilizado. Esta sessão apresenta algumas decisões de projeto sobre a LUT descrita até o momento que permitem uma redução considerável do espaço que ela ocupa no dispositivo.

Essas decisões foram tomadas tendo em mente o uso da sigmóide tradicional utilizada, não sendo diretamente aplicáveis a outras funções de ativação. De maneira sucinta, as modificações realizadas foram três:

1. Redução do intervalo de representação
2. Aumento do passo de amostragem
3. Armazenamento apenas dos bits de interesse para cada valor

O intervalo de amostragem foi reduzido do anterior [-16, +16] para [-6, +6]. O que motivou essa redução foi o fato de que a resolução que os oito bits da parte fracionária fornece acaba gerando um erro de quantização de 2^{-8} na representação de cada valor, como pode ser visto na Equação 3.4. Sendo y_0 e y_1 os valores de saída da função sigmóide para as entradas -6 e +6 respectivamente, considerando uma representação de 16 bits como a utilizada.

A distância de y_0 até a assíntota em zero e de y_1 até a assíntota em um é a mesma, e são ambas menores do que pode representar o bit menos significativo cujo valor máximo é 2^{-8} .

$$\begin{aligned} y_0 &= \sigma(-6) \\ y_1 &= \sigma(+6) \\ d_0 &= y_0 - 0 \\ d_1 &= 1 - y_1 \\ \boxed{d_0 = d_1 < 2^{-8}} & \end{aligned} \tag{3.4}$$

A representação numérica utilizada não é capaz de representar mudanças de valor de saída da sigmóide fora do intervalo acima descrito o que justifica, para este caso em particular, o novo intervalo adotado. Neste ponto o endereço para se acessar algum valor da LUT passou a ser então composto por 12 bits.

A seguir, ao se aumentar o passo de amostragem da função, foram utilizados apenas os 8 bits mais significativos dos 12 bits previamente mencionados. O resultado é um endereço de 8 bits para a entrada da *lookup table*, o que permite uma tabela com $2^8 = 256$ posições de memória, uma redução considerável em comparação com as 8192 anteriores.

A maneira que a função de ativação sigmóide reduzida deve ser armazenada na memória fica como na Figura 3.15.

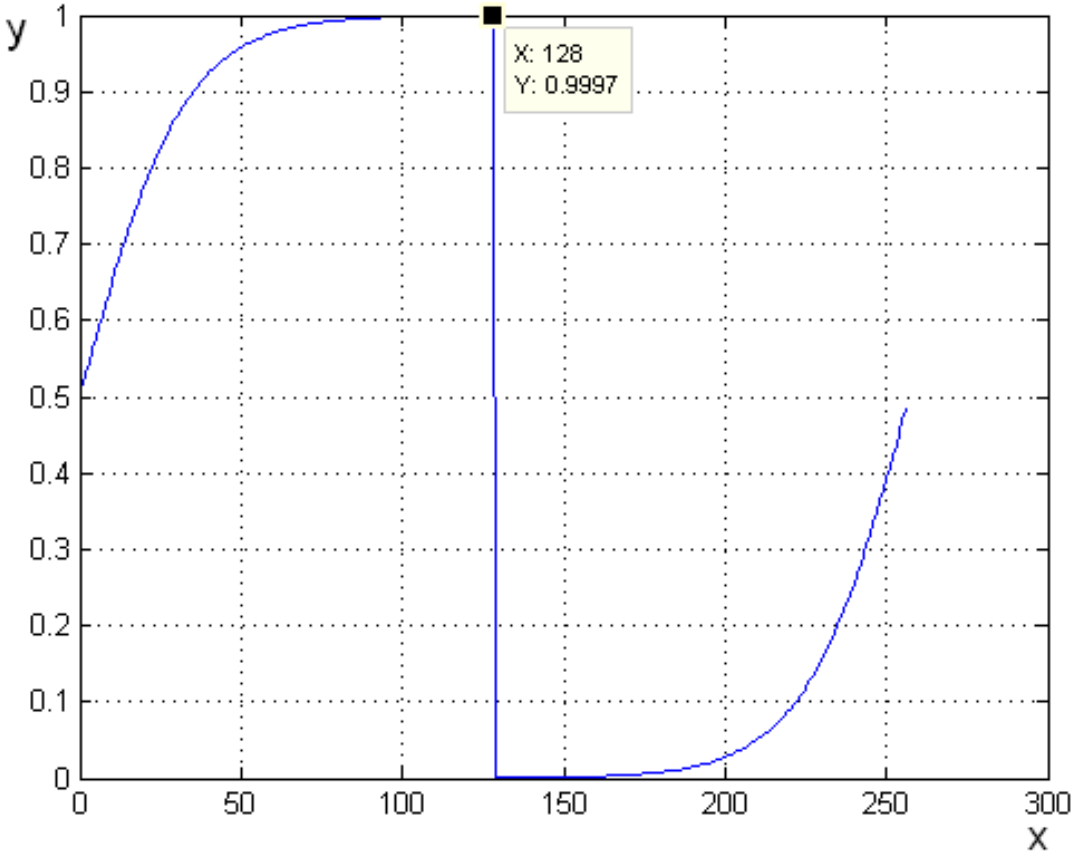

Figura 3.15: Armazenamento da função de ativação sigmóide reduzida na memória do FPGA

Por último, só foram armazenados na tabela os bits que contém informação relevante para a função em questão. Como a sigmóide produz saídas variando entre 0 e 1, a saída binária será sempre entre '00000000.00000000' e '00000001.00000000'.

Como a função sigmóide não produz valores negativos (que exigiriam o uso dos bits mais significativos devido à representação por complemento de 2) nem valores maiores do que um, os 7 bits mais significativos, do total de 16, serão sempre zero. Por conta disto, é possível armazenar apenas os 9 bits menos significativos na tabela sem que haja perda alguma de desempenho.

As modificações descritas podem ser vistas no Apêndice C.1, que contém a implementação completa da função de ativação.

O microcontrolador MicroBlaze™foi utilizado para realizar a comunicação entre o computador e o FPGA, fornecendo valores de entrada para a função de ativação implementada.

O resultado comparativo entre a função de ativação otimizada produzida e o valor calculado esperado pode ser visto na Figura 3.16.

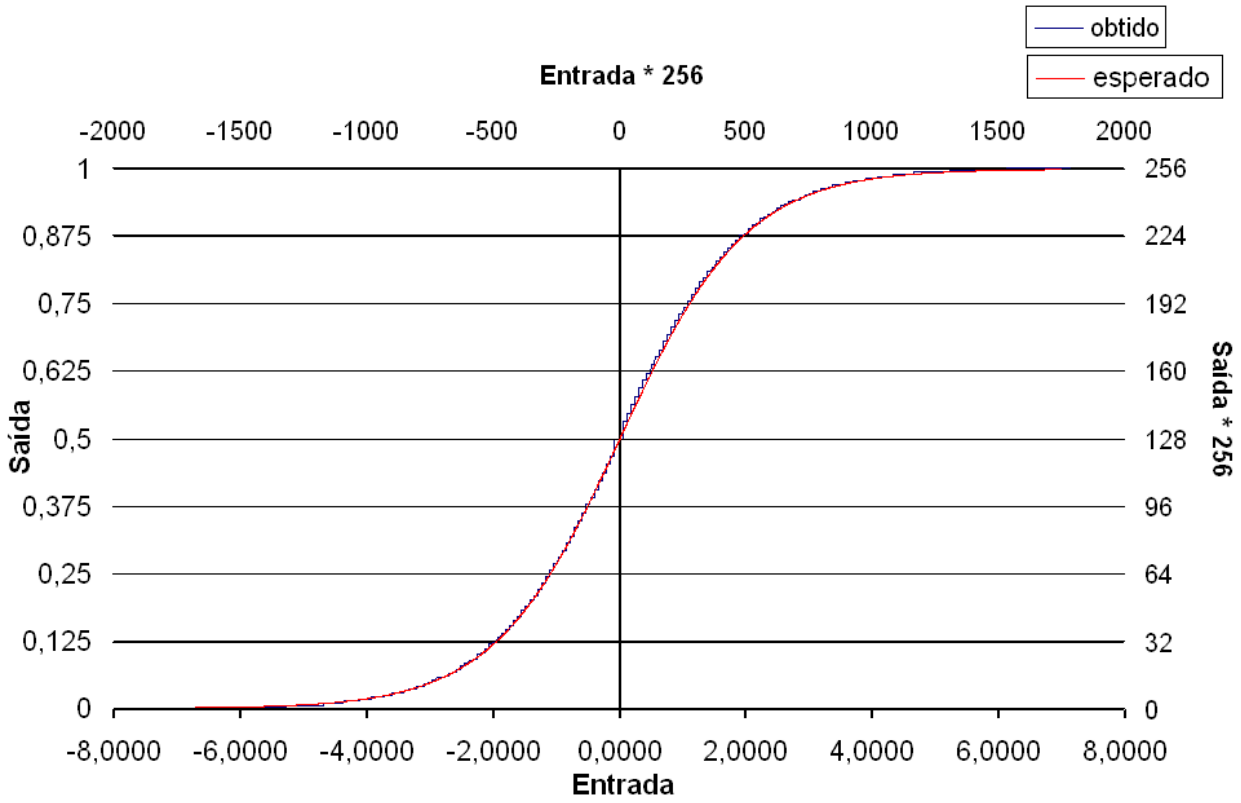

Figura 3.16: Comparação entre a sigmóide produzida e a esperada

A curva em azul se refere à saída da função implementada em *hardware* para uma entrada variando de -7 a 7.

O eixo vertical contém a escala da saída esperada à esquerda e a da saída obtida à direita, lembrando que os valores obtidos estão multiplicados por 2^8. O eixo horizontal contém a escala da entrada original em baixo e a escala da entrada multiplicada por 2^8, que servirá como endereço para o acesso à LUT.

Observar que embora a discretização seja visível ao longo da curva, o comportamento sigmoidal esperado foi produzido. O erro quadrático médio (MSE) resultante foi de $1,7 \cdot 10^{-5}$.

A estrutura da rede por componentes

A organização da rede neural obtida após a sua implementação em VHDL segue a hierarquia observada anteriormente no capítulo sobre redes neurais. Sua estrutura é composta por componentes que se interconectam para formar a rede como um todo. Um diagrama com os componentes principais pode ser visto na Figura 3.17.

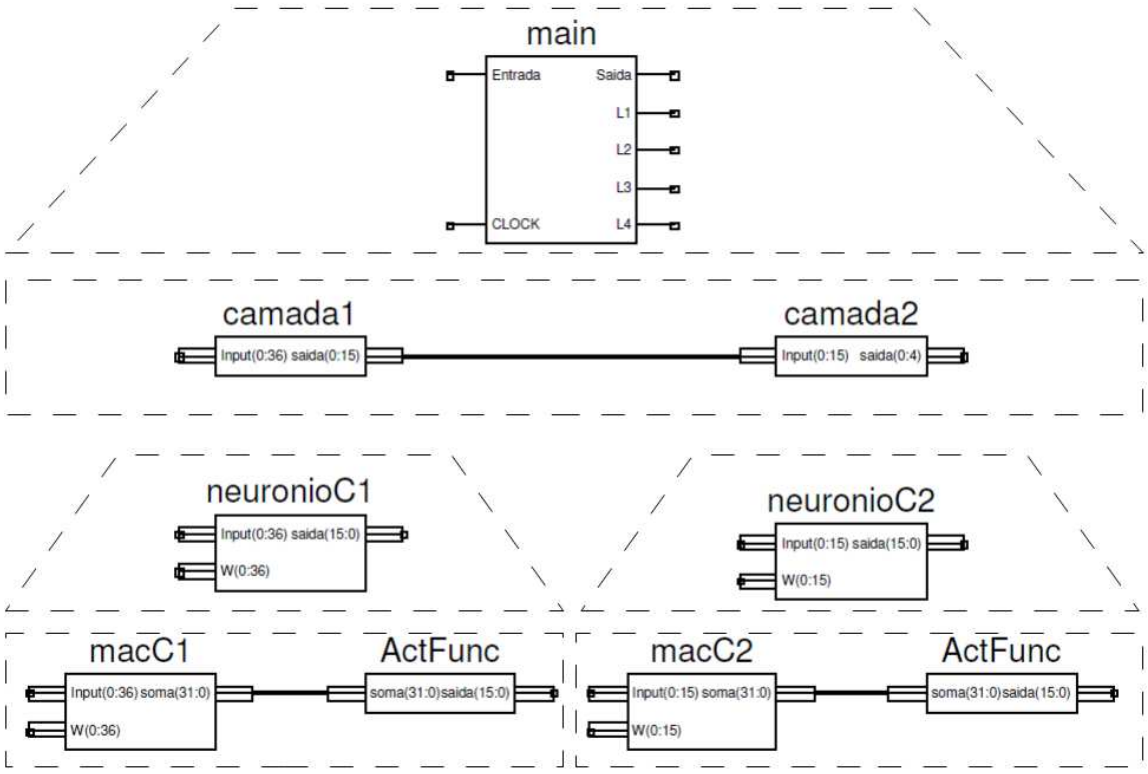

Figura 3.17: Componentes da rede neural implementada em *hardware*

O componente mais básico é o neurônio, sendo cada um deles conectados ao vetor de entrada ou aos neurônios da camada seguinte. Cada neurônio possui dois subcomponentes; MAC (multiplicador e acumulador), e a aplicação da função de ativação, como descritos anteriormente.

O resultado obtido é algo similar ao da Figura 3.18, que exemplifica a rede de duas camadas com 15 neurônios na primeira camada e quatro neurônios na camada de saída.

Observar como a hierarquia dos componentes segue a estrutura da rede neural previamente vista na Figura 2.11. Os códigos em VHDL dos componentes da rede neural podem ser vistos no Apêndice B.1.

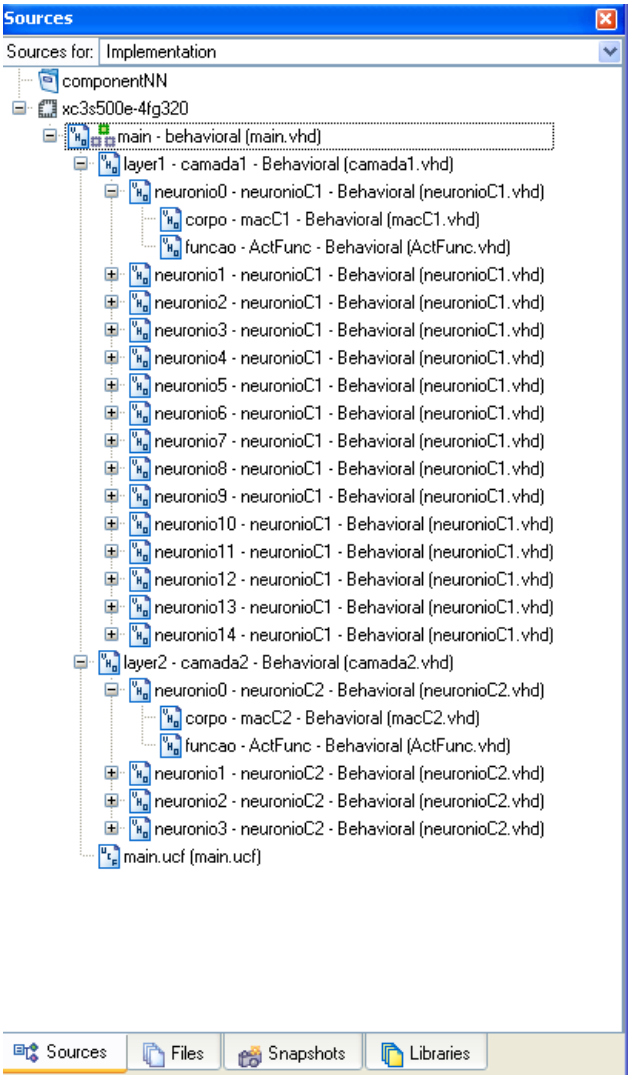

Figura 3.18: Estrutura da rede neural por componentes em VHDL

3.3.2 Modelo Matricial

Neste trabalho chama-se de *modelo matricial* a implementação em *hardware* de uma rede neural paralela que, diferentemente do *modelo por componentes* visto na seção anterior, utiliza diretamente a notação matricial para a programação em VHDL. Houve duas motivações importantes para o desenvolvimento desta forma alternativa de programação da rede neural:

1. Comparação entre diferentes modelos
2. Necessidade de se reduzir a área do FPGA ocupada pela rede

De fato, notou-se uma redução da área em *hardware* ocupada pelo código, como pode ser visto no capítulo de resultados.

Nas seções seguintes serão apresentados os desenvolvimentos e estruturas deste tipo de modelo.

Soma Ponderada

No caso matricial, a soma ponderada ocorre através da multiplicação direta da matriz de pesos, similar à apresentada na Equação 2.1, pelo vetor de entradas.

A soma ponderada que ocorre durante a multiplicação matricial acima citada é feita de maneira paralela, não havendo a necessidade de se acumular o resultado de cada multiplicação serialmente para só então se calcular a soma.

Um diagrama do processo de multiplicação e acumulação (MAC) pode ser visto na Figura 3.19.

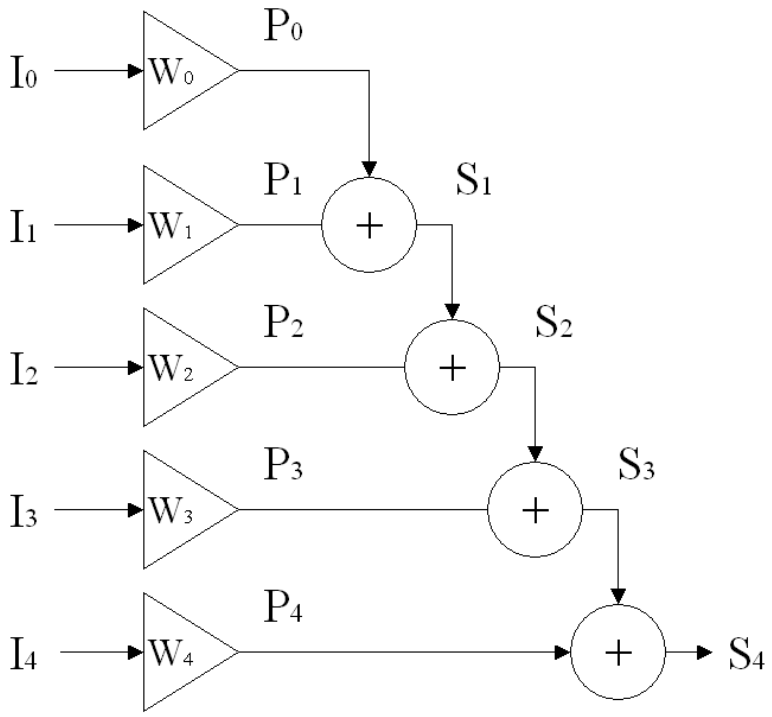

Figura 3.19: Estrutura do processo de multiplicação e acumulação

Cada entrada I é multiplicada por um coeficiente W produzindo um produto P. O produto P_0 e o P_1 por exemplo são somados então produzindo uma soma que está representada na figura como S_1. Essa soma é então acumulada com o produto P_2 seguinte e assim sucessivamente.

Importante lembrar que, para o caso da rede neural, os pesos de uma camada estão armazenados em uma matriz de ponderação. A matriz de pesos de uma determinada camada fica então armazenada em uma variável (*signal*) e seu acesso é feito por endereçamento, de acordo com a Equação 3.5.

$$W(linha, coluna) = W'(coluna + linha * nLinhas) \tag{3.5}$$

Sendo *nLinhas* o número de linhas da matriz *W*, *W* a matriz de pesos armazenada em memória no formato matricial e *W'* a mesma matriz armazenada de maneira linear.

Para tornar o processo mais claro apresenta-se a seguir o caso de uma matriz de ponderação W de dimensão 3x3 e um vetor de entradas I com 3 elementos, lembrando que o acesso ao primeiro elemento de um vetor em VDHL começa com o índice zero:

$$\begin{bmatrix} W_{0,0} & W_{0,1} & W_{0,2} \\ W_{1,0} & W_{1,1} & W_{1,2} \\ W_{2,0} & W_{2,1} & W_{2,2} \end{bmatrix} \cdot \begin{bmatrix} I_0 \\ I_1 \\ I_2 \end{bmatrix} \tag{3.6}$$

Os valores dos produtos para cada multiplicação entre vetores precisam ser acumulados de maneira paralela. Não há necessidade de se criar um vetor de somatórios para guardar as acumulações parciais vistas na Figura 3.19. P_1 pode armazenar o seu próprio produto já acumulado com P_0. O resultado para o primeiro neurônio da rede equivalente é como se segue:

$$\begin{aligned} P_{0,0} &\leftarrow W_{0,0} \cdot I_0 \\ P_{0,1} &\leftarrow P_{0,0} + W_{0,1} \cdot I_1 \\ P_{0,2} &\leftarrow P_{0,1} + W_{0,2} \cdot I_2 \end{aligned} \tag{3.7}$$

Para três neurônios, as operações necessárias são as seguintes, verificando que a primeira coluna de operações equivale às do primeiro neurônio, a segunda para o segundo e a terceira coluna descreve as operações para o último neurônio.

$$\begin{array}{lll} P_{0,0} \leftarrow W_{0,0} \cdot I_0 & P_{1,0} \leftarrow W_{1,0} \cdot I_0 & P_{2,0} \leftarrow W_{2,0} \cdot I_0 \\ P_{0,1} \leftarrow P_{0,0} + W_{0,1} \cdot I_1 & P_{1,1} \leftarrow P_{1,0} + W_{1,1} \cdot I_1 & P_{2,1} \leftarrow P_{2,0} + W_{2,1} \cdot I_1 \\ P_{0,2} \leftarrow P_{0,1} + W_{0,2} \cdot I_2 & P_{1,2} \leftarrow P_{1,1} + W_{1,2} \cdot I_2 & P_{2,2} \leftarrow P_{2,1} + W_{2,2} \cdot I_2 \end{array} \tag{3.8}$$

Essa notação permite que os resultados sejam armazenados em uma matriz de produtos, cuja última coluna possuirá os valores referentes à soma ponderada final:

$$\begin{bmatrix} P_{0,0} & P_{0,1} & P_{0,2} \\ P_{1,0} & P_{1,1} & P_{1,2} \\ P_{2,0} & P_{2,1} & P_{2,2} \end{bmatrix} \rightarrow P_{\dots,2} = Sp \tag{3.9}$$

Sendo P os produtos acumulados e Sp um vetor 3x1 com as somas ponderadas para a camada atual.

O passo seguinte é fornecer essa soma ponderada à função de ativação para que a mesma produza a saída da camada. O funcionamento da função de ativação para o modelo matricial é o mesmo descrito para o caso por componentes. A diferença está no fato de que para o caso matricial a função de ativação não é um componente, mas sim uma função declarada no corpo principal do projeto.

A estrutura da rede matricial

A rede fica organizada em apenas um arquivo VHD principal tendo mais dois arquivos VDH de suporte: um contendo as constantes principais do projeto e outro contendo a *lookup table* referente à sigmóide.

Sua estrutura pode ser mais bem compreendida através do pseudocódigo a seguir:

1. Leitura de entrada
2. Feedforward
   (a) MAC camada 1
   (b) Aplicar a função de ativação nas somas ponderadas da camada 1
   (c) Adiciona o elemento bias à saída da camada 1
   (d) MAC camada 2
   (e) Aplicar a função de ativação nas somas ponderadas da camada 2
   (f) Adiciona o elemento bias à saída da camada 2
3. Escrita de saída

Notar, no entanto que a ordem em que os itens são codificados é irrelevante para o processo visto que todos estão ocorrendo em paralelo dentro do *hardware*. Os códigos em VHDL dos componentes da rede neural com modelo matricial podem ser vistos no Apêndice B.2.

3.3.3 Metaprogramação VHDL em Matlab

O desenvolvimento de uma rede neural que atenda aos requisitos de projeto é um processo laborioso.

Embora existam técnicas de definição automática do número de neurônios na camada intermediária para MLPs, como a empregada neste trabalho, a decisão quanto ao número de neurônios da camada intermediária acaba sendo tomada com base em testes com vários números de neurônios diferentes.

Técnicas de treinamento diferentes produzem resultados melhores ou piores para casos particulares e cada uma delas tem um número considerável de constantes a serem definidas e ajustes finos a serem feitos.

Somado a isso está a dificuldade de se implementar a rede em linguagem VHDL após seu treinamento que devido a necessidade de ser feita diversas vezes acaba se tornando proibitivamente laboriosa.

Para agilizar o processo e permitir testes com diversas topologias de rede em tempo adequado, foi escrito para o presente trabalho um código em Matlab cuja função é, em um exercício de metaprogramação, escrever todo o código VHDL necessário para a implementação do projeto.

Um diagrama do processo pode ser visto na Figura 3.20.

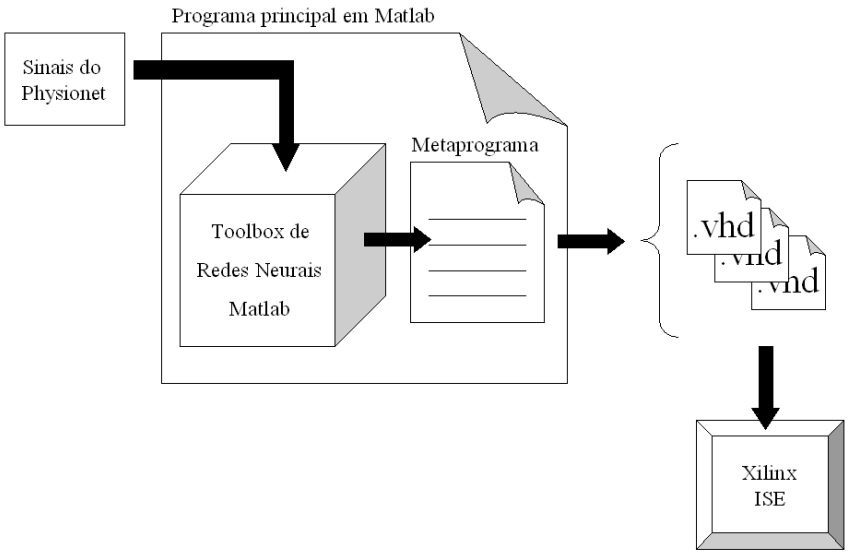

Figura 3.20: Diagrama do processo de escrita de programas VHDL por um metaprograma em Matlab

O programa principal em Matlab contém todo o tratamento dos sinais obtidos no banco de dados Physionet (Goldberger et al., e 13) e utiliza o toolbox de redes neurais do próprio Matlab para realizar o treinamento da rede.

Uma vez pronta, a rede é lida pelo metaprograma que escreve as linhas de código referentes à implementação em FPGA em arquivos *.vhd* na própria pasta de projeto do ambiente de desenvolvimento ISE. A atualização do projeto no software da Xilinx é então imediata.

Capítulo 4

Resultados

4.1 Detecção do complexo QRS

Após a detecção dos picos suas posições são remapeadas para a dimensão do sinal original, e produzem o resultado que pode ser apreciado na Figura 4.1.

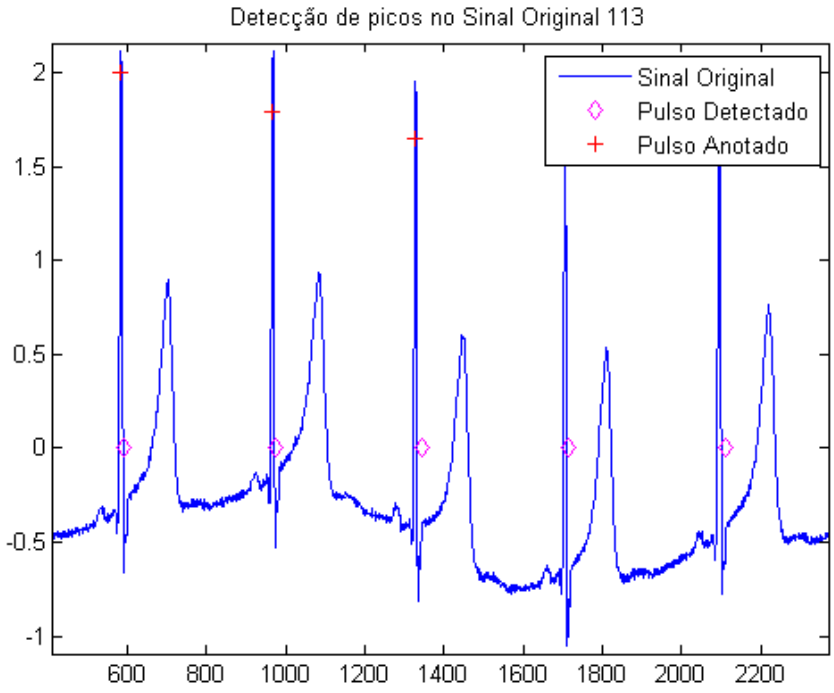

Figura 4.1: Comparação entre os pulsos detectados e os anotados

Como previamente mencionado, a análise foi feita com os sinais do banco de dados MIT-BHI Arrythmia para a MVII. Para verificar o desempenho do algoritmo, utilizou-se um

número maior de sinais do banco de dados [1]. O desempenho do processo de detecção de pulsos pode ser visto em números absolutos na Figura 4.2, sendo fP referente aos falsos positivos, fN aos falsos negativos e vP aos verdadeiros positivos.

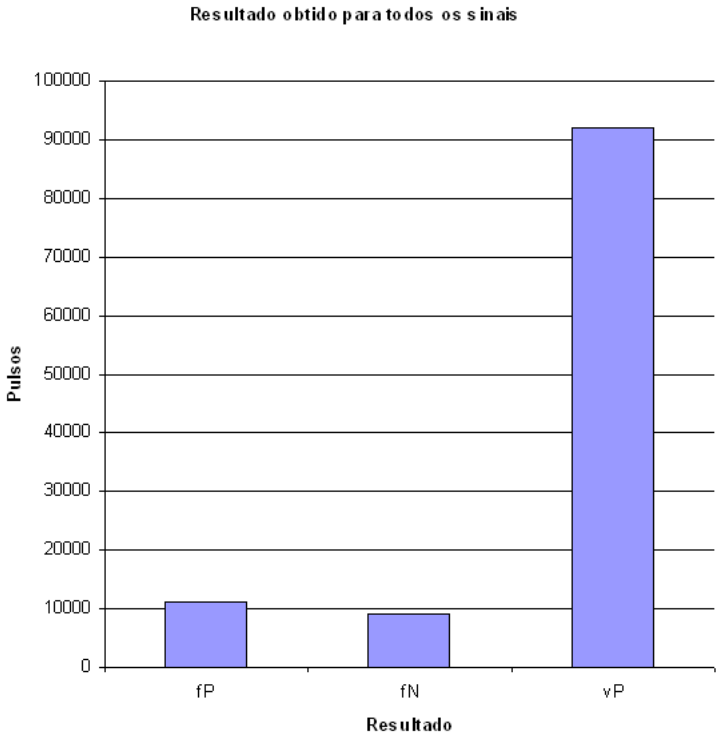

Figura 4.2: Resultado do processo de detecção para todos os sinais

O desempenho total pode então ser sintetizado pela precisão e sensibilidade do sistema, como relatado na Tabela 4.1.

Sensibilidade	Precisão
91,26%	89,58%

Tabela 4.1: Resultado médio do desempenho do algoritmo de detecção

O cálculo da sensibilidade é feito dividindo-se o número de verdadeiros positivos pela soma do número de verdadeiros positivos com o número de falsos negativos.

$$S = \frac{vP}{vP+fN}$$

[1]Sinais utilizados: 100 101 103 105 106 107 108 109 111 112 113 114 115 116 117 118 119 121 122 123 124 200 201 202 203 205 207 208 209 214 221 222 223 228 230 231 232 233 234

Uma alta sensibilidade indica que o algoritmo reconhece a maioria dos pulsos verdadeiros enquanto que uma baixa sensibilidade indica que um número considerável de pulsos não está sendo identificado.

O cálculo da precisão por sua vez é feito dividindo-se o número de verdadeiros positivos pela soma do número de verdadeiros positivos com o número de falsos positivos.

$$P = \frac{vP}{vP+fP}$$

Uma alta precisão indica também que a maioria dos pulsos está sendo identificada, enquanto que uma baixa precisão sugere que há um número expressivo de identificações erradas de pulsos, ou seja, picos que não são referentes às ondas R estão sendo identificados como tal.

O resultado final foi uma capacidade alta de detecção de pulsos, embora não tão expressiva quanto a de trabalhos mais recentes na literatura. Para alguns sinais houve um desempenho maior do que para outros: para o sinal 100, por exemplo, obteve-se uma sensibilidade e precisão ambas de 100%, ou seja, todos os pulsos foram detectados corretamente. Já para o caso do sinal 232, obteve-se uma sensibilidade de 91% e uma precisão de 68%, ou seja, embora o algoritmo tenha sido capaz de identificar a grande maioria dos pulsos, houve um número grande de falsos positivos que vieram a reduzir a precisão geral para este caso.

Importante ressaltar no entanto que os melhores resultados são fruto da utilização de sistemas híbridos, que utilizam mais de uma técnica para a detecção confiável do complexo QRS.

4.2 Desempenho da Classificação em Matlab

Para efeito de avaliação do desempenho da rede neural produzida utilizaram-se todos os pulsos dos sinais previamente relatados na Tabela 3.1. Cada pulso foi pré-processado como descrido na Seção 3.2 e em seguida foram fornecidos à rede neural.

O resultado das classificações pode ser visto nas Figuras de 4.3 a 4.6, que mostram o processo de classificação dos pulsos fornecidos à rede completa para a avaliação dos mesmos.

As figuras possuem quatro partes denominadas S1, S2, S3 e S4. Cada uma representa o valor da respectiva saída da rede quando uma amostra, no caso o pulso pré-processado, lhe é apresentada. Notar por exemplo que ao longo do eixo X para a Figura 4.3, S1 assume o valor 1 na maioria dos casos, enquanto que S2, S3 e S4 tendem a apresentar o valor 0, o que caracterizam classificações do tipo normal.

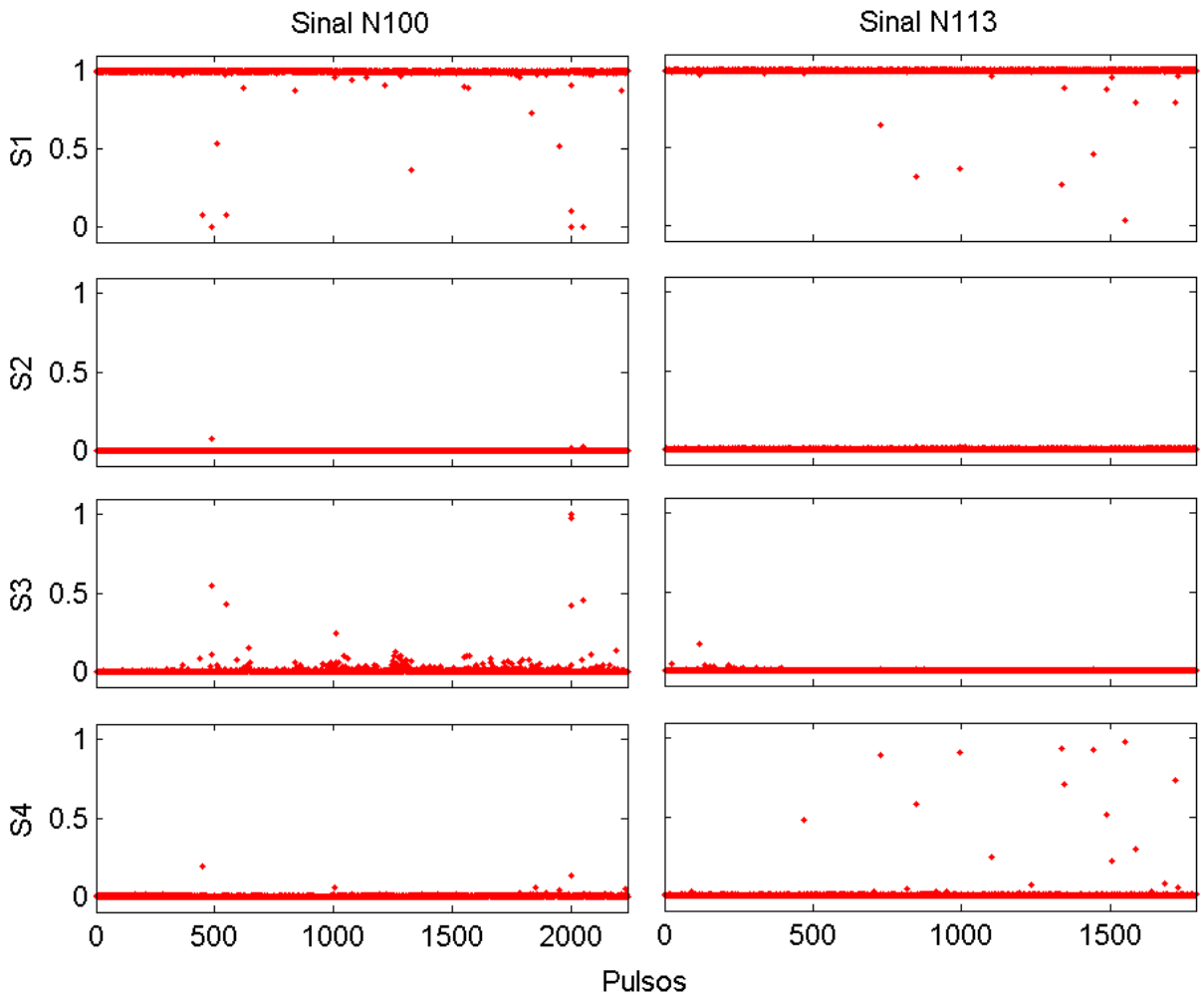

Figura 4.3: Resultado do processo de classificação dos pulsos normais

Uma maneira de sintetizar os resultados obtidos para todos os sinais é com o uso de uma *matriz de confusão*, como pode ser vista na Tabela 4.2, para o caso completo e na Tabela 4.3 para o caso da rede neural reduzida, onde Nv, Vv, Rv e Lv correspondem à natureza verdadeira do pulso, Np, Vp, Rp e Lp correspondem a como eles foram positivamente classificados e indef é a contagem de casos que ficaram indefinidos.

Verificar que a rede neural conseguiu separar os quatro grupos de anomalias com sucesso. Dos 4025 pulsos normais, 4007 foram corretamente classificados como sendo normais (N), 16 foram incorretamente classificados como sendo do tipo LBBB (L - Bloqueio de Ramo Esquerdo) e apenas dois foram erroneamente classificados como RBBB (R - Bloqueio de Ramo Direito). Para o desempenho das redes em Matlab não ocorreram casos de indefinição. Para todos os outros casos os números de classificações errôneas foram igualmente baixos.

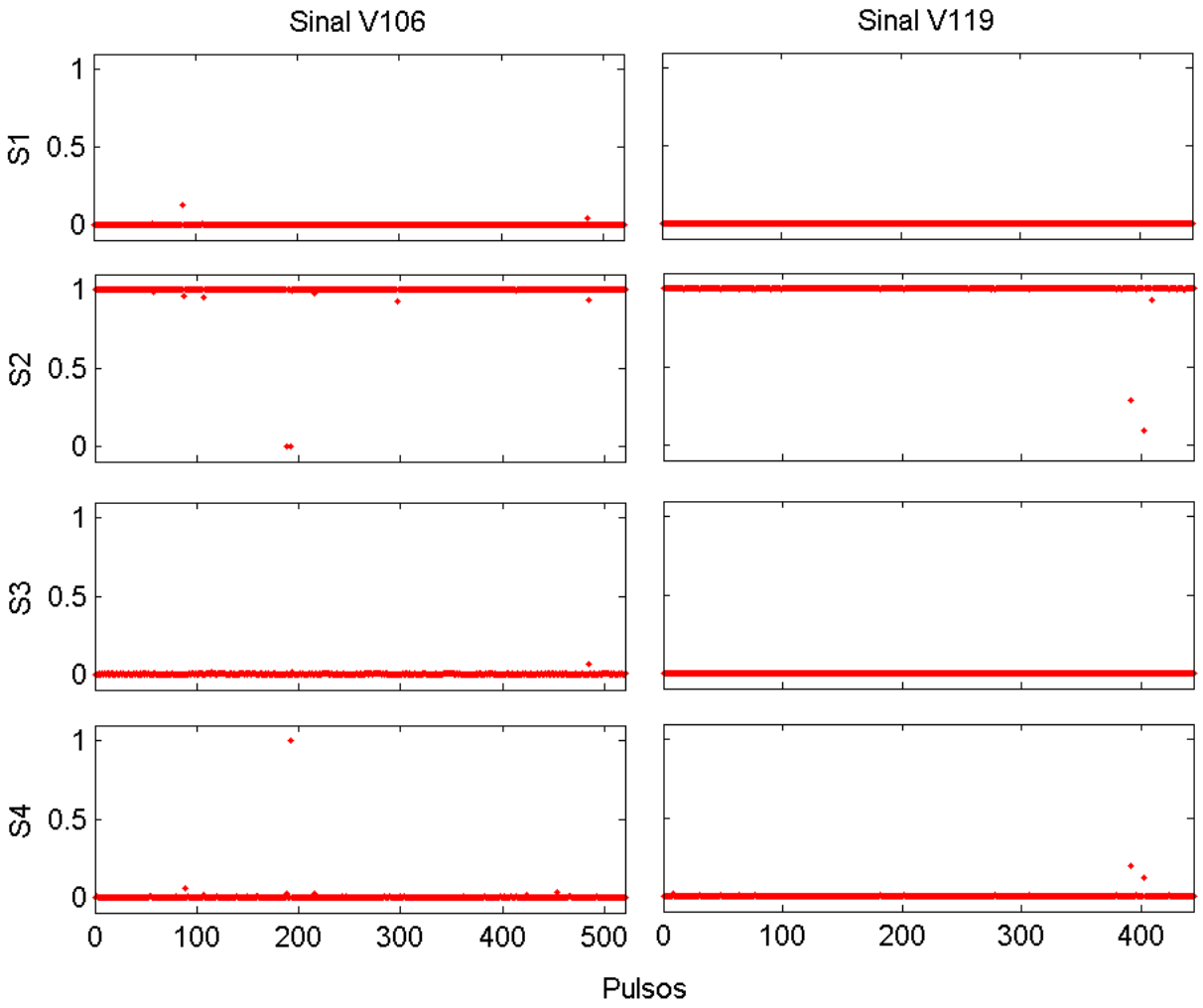

Figura 4.4: Resultado do processo de classificação dos pulsos com contração ventricular prematura

Tabela 4.2: Matriz de confusão com os resultados da classificação da rede completa

	Nv	Vv	Rv	Lv
Np	4007	0	5	8
Vp	0	961	5	8
Rp	2	0	3407	4
Lp	16	3	1	4104
indef	0	0	0	0
Total	4025	964	3418	4124

A matriz de confusão é importante, pois mostra não apenas a quantidade de acertos e erros, mas também a maneira como cada erro ocorre. Verifica-se, por exemplo, que pulsos do tipo LBBB se confundem muito mais com os do tipo normal do que com os outros casos.

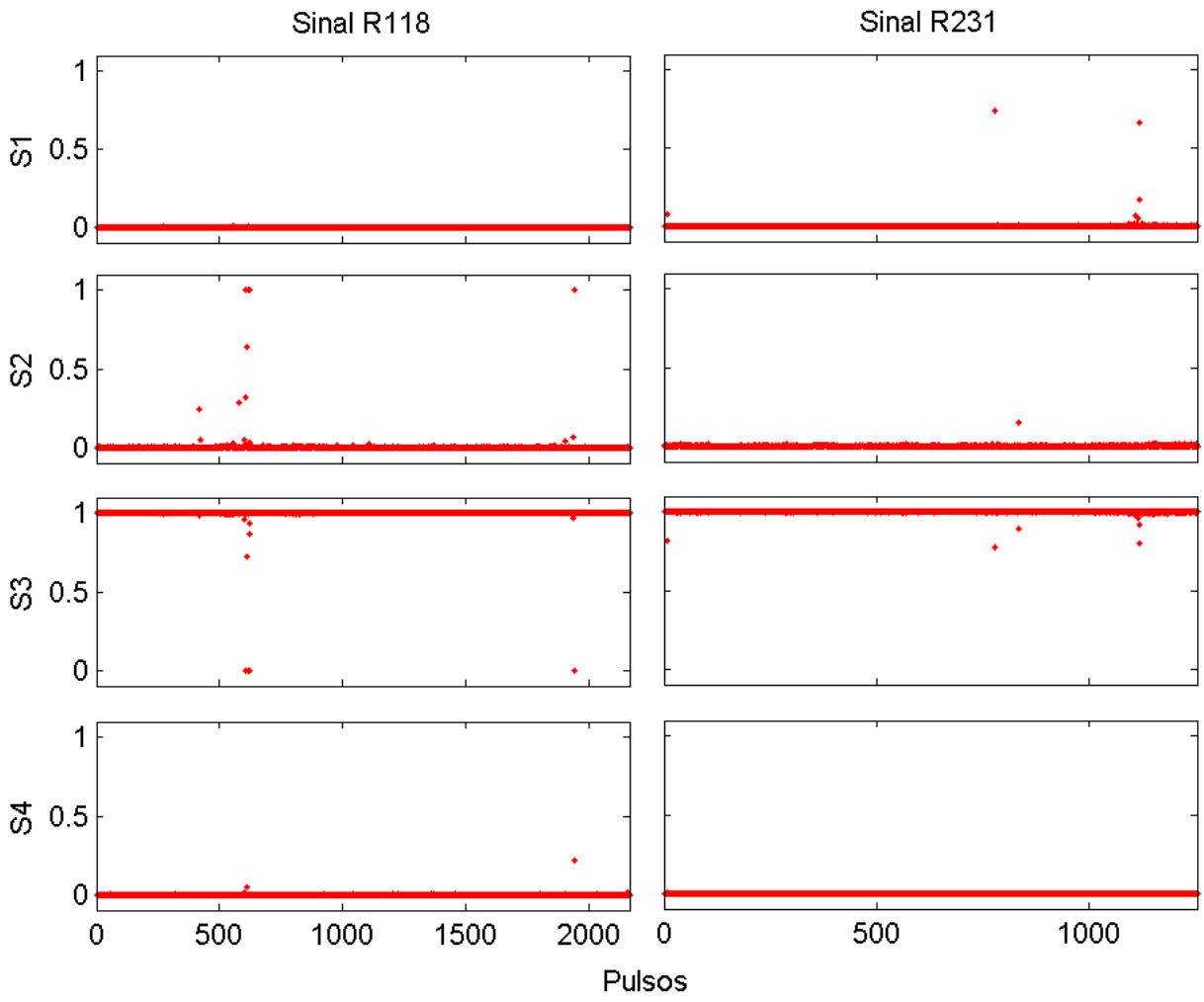

Figura 4.5: Resultado do processo de classificação dos pulsos com bloqueio de ramo direito

Tabela 4.3: Matriz de confusão com os resultados da classificação da rede reduzida

	Nv	Vv	Rv	Lv
Np	3915	1	116	20
Vp	0	938	22	234
Rp	101	1	3267	24
Lp	9	24	13	3846
indef	0	0	0	0
Total	4025	964	3418	4124

Observar que o desempenho da rede reduzida foi inferior ao da rede completa. Na rede reduzida, vários casos de bloqueio de ramo direito acabaram sendo classificados como normais, e vários com bloqueio de ramo esquerdo foram classificados como contração ventricular prematura.

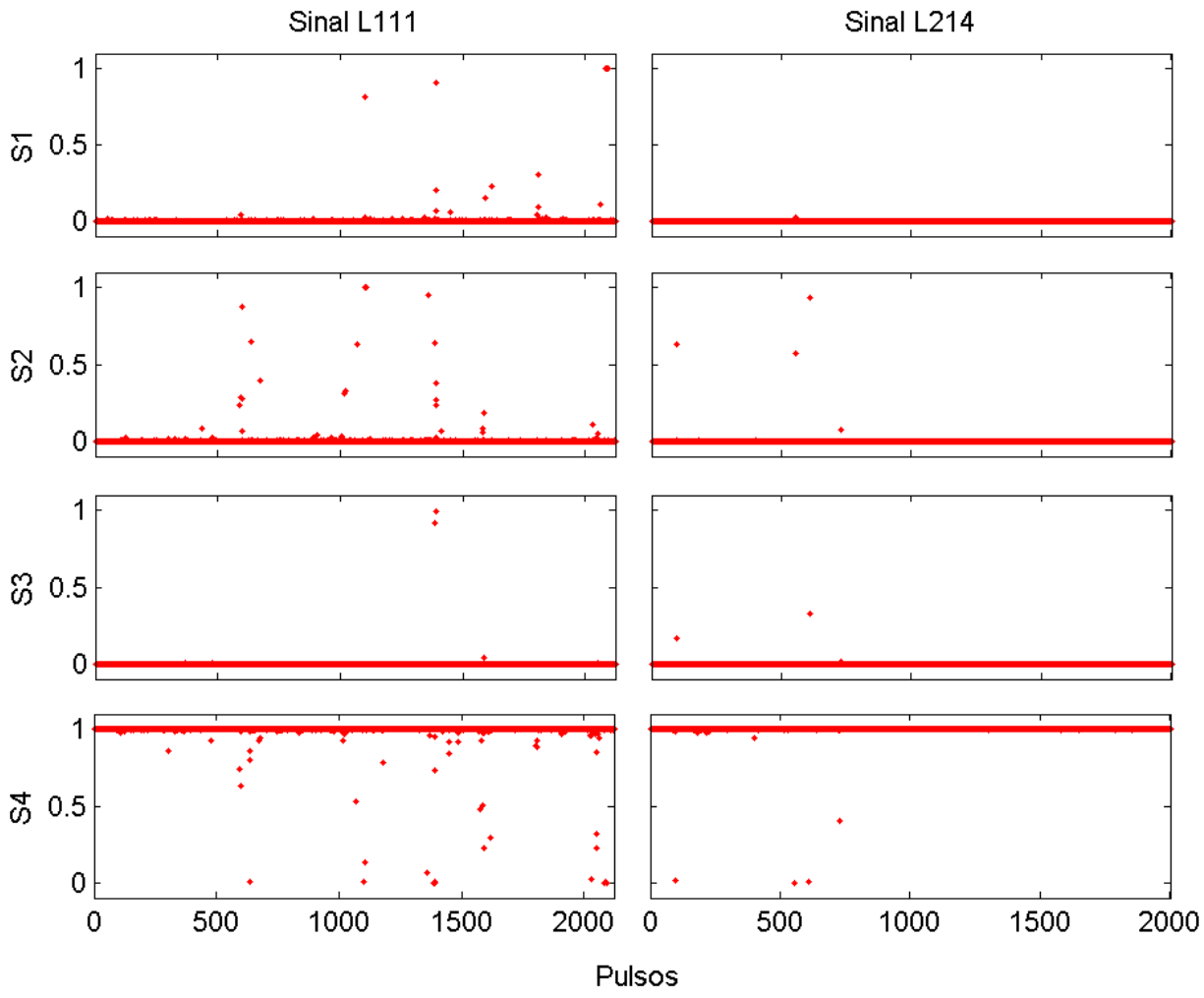

Figura 4.6: Resultado do processo de classificação dos pulsos com bloqueio de ramo esquerdo

4.3 Desempenho da Classificação em *hardware*

4.3.1 Resultado das Simulações

Os sinais foram fornecidos ao sistema e cada pulso foi classificado pela rede neural desenvolvida. As saídas L1, L2, L3 e L4, como vistas no bloco *main* da Figura 3.17, estão conectadas diretamente aos LEDs 1, 2, 3 e 4 da placa de desenvolvimento.

A Figura 4.7 demonstra o caso em que 4 tipos de anomalias são fornecidos à rede neural por componentes. As entradas foram do tipo N100 (pulso normal proveniente do sinal 100), V119 (pulso com contração ventricular prematura, proveniente do sinal 119), R118 (pulso com bloqueio de ramo direito, proveniente do sinal 118) e L214 (pulso com bloqueio de ramo esquerdo, proveniente do sinal 214).

Resultado idêntico foi obtido na simulação da rede neural modelo matricial, como pode ser visto na Figura 4.8, lembrando que tanto na saída do modelo por componentes (saidaC2) quanto na saída da rede com modelo matricial (s2), um resultado decimal 256 equivale a **1** em ponto fixo de 16 bits com 8 bits para a parte inteira e 8 para a parte fracionária.

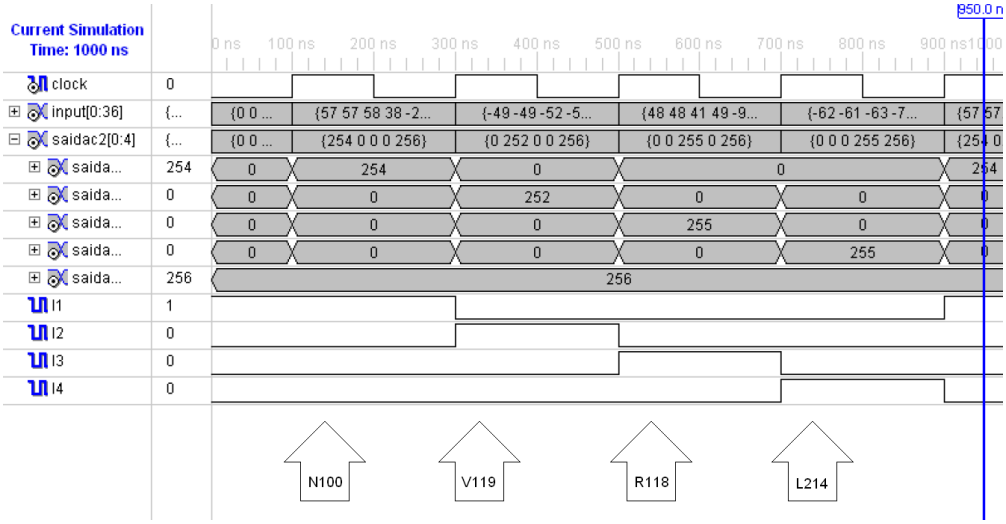

Figura 4.7: Resultado da simulação para a identificação de sinais com a rede neural por componentes

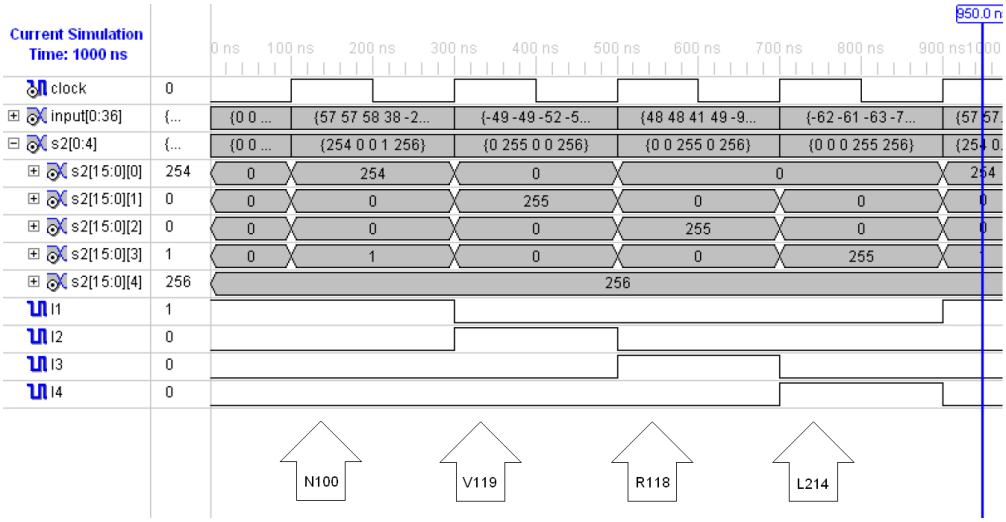

Figura 4.8: Resultado da simulação para a identificação de sinais com a rede neural matricial

4.3.2 Resultados do Sistema Embarcado

Com o intuito de completar a proposta de trabalho apresentada na Seção 1.1, buscou-se implementar a rede neural no FPGA e estabelecer um canal de comunicação com o PC externo. Devido às restrições de recursos do *kit* utilizado, foi dada preferência à implementação do modelo de rede neural matricial reduzida com a função de ativação vista na Seção 3.3.

O primeiro passo foi adicionar a rede neural em questão como um dispositivo a ser acessado pelo microcontrolador. Feito isto, foi feita a tentativa de sintetizar o projeto para que o mesmo fosse gravado no FPGA. O *software* da Xilinx acusou um excesso no uso dos recursos do *kit*. Para este caso, o projeto como um todo, que inclui tanto a implementação da rede neural quanto do microcontrolador Microblaze, estava consumindo 122% das *Slices*, 112% das LUTs de 4 entradas e 135% dos multiplicadores disponíveis. Para solucionar o problema foram tomadas algumas medidas que serão explicadas a seguir.

Como visto na Seção 3.3, uma rede neural é essencialmente composta por operações matriciais de multiplicação e aplicação de uma função de ativação. Grande parte do potencial de paralelismo da rede neural está na fase de multiplicação e acumulação o que demanda do dispositivo no qual ela está operando a capacidade de lidar com um número considerável de multiplicações concorrentes.

Considerando o número de multiplicadores disponíveis no FPGA do *kit* utilizado, foi necessário modificar a maneira como os dados de entrada são processados pela rede. Um diagrama simplificado de como o processamento dos vetores de entrada foi modificado pode ser visto na Figura 4.9, onde Sp é uma acumulador para a soma ponderada de um determinado neurônio, MULT é o processo de multiplicação e *FSL_clk* é o *CLOCK* que rege a transmissão dos dados para a rede neural através do microcontrolador Microblaze.

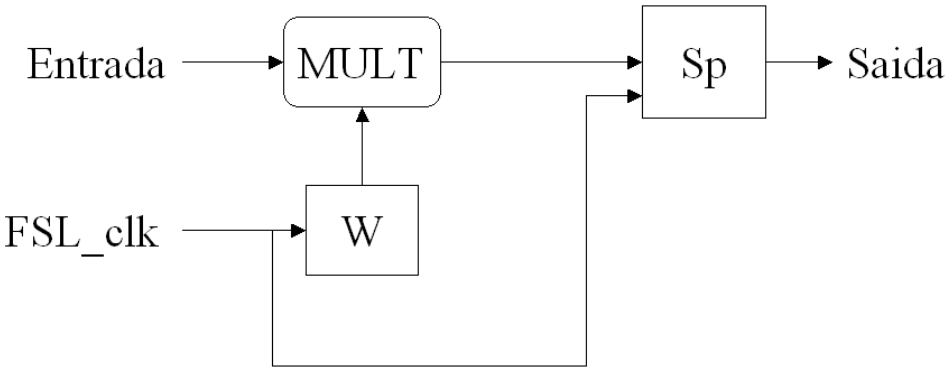

Figura 4.9: Modificação do processamento do vetor de entradas

Como o vetor de entradas da rede neural é preenchido a partir dos dados vindos do

barramento de comunicação, sendo que uma entrada por vez é atualizada a cada pulso de *FSL_clk*, optou-se por realizar as multiplicações referentes a cada entrada em paralelo para cada neurônio e acumular o resultado em Sp (visto que há um Sp para cada neurônio da camada em questão).

Anteriormente, as multiplicações dos pesos pelas entradas eram feitas mesmo para os casos em que as entradas ainda não haviam sido atualizadas. Para o caso específico em que as entradas são atualizadas uma a uma, e não todas simultaneamente, este nível de paralelismo não é necessário.

A estrutura da rede e o armazenamento e acesso aos pesos permaneceram os mesmos que os do modelo matricial. A modificação reduziu o consumo de *Slices* para 48%, o total de LUTs de 4 entradas para 44% e o consumo de multiplicadores utilizados pelo sistema para 60%.

Para os testes, foram extraídos 768 pulsos de cada tipo: normal, PVC, RBBB e LBBB. Os pulsos foram pré-processados em Matlab e em seguida processados pela rede neural em Matlab e pela rede neural implementada em *hardware* para efeito de comparação.

Os pulsos pré-processados foram enviados ao *kit* por comunicação serial. O microcontrolador Microblaze enviou cada pulso para a rede neural implementada em VHDL e contou as quantidades de cada classificação para cada tipo de sinal enviado. Os resultados foram enviados de volta para o terminal de comunicação, e podem ser vistos na Figura 4.10. Mais detalhes sobre a comunicação do *kit* com o PC externo com o uso do Microblaze podem ser vistos no Apêndice A.2.

Primeiramente apresentaram-se todos os pulsos de todos os tipos, obtendo-se o primeiro resultado observado. Em seguida apresentaram-se pulsos de cada tipo separadamente na seguinte ordem: Normais, PVC, RBBB e LBBB.

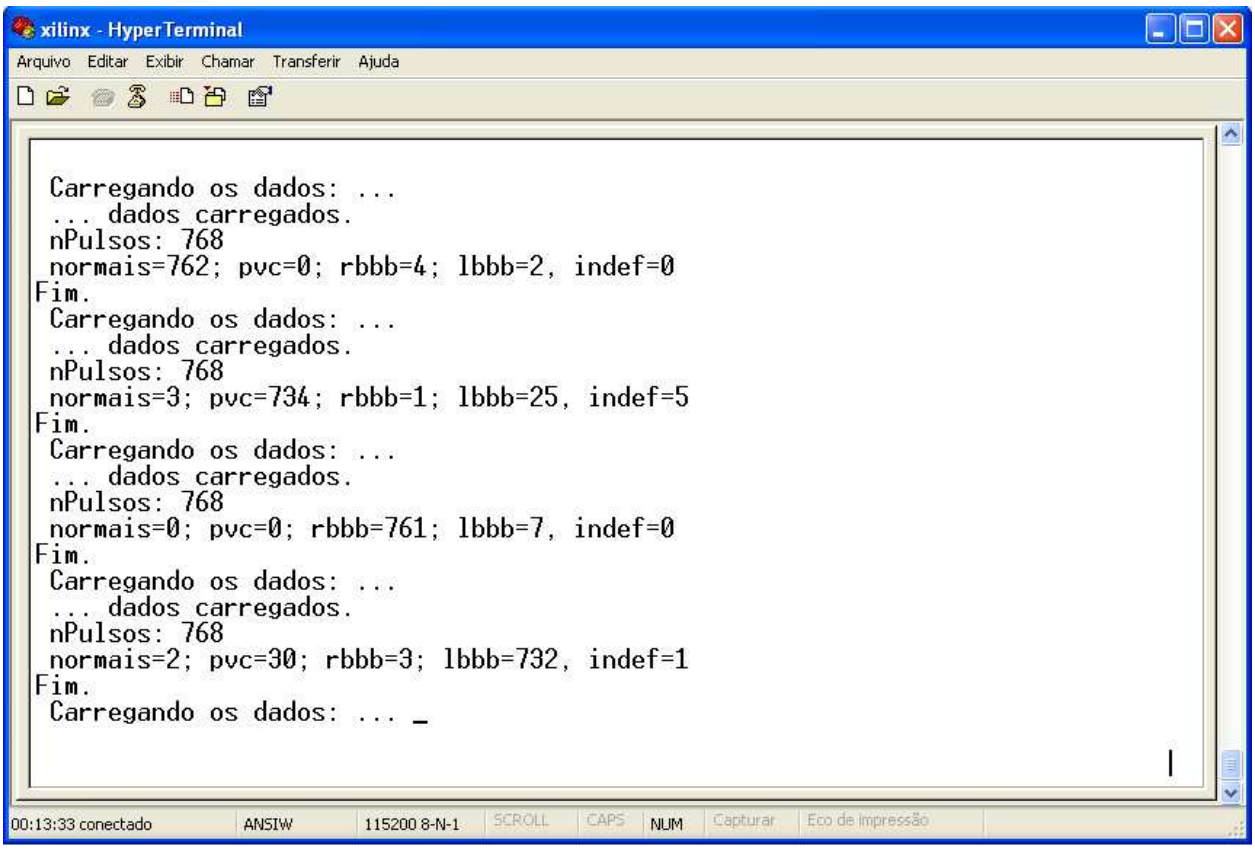

Figura 4.10: Resultados das classificações dos pulsos apresentados à rede neural implementada em *hardware*

Observar na Figura 4.10 que as classes foram separadas com sucesso na maioria dos casos. Um fenômeno interessante foi o surgimento de casos indefinidos, ou seja, casos em que duas ou mais saidas de maior magnitude apresentaram valores iguais. Isso não ocorreu nas simulações em Matlab, mas aconteceu nos testes com o sistema embarcado devido à quantização empregada no mesmo.

Para melhor visualizar o processo de classificação, foram montados os gráficos das Figuras 4.12, 4.13, 4.14 e 4.15, que comparam os desempenhos das simulações da rede neural reduzida com os obtidos pela rede neural implementada em *hardware*. Notar que os resultados obtidos em ambos os casos se apresentam de maneira similar. Importante lembrar que, em casos como na Figura 4.13, onde aparentemente há grande conflito entre as classificações, é preciso levar em consideração a relação entre as saídas para se concluir à qual classe determinada classificação pertence. Alguns exemplos podem ser vistos a seguir na Figura 4.11, uma vista ampliada da Figura 4.13 para as classificações dos pulsos 607, 608, 609 e 610.

Verificar na figura que o pulso 607 foi corretamente classificado pois S2 encontra-se em um, enquanto que S1, S3 e S4 encontram-se em zero. Os pulsos 608 e 609 são exemplos que à primeira vista parecem classificações erradas, mas que por apresentarem valores de S2 superiores aos valores de S1, S3 e S4 também fornecem classificações corretas. Já o pulso 610 é um exemplo de caso indefinido, pois todas as saídas possuem valor zero, não permitindo a classificação.

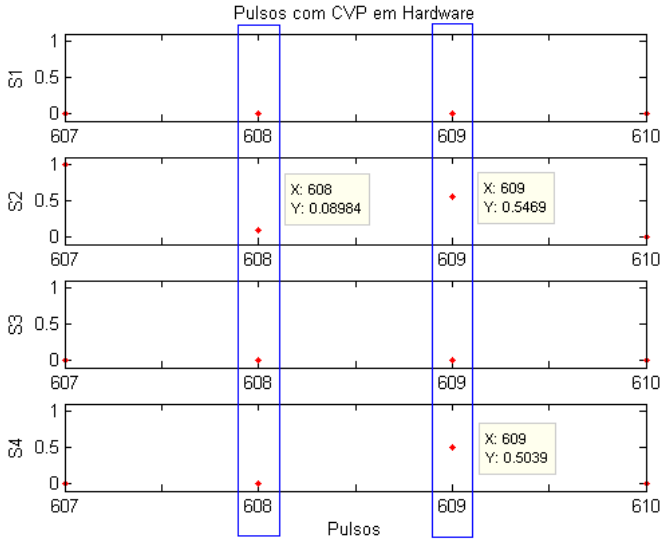

Figura 4.11: Vista ampliada da classificação de pulsos com CVP

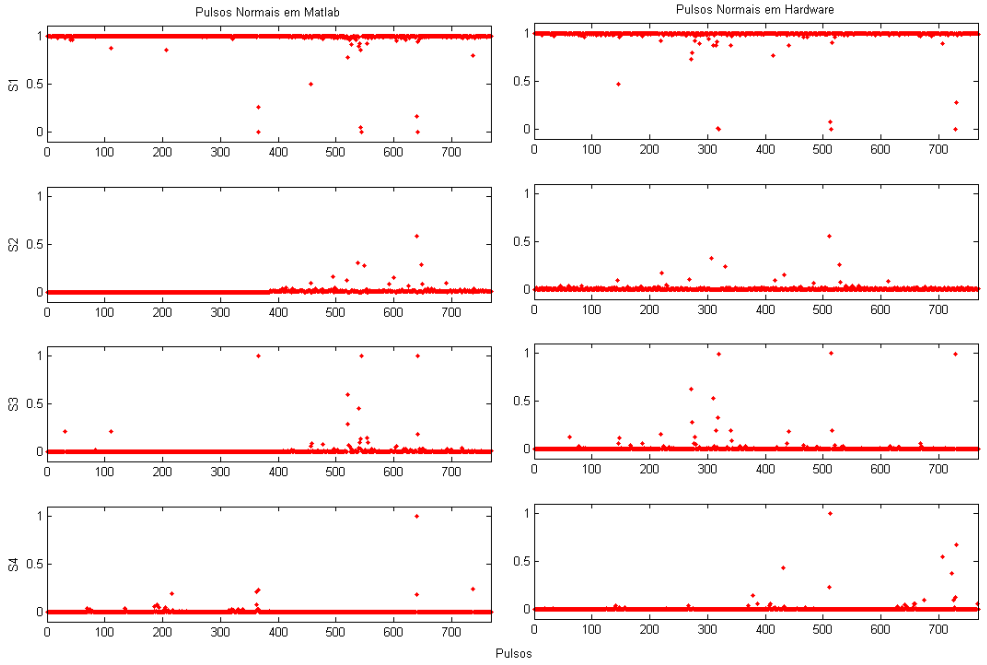

Figura 4.12: Classificação de pulsos normais

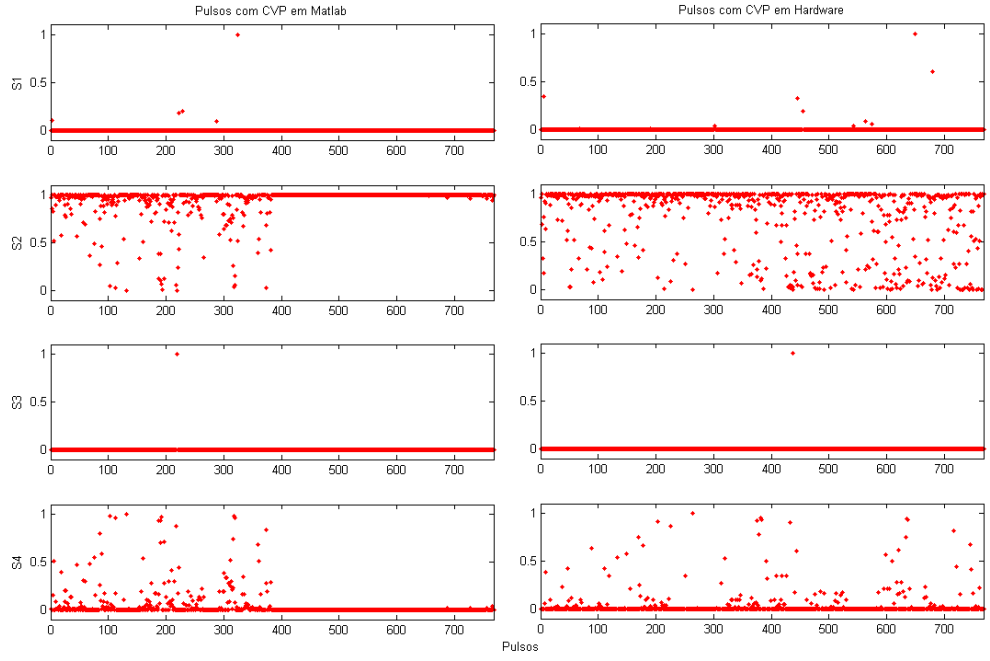

Figura 4.13: Classificação de pulsos com CVP

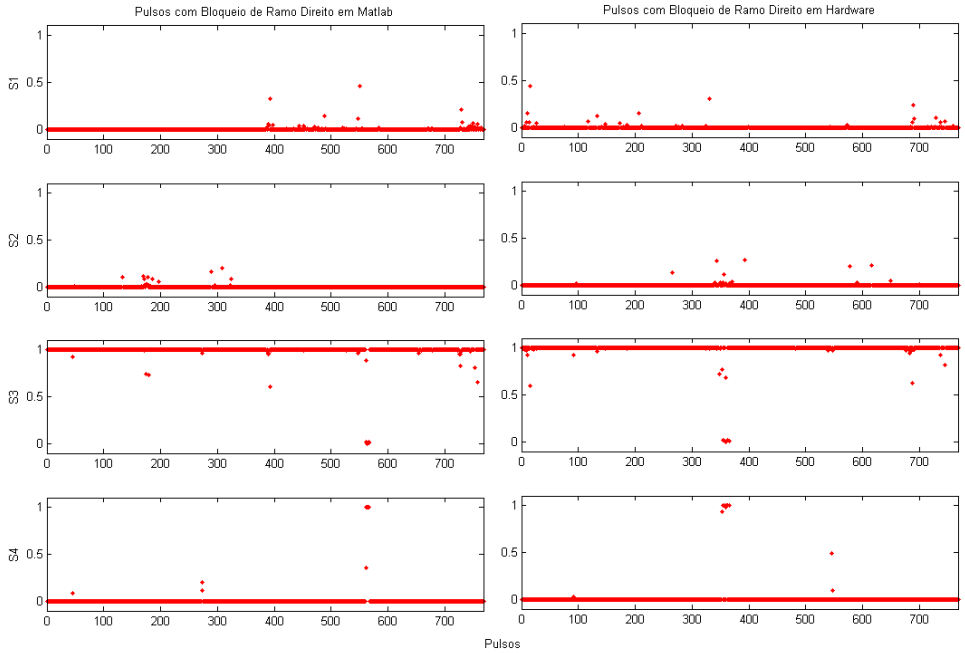

Figura 4.14: Classificação de pulsos com bloqueio de ramo direito

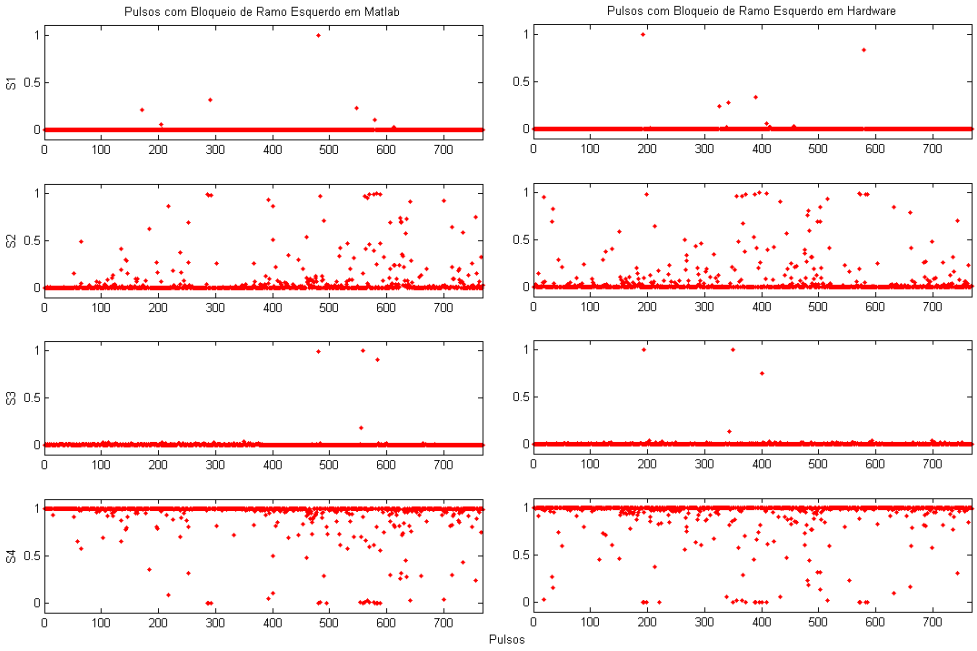

Figura 4.15: Classificação de pulsos com bloqueio de ramo esquerdo

Para facilitar então a análise do resultado obtido, e compará-lo com o esperado de acordo com a simulação em Matlab, foram construídas duas matrizes de confusão.

A Tabela 4.4 relata o resultado das simulações em Matlab, que é a referência para o que se espera do sistema embarcado.

Tabela 4.4: Matriz de confusão para a simulação em Matlab

	Nv	Vv	Rv	Lv
Np	765	0	6	2
Vp	1	737	3	19
Rp	0	0	753	6
Lp	2	31	6	741
indef	0	0	0	0
Total	768	768	768	768

Com os dados vistos na Figura 4.9, foi construída uma matriz de confusão para os dados reais, processados em *hardware*. A matriz obtida pode ser vista na Tabela 4.5.

Tabela 4.5: Matriz de confusão para o processamento em *hardware*

	Nv	Vv	Rv	Lv
Np	762	3	0	2
Vp	0	734	0	30
Rp	4	1	761	3
Lp	2	25	7	732
indef	0	5	0	1
Total	768	768	768	768

Como pode ser observado nas Tabelas 4.4 e 4.5, os resultados obtidos pela implementação em *hardware* são consistentes com os esperados.

4.4 Discussão

A capacidade de identificar os pulsos sem a ajuda das anotações manuais é importante para tornar o sistema de identificação de anomalias independente, como forma de auxílio ao analista humano. Os resultados obtidos pelo método produziram uma alta capacidade de detecção de pulsos, comparável a outros casos vistos na bibliografia pesquisada para este trabalho.

A partir das técnicas utilizadas foi possível construir um sistema classificador, baseado em redes neurais e cujo pré-processamento faz uso da análise *wavelet*, capaz de separar pulsos de eletrocardiograma afetados por cardiopatias particulares em grupos distintos com um número reduzido de casos indefinidos.

Seu desempenho é similar ao observado em trabalhos presentes na literatura e citados ao longo da Seção 3.2, o que vem a confirmar o poder de análise e classificação obtido pela junção das técnicas de redes neurais à análise pela transformada *wavelet* discreta.

Algumas otimizações foram necessárias para adequar o projeto ao *kit* utilizado. Alguns testes foram realizados visando investigar o aumento do consumo do *hardware* com um aumento do número de neurônios na camada intermediária. Os resultados dos testes podem ser vistos no Apêndice A.3. Através do uso do microcontrolador Microblaze, e com a implementação de algumas otimizações quanto ao consumo do dispositivo, foi possível estabelecer uma comunicação serial entre a rede neural desenvolvida e o PC externo, o que permitiu concluir os objetivos iniciais do trabalho.

Capítulo 5

Considerações Finais

5.1 Conclusão

A implementação da rede neural para a classificação de pulsos pré-processados de ECGs é o foco principal do presente trabalho. As contribuições principais são o sistema implementado em VHDL , com rede neural paralela.

O pré-processamento em si se beneficiou da transformada *wavelet* discreta e sua alta capacidade de análise de eventos transitórios tais como os pulsos de um eletrocardiograma. De fato, a análise com *wavelets* permite um tratamento do sinal de ECG com resultados geralmente superiores a outras técnicas. A eliminação da linha de base, presente na maioria dos ECGs, é um processo natural do método e o uso de *wavelets*-mãe da família das *daubechies* fornece resultados expressivos.

A seleção das características para a separação dos grupos no presente trabalho foi feita com base na literatura e na experimentação, e deverá ser futuramente ampliada. Os quatro grupos de pulsos (normais, contração ventricular prematura, bloqueio de ramo esquerdo, bloqueio de ramo direito) foram separados com sucesso pelas características extraídas dos pulsos individuais através da transformada *wavelet* discreta. É fato conhecido na literatura que o aumento no número de anomalias a serem detectadas torna o processo de classificação mais difícil. Anomalias diferentes podem exigir escolhas de características diferentes que possibilitem a separação dos grupos. Assim, cada projeto, tendo objetivos distintos, requer uma remodelagem do pré-processamento para atender às novas demandas.

Os resultados do processo de detecção de pulsos foram apresentados e produziram uma capacidade de detecção do complexo QRS similar às técnicas vistas na literatura. O desempenho da rede neural foi validado com o uso de várias fontes diferentes de sinais de ECG. Seu desempenho em *hardware* foi equivalente ao observado no ambiente computacional.

5.2 Trabalhos Futuros

Graças ao trabalho de Mallat em 1988, a implementação da transformada wavelet como um banco de filtros apresenta a possibilidade de se projetar em *hardware* todo o pré-processamento do sinal de ECG que faz uso da DWT. Isto tornará o sistema portátil e levará a um aumento no desempenho do projeto como um todo.

Pesquisas realizadas no Departamento de Mecânica Computacional, da Faculdade de Engenharia Mecânica na UNICAMP já estão em andamento visando a implementação da transformada *wavelet* discreta como filtros digitais em FPGA, bem como para a melhoria do sistema detector de pulsos. Um estudo interessante seria a influência que a capacidade do algoritmo detector de pulsos tem sobre o algoritmo classificador.

O aumento no número de cardiopatias analisadas pelo sistema e na quantidade de sinais provenientes de diversas fontes viria a tornar o sistema ainda mais aplicável e robusto na detecção de doenças cardíacas. Para tanto, outros parâmetros podem ser necessários para que os grupos sejam separados com sucesso. Visto que certas cardiopatias são melhor identificadas analisando-se o sinal proveniente de determinadas terminações ao invés de outras, talvez seja necessário uma análise em múltiplas terminações para que um sistema classificador consiga discernir a contento um número grande de doenças cardíacas. Nesta mesma linha, um algoritmo capaz de identificar o início e o fim de cada parte do pulso de ECG (como a duração do complexo QRS ou do segmento ST) forneceria informação valiosa à rede neural para a identificação de certas cardiopatias.

Por fim, planeja-se reunir este trabalho a um projeto maior, em desenvolvimento no laboratório, visando a construção de um sistema de análise e interpretação de eletrocardiogramas.

Referências Bibliográficas

Castro, B., Kogan, D., and Geva, A. (2000). ECG feature extraction using optimal mother wavelet. In *Electrical and ELectronic Engineers in Israel, 2000. The 21st IEEE Convention of the*, pages 346–350.

Dokur, Z., Olmez, T., and Yazgan, E. (1999). Comparison of discrete wavelet and Fourier transforms for ECG beatclassification. *Electronics Letters*, 35(18):1502–1504.

Erick, L. (2007). Fixed-Point Representation and Fractional Math. *Oberstar Consulting*.

Freeman, J. and Skapura, D. (1991). *Neural networks: algorithms, applications, and programming techniques*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA.

Gabor, D. (1946). Theory of communication: J. Inst. *Electr. Eng*, 93:429–457.

Güler, I. and Übeyll, E. (2005). ECG beat classifier designed by combined neural network model. *Pattern Recognition*, 38(2):199–208.

Goldberger, A. L., Amaral, L. A. N., Glass, L., Hausdorff, J. M., Ivanov, P. C., Mark, R. G., Mietus, J. E., Moody, G. B., Peng, C.-K., and Stanley, H. E. (2000 (June 13)). PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220. Circulation Electronic Pages: http://circ.ahajournals.org/cgi/content/full/101/23/e215.

Haykin, S. (1998). Neural networks: a comprehensive foundation. *New Jersey: Prentice Hall*.

Higham, D. and Higham, N. (2005). *MATLAB guide*. Society for Industrial Mathematics.

Ilic, S. (2007). Detection of the Left Bundle Branch Block in Continuous Wavelet Transform of ECG Signal. *signal*, 2(2):2.

Khan, M. (2007). *Rapid ECG interpretation*. Humana Press.

Kilts, S. (2007). *Advanced FPGA design: architecture, implementation, and optimization*. Wiley-IEEE Press.

Mahmoodabadi, S., Ahmadian, A., Abolhasani, M., Eslami, M., and Bidgoli, J. (2005). Ecg feature extraction based on multiresolution wavelet transform. In *Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005. 27th Annual International Conference of the*, pages 3902–3905.

Mallat, S. (1999). *A wavelet tour of signal processing*. Academic Pr.

Mallat, S. et al. (1989). A theory for multiresolution signal decomposition: The wavelet representation. *IEEE transactions on pattern analysis and machine intelligence*, 11(7):674–693.

Meyer-Baese, U. (2007). *Digital signal processing with field programmable gate arrays*. Springer Verlag.

Neufeld, H., Sive, P., Riss, E., Yahini, J., Marszalkowicz, A., Baruch, D., Kahn, H., and Medalie, J. (1971). The use of a computerized ECG interpretation system in an epidemiologic study. *Methods of information in medicine*, 10(2):85.

Omondi, A. and Rajapakse, J. (2006). *FPGA implementations of neural networks*. Springer Verlag.

Pan, J. and Tompkins, W. (1985). A real-time QRS detection algorithm. *IEEE Transactions on Biomedical Engineering*, pages 230–236.

Pedroni, V. (2004). *Circuit design with VHDL*. The MIT Press.

Savran, A. and Unsal, S. (2003). Hardware implementation of feedforward neural network using fpgas. In *Proceedings of the 7th International Conference on Electrical and Electronics Engineering*.

Saxena, S., Kumar, V., and Hamde, S. (2002). Feature extraction from ECG signals using wavelet transforms for disease diagnostics. *International Journal of Systems Science*, 33(13):1073–1085.

Silipo, R. and Marchesi, C. (1998). Artificial neural networks for automatic ECG analysis. *IEEE Transactions on Signal Processing*, 46(5):1417–1425.

Song, MH; Lee, J. P. H. and Lee, K. (2005). Classification of heartbeats based on linear discriminant analysis and artificial neural network. In *Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005. 27th Annual International Conference of the*, pages 1151–1153.

Suzuki, Y. (1995). Self-organizing QRS-wave recognition in ECG using neural networks. *IEEE Transactions on Neural Networks*, 6(6):1469–1477.

Vijaya, G., Kumar, V., and Verma, H. (1997). Artificial neural network based wave complex detection in electrocardiograms. *International journal of systems science*, 28(2):125–132.

Xilinx (2007). *MicroBlaze Development Kit Spartan-3E 1600E Edition User Guide*.

Xilinx and Mehta, N. (2008). *Programmable Logic Design - Quick Start Guide*.

Yu, S. and Chen, Y. (2007). Electrocardiogram beat classification based on wavelet transformation and probabilistic neural network. *Pattern Recognition Letters*, 28(10):1142–1150.

Yu, S. and Chou, K. (2008). Integration of independent component analysis and neural networks for ECG beat classification. *Expert Systems with Applications*, 34(4):2841–2846.

Zywietz, C. and Schneider, B. (1973). *Computer application on ECG and VCG analysis.* North-Holland Pub. Co.

Apêndice A

A.1 Representação numérica

Ao implementar um sistema em FPGA é importante escolher com cuidado o tipo de representação que será utilizada para se efetuar as operações matemáticas que o projeto requer. Em sistemas embarcados, como no caso daqueles implementados em FPGA, existem certas limitações que precisam ser observadas durante essa escolha. De maneira geral, o hardware é capaz de lidar somente com versões binárias dos números. Cada número binário possui um equivalente inteiro direto de acordo com o seguinte modelo:

$$\underline{\quad} \quad \underline{\quad} \quad \underline{\quad} \quad \underline{\quad}$$
$$2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

Seguindo essa representação binária, o número de quatro bits 0001_2 é equivalente ao número inteiro $0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1_{10}$ e 1001_2 equivale a $1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 9_{10}$. Notar no entanto que a representação binária direta permite apenas o uso de inteiros variando de 0 até 2^n sendo n o número de bits utilizados. Muitas vezes há a necessidade de se representar números negativos, não disponíveis pela representação direta acima descrita. Uma das técnicas que permite contornar esse problema é a representação por complemento de dois. Na representação por complemento de dois, o bit mais significativo fica reservado para o sinal, e uma série de operações são efetuadas sobre o número em questão para que as operações matemáticas sejam consistentes.

O processo é bem simples, e pode ser implementado em 2 passos:

1. Inverte-se cada bit do número binário.

2. Soma-se 1 ao resultado.

Tomando-se como exemplo o número 7_{10}, que pode ser escrito em quatro bits como 0111_2. Invertendo-se cada bit tem-se 1000 e somando-se 1 obtém-se: 1001. Verificar que o

resultado final 1001_2 é a representação binária simples equivalente ao número inteiro 10_{10}. O resultado prático então é que o intervalo absoluto de representação é reduzido pela metade, sendo que tal intervalo pode agora ser representado tanto positiva quanto negativamente. Matematicamente tem-se que $-2_n <= N <= 2_{n-1}$, o que gera uma assimetria em torno do zero. Um diagrama que ilustra o caso de quatro bits pode ser visto na Figura A.1.

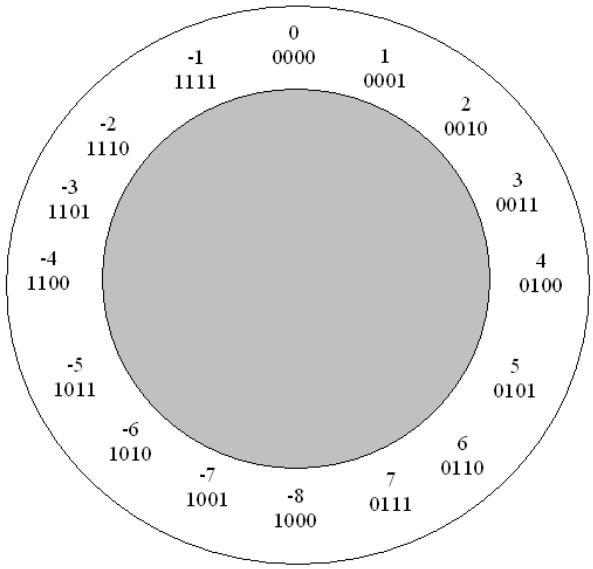

Figura A.1: Representação decimal em complemento de dois

Aplicações reais no entanto exigem componentes fracionários que não podem ser representados com as técnicas vistas até o momento. Uma das maneiras de se contornar essa limitação é através do uso de uma representação por ponto fixo. Na representação por ponto fixo, adiciona-se um ponto decimal imaginário entre os bits do número binário da seguinte maneira: *Bi.Bf*

Bi equivale aos bits que representam a parte inteira do número enquanto que *Bf* é responsável por representar a sua parte fracionária. Tomando o exemplo de quatro bits anterior, o resultado seria:

$$\underset{2^1}{\_\_} \quad \underset{2^0}{\_\_} \quad \bullet \quad \underset{2^{-1}}{\_\_} \quad \underset{2^{-2}}{\_\_}$$

Seguindo essa representação binária, o número de quatro bits 0110_2 é equivalente ao número decimal $0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} = 1.510$ e 0111_2 equivale a $0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 1.7510$. O mais interessante é que como a representação acima manteve inalterada a forma básica do número binário (ou seja, não foi necessária a adição

de bits extras que necessitem de tratamento especial, como no caso de mantissas) a mesma técnica do complemento de dois pode ser utilizada para representar números fracionários tanto negativos como positivos. Para um número de bits tão pequeno, a resolução alcançada é extremamente limitada. A resolução r pode ser aumentada, aumentando-se o número de bits da parte fracionária Bf seguindo a Equação A.1:

$$r = \frac{1}{2^{Bi}} \tag{A.1}$$

Os limites da representação por ponto fixo ficam então definidos como (Erick, 2007): $-2^{Bi-1} \leq N \leq 2^{Bi-1} - 2^{-Bf}$

De maneira geral, implementações de sistemas numéricos de ponto-fixo apresentam maior velocidade e menor custo enquanto que implementações com ponto-flutuante têm um *range* maior de representação e não requer escalonamento (Meyer-Baese, 2007).

No presente trabalho foi utilizada a representação por ponto fixo em complemento de 2 com 16 bits, 8 dos quais foram destinados à parte fracionária.

A.2 Comunicação com o Microprocessador Microblaze™

Neste projeto utilizou-se a memória SDRAM para o armazenamento temporário dos sinais a serem analisados e a porta de comunicação RS232 para o envio de dados ao *kit* e sua posterior recuperação.

O projeto em VHDL é importado como um componente externo e o fluxo de dados entre o componente VHDL e o microprocessador é controlado por uma máquina de estados, como na Figura A.2.

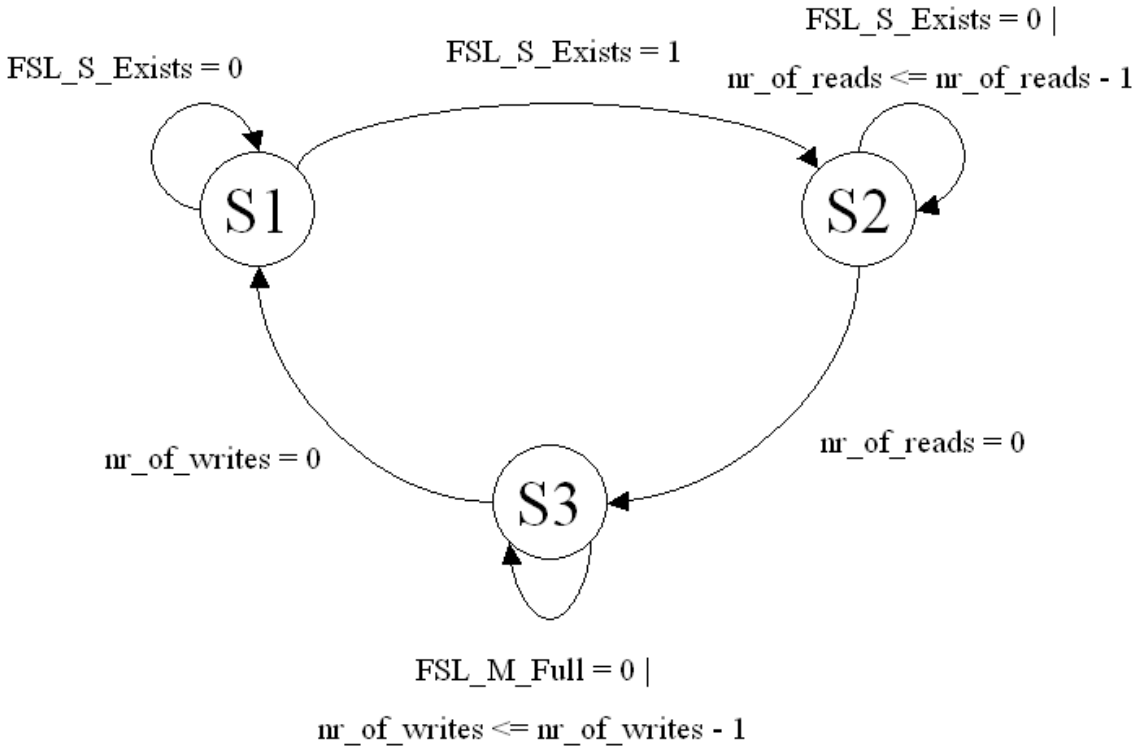

Figura A.2: Máquina de estados para o controle do fluxo de dados

Sendo:

O sistema inicia no estado de espera S1, assim que recebe um bit sinalizando a existência de dados no barramento FSL, por meio de FSL_S_Exists, o sistema passa para o estado S2 de leitura. O barramento FSL (*Fast Simplex Link*) é um canal de comunicação unidirecional ponto-a-ponto que viabiliza a comunicação em alta velocidade entre dois elementos do projeto.

S1:	Estado Idle
S2:	Estado Read_Inputs
S3:	Estado Write_Outputs
FSL_S_Exists:	Bit que sinaliza a existência de dados no barramento FSL
nr_of_reads:	Número de dados a serem lidos faltantes
nr_of_writes:	Número de dados a serem escritos faltantes

No estado de leitura, um dado é lido a cada pulso de *CLOCK* e armazenado no vetor de entradas a ser processado. O que define o tamanho do vetor é o número de dados a serem lidos (*nr_of_reads*), que é decrescido conforme os dados são enviados pelo microprocessador para o componente em *hardware*. Quando o número restante de dados a serem lidos alcança zero, o sistema passa para o estado S3 de escrita.

No estado de escrita, os dados processados são enviados um a um para o microprocessador, decrescendo o número restante de dados a serem enviados (*nr_of_writes*).

Quando não há mais dados a serem enviados, o sistema retorna ao estado S1 de espera, e aguarda pelo próximo vetor de entradas.

A.3 Consumo de *hardware*

Para se investigar o consumo de cada modelo foi selecionado o problema lógico XOR, resolvido com o uso de redes neurais. A escolha foi baseada na facilidade de se aumentar gradativamente a estrutura da rede para a análise.

A.3.1 Consumo de Recursos do FPGA

Um dos grandes desafios ao se projetar um sistema que será implementado em hardware são as limitações impostas pelo dispositivo. Mesmo com o aumento dos recursos dos FPGAs ao longo dos anos, cuidado especial deve ser tomado ao se programar o sistema desejado para que não se esbarre nessas limitações.

Em muitos casos, não é apenas o que se codifica, mas sim como se codifica que acaba permitindo o desenvolvimento de um produto ou idéia em hardware programável.

Buscando-se investigar esse fenômeno é que se resolveu implementar a mesma rede neural, desenvolvida com o auxilio do toolbox do Matlab, de duas maneiras estruturalmente distintas, como visto em capítulos anteriores.

Cada estrutura apresenta um consumo do hardware disponível distinto para diferentes topologias de rede. De modo a observar este comportamento característico para um nível crescente de complexidade estrutural, treinou-se redes neurais para resolver um problema com número de neurônios reduzido, como o do XOR de 2 entradas.

O número de neurônios na camada intermediária variou de 3 até 32. Cada resultado foi implementado tanto no modelo por componentes quanto no matricial, produzindo os resultados que podem ser vistos nas Figuras A.3 e A.4. Notar que há um aumento proporcional no consumo dos vários tipos de componentes que compõe o FPGA para ambos os casos.

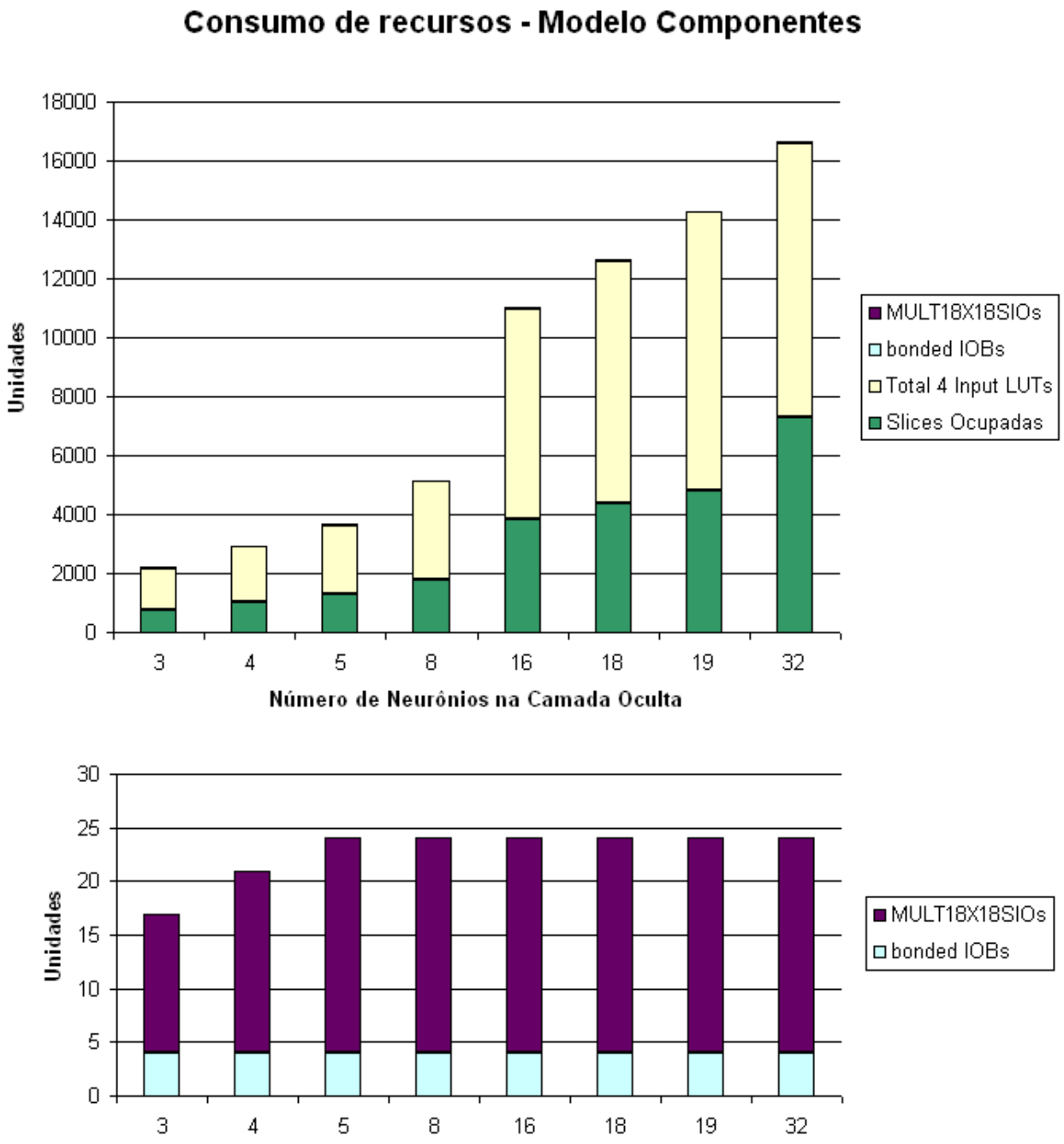

Figura A.3: Consumo de recursos do FPGA pela rede neural baseada em componentes para a solução do problema XOR

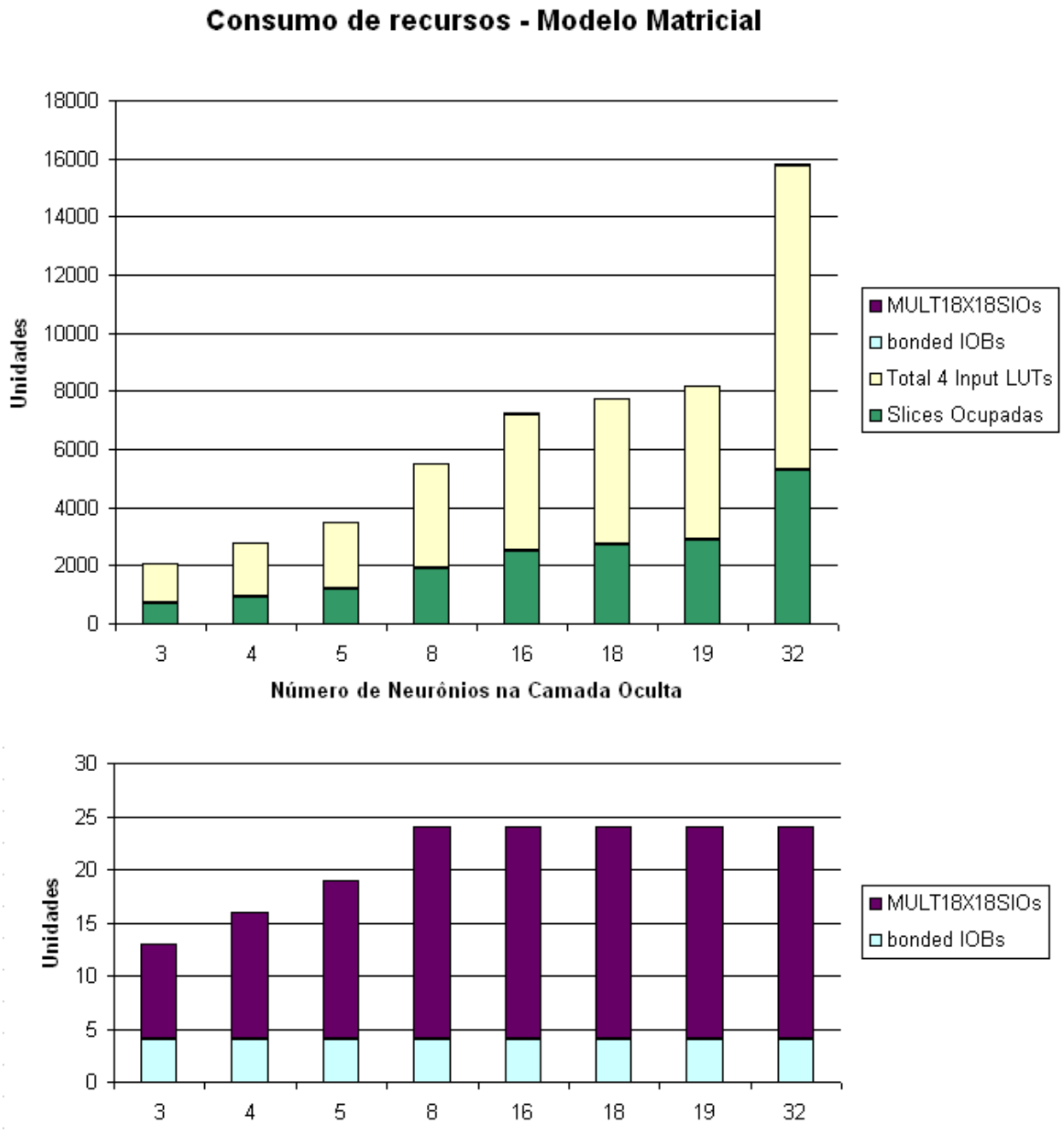

Figura A.4: Consumo de recursos do FPGA pela rede neural matricial para a solução do problema XOR

A Figura A.5 permite uma melhor visualização das diferenças entre as duas implementações. Verificar que o modelo matricial apresenta um consumo menor dos recursos do *hardware* para uma mesma topologia de rede.

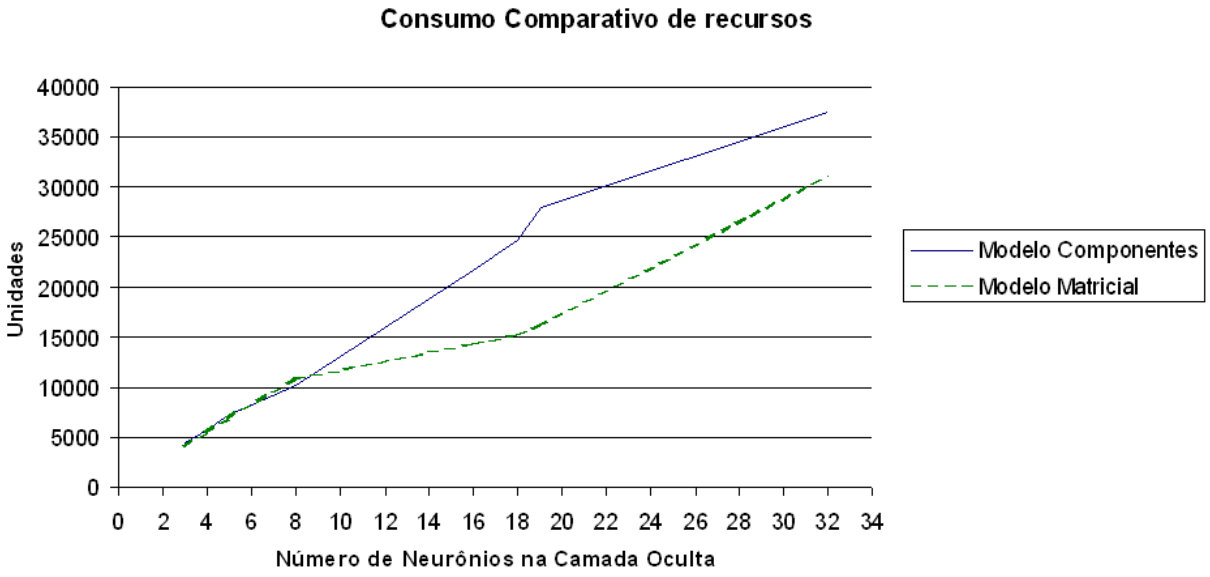

Figura A.5: Comparação dos consumos totais das redes baseadas em componentes e das matriciais

Os números apresentados não são absolutos. Devido à complexidade do processo de sintetização e otimização do projeto, é comum que o software acabe tomando decisões diferentes quanto à disponibilização e utilização dos recursos. Observou-se que ao se implementar o mesmo caso mais de uma vez, há pequenas alterações nos números de dispositivos consumidos, bem como no tempo que se leva para o término do processo.

O tempo total de cada implementação abrange todo o processo e pode ser visto na Figura A.6. O tempo total foi medido partindo da sintetização do código reescrito até o *hardware* ser gravado com o *bitstream*, como discutido na Seção 2.4. Notar que o tempo para a implementação do projeto cresce de maneira considerável com o aumento da rede.

A configuração do computador utilizado para o desenvolvimento do projeto é a seguinte:

Sistema operacional:	Windows XP Professional, Service Pack 3
Tipo de processador:	DualCore Intel Core 2 Duo E4600, 2400 MHz
Memória do Sistema:	2039 MB (DDR2-800 DDR2 SDRAM)

Importante ressaltar que todos os testes, até o máximo de consumo de *hardware* permitido pelo *kit*, foram completos. A rede neural foi reescrita em VHDL, sua sintetização e imple-

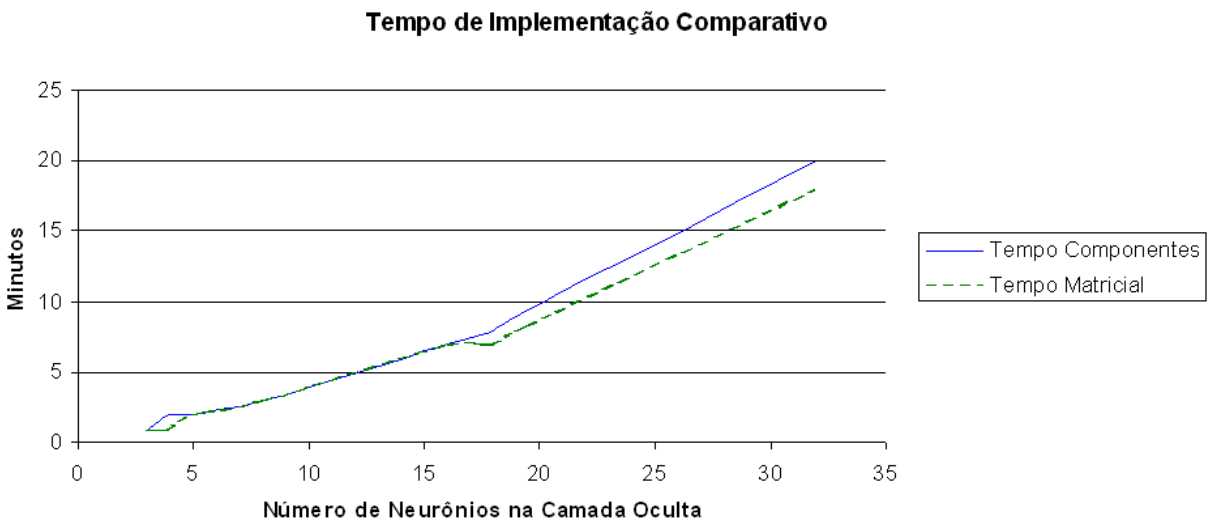

Figura A.6: Comparação dos tempos de sintetização e programação totais para as redes baseadas em componentes e para as matriciais

mentação seguiram os passos descritos na Seção 2.4, o *bitstream* foi gravado em *hardware* e as saídas foram confirmadas experimentalmente.

Apêndice B

B.1 Rede Neural por Componentes em FPGA

B.1.1 Programa Principal

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use work.tipos.all;
use work.components.all;
use work.constantes.all;

entity main is
-- Lembrar de modificar o arquivo constantes.vhd

port (  signal  Entrada     :   in signed(15 downto 0);
        signal  CLOCK           :   in  std_logic;
        signal  Saida           :   out std_logic;
        signal  L1          :   out std_logic;  -- LED 1 N
        signal  L2          :   out std_logic;  -- LED 2 V
        signal  L3          :   out std_logic;  -- LED 3 R
        signal  L4          :   out std_logic   -- LED 4 L
);


end main;

architecture behavioral of main is

--0000000100000000 -- 1.0
--0000000010000000 -- 0.5

    signal Input: vetor16bits(0 to nInputs1-1):=(
    "0000000000000000","0000000000000000","0000000000000000","0000000000000000",
    "0000000000000000","0000000000000000","0000000000000000","0000000000000000",
```

```
    "0000000000000000","0000000000000000","0000000000000000","0000000000000000",
    "0000000000000000","0000000000000000","0000000000000000","0000000000000000",
    "0000000000000000","0000000000000000","0000000000000000","0000000000000000",
    "0000000000000000","0000000000000000","0000000000000000","0000000000000000",
    "0000000000000000","0000000000000000","0000000000000000","0000000000000000",
    "0000000000000000","0000000000000000","0000000000000000","0000000000000000",
   "0000000100000000");


    signal saidaC1: vetor16bits(0 to nNeuronios1); -- ultima posição com o bias de 1
    signal saidaC2: vetor16bits(0 to nNeuronios2);

    signal initJanela   : integer :=0; -- Aponta o inicio da janela sendo analisada
    signal i : integer:=0;

    constant zero   :   signed  (15 downto 0)   :=  "0000000000000000";
    constant um     :   signed  (15 downto 0)   :=  "0000000100000000";
    constant meio   :   signed  (15 downto 0)   :=  "0000000010000000";



begin


 process (CLOCK)
 begin
    if CLOCK='1'and CLOCK'event then
        input(i)<=Entrada;

        if i>=nInputs1 then
            i<=0;
        elsif i<nInputs1  then
            i<=i+1;
        end if;
    end if;
 end process;


layer1: camada1 port map (Input,saidaC1);

layer2: camada2 port map (saidaC1,saidaC2);



process (saidaC2)
begin

    if ((saidaC2(0)>=saidaC2(1)) and (saidaC2(0)>=saidaC2(2)) and(saidaC2(0)>=saidaC2(3)))↩
         then
        L1<='1';
        L2<='0';
        L3<='0';
        L4<='0';
```

```
    elsif ((saidaC2(1)>=saidaC2(0)) and (saidaC2(1)>=saidaC2(2)) and(saidaC2(1)>=saidaC2↩
        (3))) then
        L1<='0';
        L2<='1';
        L3<='0';
        L4<='0';
    elsif ((saidaC2(2)>=saidaC2(0)) and (saidaC2(2)>=saidaC2(1)) and(saidaC2(2)>=saidaC2↩
        (3))) then
        L1<='0';
        L2<='0';
        L3<='1';
        L4<='0';
    elsif ((saidaC2(3)>=saidaC2(0)) and (saidaC2(3)>=saidaC2(1)) and(saidaC2(3)>=saidaC2↩
        (2))) then
        L1<='0';
        L2<='0';
        L3<='0';
        L4<='1';
    end if;

end process;


end behavioral;
```

B.1.2 Neurônio

```
------ File Neuronio.vhd: ------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use work.tipos.all;
use work.components.all;
use work.constantes.all;
--------------------------------------------------


entity neuronioC1 is

    port (  signal  Input       : in vetor16bits (0 to nInputs1-1);
                signal  W           : in    vetor16bits(0 to nInputs1-1);
                signal  saida       : out   signed (nbitsi-1 downto 0));

end neuronioC1;

architecture Behavioral of neuronioC1 is

signal soma     : signed (nbitsprod-1 downto 0);

begin
```

```
    corpo       : macC1 port map (Input, W, soma);
    funcao  : ActFunc port map (soma, saida);

end Behavioral;
```

B.1.3 Soma Ponderada

```
------ File MAC.vhd: ------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use work.tipos.all;
use work.constantes.all;
---------------------------------------------------------

entity macC1 is
    port (signal    Input       : in    vetor16bits(0 to nInputs1-1);
          signal    W           : in    vetor16bits(0 to nInputs1-1);
          signal    soma        : out   signed (nbitsprod-1 downto 0));
end macC1;
---------------------------------------------------------
architecture Behavioral of macC1 is

signal product : vetor32bits (0 to nInputs1-1);
signal sum : vetor32bits (0 to nInputs1-2);

begin

 multiplica: FOR i IN 0 TO nInputs1-1 GENERATE

        product(i)<= Input(i)*W(i);

 END GENERATE;

 sum(0)<=product(0)+product(1);
 acumula: FOR i IN 1 TO nInputs1-2 GENERATE

         sum(i)<=sum(i-1)+product(i+1);

 END GENERATE;

soma<=sum(nInputs1-2);

end Behavioral;
```

B.1.4 Camada 1

```
------ File camada1.vhd: ------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use work.tipos.all;
use work.components.all;
use work.constantes.all;
-------------------------------------------------------

entity camada1 is

    port (  signal  Input       : in vetor16bits (0 to nInputs1-1);
            signal  saida       : out   vetor16bits (0 to nNeuronios1));

end camada1;

architecture Behavioral of camada1 is

-- Pesos ( o ultimo de cada sinal é o peso do bias)

signal w0: vetor16bits (0 TO nInputs1-1):= ("1111111011101001","0000000100001110","↩
    0000001000011011","0000000001011110","0000000110101011","0000000011100110","↩
    0000000000111100","1111111100100101","1111111011000001","0000000010101101","↩
    1111111100100001","1111111011111110","1111111011011101","1111111100000101","↩
    1111111010111011","1111111011000111","1111111100000110","1111111011110110","↩
    1111111100011101","1111111101100110","0000000001111100","0000000011011000","↩
    1111111111000111","1111111110010110","1111111111110000","1111111111001101","↩
    1111101110111101","1111111101100000","1111111001010110","1111111000100001","↩
    1111110111000001","1111111001001000","1111111011010010","1111110101110001","↩
    1111110101010011","1111110011100001","0000000100111100");
signal w1: vetor16bits (0 TO nInputs1-1):= ("1111111101000110","1111111110011010","↩
    0000000000111000","0000000001111101","0000000001011101","1111111010011000","↩
    1111111000101010","0000000111001000","0000000011011101","1111111011101001","↩
    0000000010011111","0000000010101100","0000000010101010","0000000001101001","↩
    0000000010010000","0000000001001111","0000000001101100","0000000001010111","↩
    0000000010100001","1111111111101001","1111111001110111","1111111001001000","↩
    0000000001011101","0000000011010110","0000000000010111","1111101111001000","↩
    1111111110001100","0000000000010010","0000001011100100","0000001001001000","↩
    0000000011001010","1111110101110110","1111111000110101","0000000111100101","↩
    0000000011101000","0000000100110011","1111111100101110");
signal w2: vetor16bits (0 TO nInputs1-1):= ("1111111010000101","0000000111111011","↩
    1111111111110010","1111111100111010","0000000110111101","1111111100010101","↩
    0000000100100011","1111111001010001","1111111111011010","0000010100110101","↩
    1111111011011100","1111111011110100","1111111100100100","1111111100010101","↩
    1111111010111000","1111111011010001","1111111011011111","1111111100100011","↩
    1111111010110111","0000000101000101","0000001000010001","0000001101110000","↩
    0000001011100011","0000001011101010","0000001100010100","1111111000000010","↩
    1111111001011110","0000000100111110","1111111111010100","1111111011001001","↩
    1111110110100101","1111111011000010","0000000011101111","0000001011110001","↩
```

```
    1111111101111111","1111111110001110","1111111111001001");
signal w3: vetor16bits (0 TO nInputs1-1):= ("1111111101001111","0000000011010011","↩
    0000000000110111","1111111001111001","1111111100110001","1111111101101001","↩
    0000000111001010","1111111000110111","1111111101100010","0000000000001101","↩
    0000000001111010","0000000010001011","0000000001101111","0000000001110010","↩
    0000000001101100","0000000001101001","0000000010110101","0000000010011110","↩
    0000000010100101","0000000010100101","0000000010011110","0000000010001101","↩
    0000000100001100","0000000100100001","0000000010011000","0000000000100100","↩
    0000000100100100","0000000000001000","1111111111100010","1111111111110010","↩
    1111111110111100","1111111101001001","1111111010101001","1111111000111100","↩
    1111111100111010","1111111100011101","0000001011100110");
signal w4: vetor16bits (0 TO nInputs1-1):= ("1111110111010111","0000000110011111","↩
    1111111101110000","0000000011000001","0000000011111000","1111111100010011","↩
    0000000000000010","0000000101100010","0000000011011000","1111101011101110","↩
    0000000001111000","0000000001100010","0000000000101111","0000000001010100","↩
    0000000011011001","0000000000111111","0000000000110101","0000000001101101","↩
    0000000001101000","1111111101100000","1111111111000010","1111111100011010","↩
    1111110100011111","1111110001101001","1111101011000110","1111111100110111","↩
    0000001001100110","0000001100011111","0000001011010011","1111111111101001","↩
    0000001100000000","0000000101010001","1111111011011100","0000000010111101","↩
    1111111110111011","0000000000111010","0000010000100111");
signal w5: vetor16bits (0 TO nInputs1-1):= ("1111111100011000","0000000010000000","↩
    0000000010101011","1111111111101010","0000000001000000","0000000011111101","↩
    0000000001101000","1111111001001000","1111111011100010","1111111000010110","↩
    0000000100111010","0000000100010011","0000000011101110","0000000100010111","↩
    0000000011111010","0000000100111000","0000000100010001","0000000011010101","↩
    0000000101010000","0000000010001011","0000000000010110","1111111101011101","↩
    0000000011010001","0000000011111011","0000000010110011","1111111010111011","↩
    1111111110110101","1111111111001100","0000000101111111","0000001000101001","↩
    0000000110000001","1111111111101110","1111111101100110","0000000010110110","↩
    0000000111100100","0000000111111100","0000000011111010");
signal w6: vetor16bits (0 TO nInputs1-1):= ("0000000000011000","1111111010101011","↩
    0000000000000100","1111111010010100","0000000101110000","0000000000001101","↩
    1111111111011110","0000000010010101","0000000000001100","1111111110101101","↩
    1111111011101111","1111111011010110","1111111011000001","1111111100000101","↩
    1111111010010000","1111111100000011","1111111011000010","1111111010111001","↩
    1111111010100101","1111111100110011","0000000000011010","0000000100100011","↩
    0000000110101110","0000000110100010","0000000100001010","0000000011010100","↩
    0000000000101110","0000010000010101","0000000001011110","1111111101100010","↩
    1111111011000101","1111111010111101","1111111001110011","1111110110001010","↩
    1111111110111100","1111111101000000","1111111100100010");
signal w7: vetor16bits (0 TO nInputs1-1):= ("1111111001100100","0000000000110001","↩
    0000000000000011","0000000001101001","0000000001101101","1111111011000001","↩
    1111111100111101","1111111100111011","1111111010010011","1111111110001010","↩
    1111111111011010","1111111111111010","1111111110101001","1111111111111011","↩
    1111111111111010","1111111111101100","1111111111010011","1111111110101111","↩
    1111111111011101","1111111101100001","0000000011101000","0000000000001101","↩
    1111111100010010","1111111011111110","1111111100111000","0000010101100101","↩
    1111111110010101","0000000011110001","1111111000000101","1111111011111110","↩
    1111111111000010","0000001001111111","0000000001010110","1111110010010011","↩
    0000000011100010","0000000001100111","0000001000000111");
signal w8: vetor16bits (0 TO nInputs1-1):= ("1111111011100101","1111110110010000","↩
    0000000101011111","1111111110110001","0000000101001111","0000000011010110","↩
    0000000101101000","1111111110110110","1111111110100000","0000000100111111","↩
    1111111110111100","1111111111100101","1111111110101101","1111111110110011","↩
```

```
    1111111110000001","1111111111000001","1111111110110100","1111111110111101","↩
    1111111101110010","0000000011000101","0000000000011001","1111111111000100","↩
    0000000000000000","0000000000110000","0000000010110000","1111101011000110","↩
    1111111000010000","1111111010100010","1111111100101100","1111111011000001","↩
    1111111100011110","1111111011101110","0000000011000000","0000001011010011","↩
    0000000000110111","0000000000101011","1111111111110110");
signal w9: vetor16bits (0 TO nInputs1-1):= ("0000001001001010","1111111110001101","↩
    1111111110110011","0000001000100111","1111111100010111","0000000000100100","↩
    0000000110010010","0000000110000010","0000000100100011","1111101110100111","↩
    0000000000110000","0000000000111010","0000000000111010","0000000000110010","↩
    0000000010100000","0000000000111100","0000000010010110","0000000001000010","↩
    0000000011001010","1111111001111011","1111111100101010","1111111101110000","↩
    0000000101111100","0000000001110100","1111111101101100","0000000000001001","↩
    1111110001000011","1111100111101010","0000000111111100","0000001010111000","↩
    0000000001101111","1111111001100000","1111110111001110","1111111010110110","↩
    0000001000000011","0000000111001110","0000000001001010");
signal w10: vetor16bits (0 TO nInputs1-1):= ("1111111111100110","1111111011011100","↩
    0000000000111010","0000000100001000","1111111101010101","1111111011100000","↩
    0000000011011001","0000000100101011","0000000010001101","0000001111100000","↩
    1111111011111011","1111111100001000","1111111101001000","1111111101101000","↩
    1111111101010110","1111111100101110","1111111101010110","1111111100110100","↩
    1111111101000101","1111111101111110","1111111101011100","1111111111110100","↩
    1111111111001000","1111111111111000","0000000000001011","1111111000111011","↩
    0000000011000110","0000001001011111","1111111100100111","1111111010011011","↩
    1111111100011110","1111111110001001","1111111011000011","1111111101011100","↩
    1111110110011000","1111110111001111","0000001000001100");
signal w11: vetor16bits (0 TO nInputs1-1):= ("0000000100110011","0000000101111011","↩
    0000000100100011","0000000100111000","1111111110100101","1111111100001101","↩
    1111111100001100","1111111110011001","0000000010011000","1111111010101101","↩
    0000000100001101","0000000011100011","0000000010110101","0000000011001100","↩
    0000000011000100","0000000011111111","0000000011011001","0000000011010101","↩
    0000000011110101","1111111111000000","1111111010100100","1111111010000001","↩
    0000000001000110","0000000000010111","0000000010001001","1111110000010101","↩
    1111110111111110","0000000000101111","0000000100011101","0000000101111010","↩
    0000000111001010","0000000101000101","1111111111001000","0000001000011100","↩
    0000001010111110","0000001100110100","0000010000101101");
signal w12: vetor16bits (0 TO nInputs1-1):= ("1111111011101001","0000000000101110","↩
    1111111110001001","1111111110101101","0000000010011101","0000000000001111","↩
    1111111010101010","0000000111100110","0000000100011010","0000000001110010","↩
    1111111111010011","1111111111100000","1111111111110010","1111111111110011","↩
    1111111110011011","1111111111100011","1111111111011110","1111111111010011","↩
    1111111111101000","0000000001111111","0000000101000101","0000000111010111","↩
    0000000101011010","0000000011110011","0000000111000000","1111111111001100","↩
    1111110001110011","0000000000001110","1111111110001011","1111111111110101","↩
    0000000000011101","0000000010100010","0000000101001011","0000000001100001","↩
    1111111100101001","1111111011001100","1111110100101001");
signal w13: vetor16bits (0 TO nInputs1-1):= ("0000000000011111","1111111011111000","↩
    1111111001011001","0000000011101111","0000000010101101","0000000110101110","↩
    0000000111001000","0000000010000010","0000000000011100","1111111011110010","↩
    1111111110001111","1111111110000001","1111111111100010","1111111110111010","↩
    1111111110000101","1111111111101001","1111111110001101","1111111110111111","↩
    1111111110100001","1111111011101111","1111111110001100","1111111101011101","↩
    1111111100010111","1111111101001111","1111111010100100","1111111111100011","↩
    1111111100100001","1111111110110011","0000000000100001","1111111111010001","↩
    1111111100011101","1111110111110110","1111110110101000","1111111001001110","↩
```

```
    1111111011011111","1111111011001010","0000000101111110");
signal w14: vetor16bits (0 TO nInputs1-1):= ("0000000000011101","0000000001110001","↩
    1111111111001101","0000001000000001","1111111110110000","1111111011000001","↩
    0000000001100011","1111111001001110","1111111001010011","0000000101011101","↩
    1111111001111110","1111111010000101","1111111011001101","1111111010110101","↩
    1111111001010001","1111111010100011","1111111001110100","1111111010011001","↩
    1111111010011101","1111111010101101","1111111001100110","1111111111000000","↩
    0000000010100110","0000000011100111","0000000111101000","1111111101001111","↩
    1111110111011010","1111111110000010","1111111001011101","1111111101101011","↩
    0000000011110001","0000001100100110","0000001010101101","0000000100011001","↩
    0000000001111111","0000000010010011","1111101110000011");

 begin
 neuronio0      : neuronioC1 port map (Input, w0, saida(0));
 neuronio1      : neuronioC1 port map (Input, w1, saida(1));
 neuronio2      : neuronioC1 port map (Input, w2, saida(2));
 neuronio3      : neuronioC1 port map (Input, w3, saida(3));
 neuronio4      : neuronioC1 port map (Input, w4, saida(4));
 neuronio5      : neuronioC1 port map (Input, w5, saida(5));
 neuronio6      : neuronioC1 port map (Input, w6, saida(6));
 neuronio7      : neuronioC1 port map (Input, w7, saida(7));
 neuronio8      : neuronioC1 port map (Input, w8, saida(8));
 neuronio9      : neuronioC1 port map (Input, w9, saida(9));
 neuronio10     : neuronioC1 port map (Input, w10, saida(10));
 neuronio11     : neuronioC1 port map (Input, w11, saida(11));
 neuronio12     : neuronioC1 port map (Input, w12, saida(12));
 neuronio13     : neuronioC1 port map (Input, w13, saida(13));
 neuronio14     : neuronioC1 port map (Input, w14, saida(14));
 saida(nNeuronios1) <= "0000000100000000"; -- 1 bias

end Behavioral;
```

B.1.5 Camada 2

```
------ File camada2.vhd: ----------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
library main_v1_00_a;
use main_v1_00_a.tipos.all;
use main_v1_00_a.components.all;
use main_v1_00_a.constantes.all;
-----------------------------------


entity camada2 is

    port (  signal  Input       : in vetor16bits (0 to nInputs2-1);
            signal  saida       : out   vetor16bits (0 to nNeuronios2));

end camada2;
```

```
architecture Behavioral of camada2 is

-- Pesos ( o ultimo de cada sinal é o peso do bias)

signal w0: vetor16bits (0 TO nInputs2-1):= ("1111111110011110","0000011000110110","↩
    0000001110011111","1111110000110111","0000101100110010","1111111010110111","↩
    1111101001100001","1111011011000100","0000001110101110","1111100011000010","↩
    0000000010110100","1111111110010010","1111111000101001","1111111111110111","↩
    1111110110000101","1111011111100111");
signal w1: vetor16bits (0 TO nInputs2-1):= ("0000000100010110","1111110101111100","↩
    0000001111110000","1111111010011110","1111000001100110","1111101011000000","↩
    0000011100110101","1111111111100010","0000001101001001","1111100001100000","↩
    0000000110111000","1111110001100010","0000001011100110","1111110000011101","↩
    0000011000010101","1111110101101101");
signal w2: vetor16bits (0 TO nInputs2-1):= ("0000000101010010","0000011010001101","↩
    1111101101011111","0000000000110000","1111100001011011","0000000110110010","↩
    1111101101100010","1111110111111111","1111111111000001","0000100111100101","↩
    1111111010111010","1111101111110110","1111101111100001","1111111011111010","↩
    1111111000011010","1111111111000111");
signal w3: vetor16bits (0 TO nInputs2-1):= ("1111110000100000","1111110110001111","↩
    1111011111110110","1111111110011011","0000100011000010","1111110001000111","↩
    0000000101000001","0000010011101110","1111101010010011","1111110001100010","↩
    1111111011111010","1111110000111011","0000000000111100","1111111110010111","↩
    1111111100111001","0000000001100001");

 begin
            neuronio0       : neuronioC2 port map (Input, w0, saida(0));
            neuronio1       : neuronioC2 port map (Input, w1, saida(1));
            neuronio2       : neuronioC2 port map (Input, w2, saida(2));
            neuronio3       : neuronioC2 port map (Input, w3, saida(3));
            saida(nNeuronios2) <= "0000000100000000"; -- 1 bias

end Behavioral;
```

B.1.6 Declaração de componentes

```
------ File components.vhd: ------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use work.tipos.all;
use work.constantes.all;
----------------------------------------------------

 PACKAGE components IS

COMPONENT macC2 IS
port(signal Input : in vetor16bits(0 to nInputs2-1);
```

```
 signal W : in vetor16bits(0 to nInputs2-1);
 signal soma        : out   signed (nbitsprod-1 downto 0));
END COMPONENT;

COMPONENT macC1 IS
port(signal Inpu : in vetor16bits(0 to nInputs1-1);
 signal W : in vetor16bits(0 to nInputs1-1);
 signal soma : out signed (nbitsprod-1 downto 0));
END COMPONENT;

COMPONENT ActFunc is
port(signal soma : in signed (31 downto 0);
 signal saida : out signed (15 downto 0));
END COMPONENT;

COMPONENT neuronioC2 is
port(signal Input : in vetor16bits (0 to nInputs2-1);
 signal W : in vetor16bits(0 to nInputs2-1);
 signal saida : out signed (nbitsi-1 downto 0));
END COMPONENT;

COMPONENT neuronioC1 is
port(signal Input : in vetor16bits (0 to nInputs1-1);
 signal W : in  vetor16bits(0 to nInputs1-1);
 signal saida    : out  signed (nbitsi-1 downto 0));
END COMPONENT;

COMPONENT camada1 is
port(signal nput : in vetor16bits (0 to nInputs1-1);
 signal saida : out vetor16bits (0 to nNeuronios1));
END COMPONENT;

COMPONENT camada2 is
port( signal Input : in vetor16bits (0 to nInputs2-1);
 signal saida : out vetor16bits (0 to nNeuronios2));
END COMPONENT;

END components;
```

B.1.7 Função de ativação

```
------ File ActFunc.vhd: ------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use work.tipos.all;
use work.LUT.all;
use work.constantes.all;
-------------------------------------------------
```

```
entity ActFunc is

    port (signal    soma        : in signed (31 downto 0);
          signal    saida       : out   signed (15 downto 0));

end ActFunc;

architecture Behavioral of ActFunc is

    -- Variaveis para a Funcao de ativacao
        constant    nbitsTab        : INTEGER :=13; --# bits para endereço da tabela
        constant    nwords      : INTEGER := 8192;  --# of entries on the LUT
    -- Endereço para a entrada da tabela
        signal address: unsigned (nbitsTab-1 downto 0):="0000000000000";
        signal LUTadd:  integer := 0; -- Inteiro
    -- Limites da função de ativação representada na tabela (com margem de segurança para ↩
        evitar as bordas)
        CONSTANT LimiteInferior: signed (nbitsi-1 downto 0) :="1111000100000000"; -- -15.0
        CONSTANT LimiteSuperior: signed (nbitsi-1 downto 0) :="0000111100000000"; -- +15.0
        CONSTANT ZeroPositivo: signed (nbitsi-1 downto 0)   :="0000000000000001"; -- ↩
            +0.003906 ou 2^(-8) a mínima resolução
        CONSTANT ZeroNegativo: signed (nbitsi-1 downto 0)   :="1111111111111111"; -- ↩
            -0.003906
        CONSTANT Zero: signed (nbitsi-1 downto 0)   :="0000000000000000"; -- 0.0

signal Output: signed (15 downto 0);
signal somaTemp:signed (15 downto 0);

begin

somaTemp<=soma(23 downto 8);
address<=conv_unsigned(soma(20 downto 8), nbitsTab);

----Activation Function-----------------------------
process(somaTemp)
begin
        if (((LimiteInferior < somaTemp) and (somaTemp < ZeroNegativo))or((ZeroPositivo < ↩
            somaTemp) and (somaTemp < LimiteSuperior))) then
            -- Transforma em inteiro para entrar na tabela
        Output<=LUTmemory(conv_integer(conv_unsigned(soma(20 downto 8), nbitsTab)));

        elsif (somaTemp <= LimiteInferior) then
            Output<="0000000000000000";
        elsif (LimiteSuperior<= somaTemp) then
            Output<="0000000100000000";
        else
            Output<="1000000000000000";
        end if;
end process;
-------------------------------------------------
saida<=Output;

end Behavioral;
```

B.1.8 Constantes

```
-------- Package: constantes.vhd ----------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
-------------------------------------------------------

PACKAGE constantes IS

    constant    nbitsi   : INTEGER := 16;    -- # of bits per word for inputs
    constant    nbitsw   : INTEGER := 16;    -- # of bits per word for weights
    constant    nbitsprod    : INTEGER := nbitsi+nbitsw; -- # of bits of input*weights
    constant    nInputs1     : INTEGER := 37;    -- # of inputs for layer 1 ( with bias)
    constant    nInputs2     : INTEGER := 16;    -- # of inputs for layer 2 ( with bias)
    constant nNeuronios1     : INTEGER := 15;    -- # of neurons in this layer
    constant nNeuronios2     : INTEGER := 4; -- # of neurons in this layer
END constantes;
```

B.1.9 Tipos

```
-------- Package: -------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use work.constantes.all;
-------------------------------------------------------
PACKAGE tipos IS

TYPE vetor16bits IS ARRAY (NATURAL range <>) OF signed (15 downto 0);
TYPE vetor32bits IS ARRAY (NATURAL range <>) OF signed (31 downto 0);

END tipos;
```

B.2 Rede Neural Matricial em FPGA

B.2.1 Programa Principal

```
------ File main.vhd: ----------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use work.constantes.all;
use work.LUT.all;
-----------------------------------

entity main is

    -- Lembrar de modificar o arquivo constantes.vhd
    port (  signal  Entrada     :   in signed(15 downto 0);
            signal  CLOCK           :   in  std_logic;
            signal  Saida           :   out std_logic;
            signal  L1          :   out std_logic;  -- LED 1 N
            signal  L2          :   out std_logic;  -- LED 2 V
            signal  L3          :   out std_logic;  -- LED 3 R
            signal  L4          :   out std_logic   -- LED 4 L
    );

end main;

architecture behavioral of main is

-- SIGMOID FUNCTION ------------------------------------------
FUNCTION sigmoid (soma: signed) RETURN signed IS

-- Constantes para a Funcao de ativacao-----------------------------

      constant    nbitsTab        : INTEGER :=13;         --# bits para endereço da tabela
      constant    nwords      : INTEGER := 8192;  --number of entries on the LUT

-- Limites da função de ativação representada na tabela (com margem de segurança para ↩
    evitar as bordas)
  CONSTANT LimiteInferior: signed (nbitsi-1 downto 0) :="1111000100000000"; -- -15.0
  CONSTANT LimiteSuperior: signed (nbitsi-1 downto 0) :="0000111100000000"; -- +15.0
  CONSTANT ZeroPositivo: signed (nbitsi-1 downto 0)   :="0000000000000001"; -- +0.003906 ↩
      ou 2^(-8) a mínima resolução
  CONSTANT ZeroNegativo: signed (nbitsi-1 downto 0)   :="1111111111111111"; -- -0.003906
  CONSTANT Zero: signed (nbitsi-1 downto 0)   :="0000000000000000"; -- 0.0

-- Variaveis para a Funcao de ativacao-----------------------------
  VARIABLE somaTemp:signed (15 downto 0);
  VARIABLE Output: signed (15 downto 0);
  -- Endereço para a entrada da tabela
  VARIABLE address: unsigned (nbitsTab-1 downto 0):="0000000000000";
  VARIABLE LUTadd:  integer := 0; -- Inteiro
```

```
BEGIN

  somaTemp:=soma(23 downto 8);
  address:=conv_unsigned(soma(20 downto 8), nbitsTab);

      if (((LimiteInferior < somaTemp) and (somaTemp < ZeroNegativo))or((ZeroPositivo < ↩
          somaTemp) and (somaTemp < LimiteSuperior)))then
          -- Transforma em inteiro para entrar na tabela
          Output:=LUTmemory(conv_integer(conv_unsigned(soma(20 downto 8), nbitsTab)));

      elsif (somaTemp <= LimiteInferior) then
          Output:="0000000000000000";
      elsif (LimiteSuperior<= somaTemp) then
          Output:="0000000100000000";
      elsif ((ZeroNegativo <= somaTemp) and (somaTemp <= ZeroPositivo)) then
          Output:="0000000010000000";
      else
          Output:="1000000000000000"; -- E
      end if;

RETURN signed(Output);
--RETURN signed(soma(15 downto 0)); -- linear
END sigmoid;

-------------------------------------------------------------------
-- Pesos da camada 1 ---------------------------------
TYPE W1Type IS ARRAY (0 to (nInputs1*nNeuronios1)-1) of signed(15 downto 0);
  constant W1: W1Type :=(
"0000000011000101", "0000000111001000", "1111111111100001", "0000001001100110", ↩
    1111111111010000", "1111111101000011", "0000000111100110", "1111111011011000", ↩
    1111111100100010", "1111110001000000", "0000000110000010", "0000000110111111", ↩
    0000000111000000", "0000000110000001", "0000000110111011", "0000000110011111", ↩
    0000000110011101", "0000000111001110", "0000000110011101", "0000000101000111", ↩
    1111111111000000", "1111111110101001", "0000000101111000", "0000000101001001", ↩
    0000000011010010", "1111111000110110", "1111111110011101", "1111110110010110", ↩
    0000000010101110", "0000000011111110", "1111111111010110", "1111111000000000", ↩
    1111110110010100", "1111111000100001", "1111111101101001", "1111111110001000", ↩
    1111110011111100",
 "0000000100101100", "0000000001001000", "0000001000101110", "0000000010000101", "↩
     0000000001110111", "0000001001110111", "1111111101000101", "1111111011000111", ↩
     0000000010001100", "1111101001011010", "0000000000001011", "0000000000111001", ↩
     0000000001011011", "0000000000010101", "0000000001011010", "0000000000101010", ↩
     0000000000011010", "0000000000000111", "0000000000111111", "1111111101100001", ↩
     1111110110011001", "1111110101000011", "1111111110111111", "1111111111101000", ↩
     1111111101001000", "1111111000000000", "1111110011011111", "1111100101011000", ↩
     1111111110001110", "0000000001110000", "1111111110100100", "1111110111011000", ↩
     1111110100111010", "1111110100000001", "1111111100001000", "1111111010011011", ↩
     1111110010010101",
 "0000000000011101", "0000000101011010", "0000000011111101", "0000000100101101", "↩
     0000000010110101", "1111111100100111", "1111111101100110", "0000000110000001", ↩
     1111111101001101", "1111111010100001", "0000000010111001", "0000000011110100", ↩
     0000000100001101", "0000000010111001", "0000000100100100", "0000000010111110", ↩
     0000000011000010", "0000000011101110", "0000000101000101", "1111111111000001", ↩
     1111111011100110", "1111110111101001", "1111110001011010", "1111110011111011", ↩
     1111110011010011", "0000000110000010", "0000000100011011", "0000010000100010", ↩
```

```
    "0000000010000010", "0000000000001010", "0000000110010111", "0000000101110000", "↩
    0000000010000011", "1111111010001001", "1111111011110111", "1111111010100010", "↩
    0000000101110101",
"1111111010001011", "0000000100000010", "1111111001011101", "1111111000000010", "↩
    1111111100101111", "0000000011101010", "1111111000011001", "1111111010110110", "↩
    0000000110000101", "0000000011100111", "0000000000111100", "0000000001110000", "↩
    0000000000010011", "0000000001111010", "0000000001000001", "0000000000100110", "↩
    0000000001111110", "0000000001101010", "0000000001110011", "0000000001010001", "↩
    1111111100110000", "1111111011110011", "0000000010000111", "0000000011110001", "↩
    0000000011101000", "1111101011001000", "0000000110010001", "0000000100101001", "↩
    0000001000001010", "0000000111100010", "0000000000100100", "1111101111011110", "↩
    1111111011111010", "0000001000010011", "0000000001000011", "1111111111101001", "↩
    0000001000110010",
"0000000100011011", "0000000100111001", "1111111111100011", "0000000000110101", "↩
    1111111110110001", "1111111001101011", "1111111101010101", "1111111001000011", "↩
    1111111110100100", "1111111111011000", "1111111111010100", "1111111111010010", "↩
    1111111101111001", "1111111101110100", "1111111110111111", "1111111110101100", "↩
    1111111111000000", "1111111110010000", "1111111111000001", "1111111101010011", "↩
    1111111011100000", "1111111000001111", "1111111001111000", "1111111100111010", "↩
    1111111100001001", "0000000000000111", "0000000001111011", "1111111111111101", "↩
    0000000001111110", "0000000000001100", "1111111110011111", "1111111001111100", "↩
    1111111010000100", "1111111111100101", "0000000001111001", "0000000010111100", "↩
    0000000100000001",
"0000000001000111", "0000000010101010", "0000000000111111", "1111111110011010", "↩
    0000000010010100", "1111111000101110", "0000000101111001", "1111111100111000", "↩
    0000000011100100", "0000000001010111", "0000000000110000", "1111111111110000", "↩
    0000000000010000", "1111111111011011", "1111111111010100", "1111111111111110", "↩
    1111111111011100", "0000000000001100", "1111111111110100", "0000000000010010", "↩
    0000000001010010", "0000000001101001", "0000000011010000", "0000000101010110", "↩
    0000000011110101", "1111111100000000", "1111111000111111", "1111111101111001", "↩
    1111111100110100", "1111111111001011", "0000000000111111", "0000000010010111", "↩
    0000001001010100", "0000000101110011", "1111111110000111", "1111111011101010", "↩
    1111111000000110",
"1111111011110001", "0000000001100001", "0000000100110100", "0000000101101100", "↩
    0000001001011110", "1111111110001001", "1111110110111111", "0000000111000101", "↩
    1111111110001110", "0000001010111001", "1111111011011101", "1111111011010100", "↩
    1111111010010000", "1111111010100100", "1111111001011000", "1111111010100000", "↩
    1111111010000010", "1111111010011000", "1111111000101001", "1111111111000110", "↩
    0000000111010110", "0000001001010011", "1111111111000111", "1111111110101001", "↩
    0000000001000000", "1111111111111000", "1111110010101111", "0000000111001000", "↩
    1111111001100010", "1111110110010011", "1111110110110011", "1111111110110111", "↩
    0000000111011010", "0000001011010010", "1111111101010110", "1111111101001011", "↩
    0000001001110101",
"1111111001001110", "1111111111100101", "1111111110111110", "1111111010111111", "↩
    0000000110111011", "0000000001101001", "0000000111100000", "1111111011001100", "↩
    0000000101111100", "0000000001101110", "1111111111100001", "1111111111010110", "↩
    1111111111110010", "0000000000100010", "1111111111101100", "1111111111011111", "↩
    0000000000101001", "0000000000111000", "1111111111100011", "0000000001101101", "↩
    0000000000010110", "1111111111000101", "1111111101100100", "1111111111010111", "↩
    0000000000011001", "1111111000110010", "0000001001101010", "0000001101101011", "↩
    0000000010000101", "1111111101010110", "1111111010110001", "1111111001011011", "↩
    0000000010001010", "0000000111100000", "0000000000110101", "0000000000000011", "↩
    1111110100010101",
"1111111110000111", "1111111001011110", "1111111111101010", "1111111101101101", "↩
    0000000001110000", "1111111010111100", "0000000010010110", "1111111100000001", "↩
```

```
    "0000000001111010", "1111111100011101", "0000000000101010", "0000000000001100", "↩
    0000000001001011", "0000000001011001", "0000000000100101", "0000000001010001", "↩
    0000000001010110", "0000000000101110", "0000000000010111", "0000000001000011", "↩
    0000000010011000", "0000000010101101", "1111111010111011", "1111111011110001", "↩
    1111111011001111", "0000001100100010", "1111111010010111", "1111111100010111", "↩
    1111111100010101", "1111111101101110", "0000000010001011", "0000001011111000", "↩
    0000000010000101", "1111111001100100", "1111111101111100", "1111111101110110", "↩
    1111111010001000",
"1111111101011011", "0000001000000110", "0000000010110100", "0000000000111101", "↩
    0000000010100010", "0000000000001100", "0000000011001101", "1111111100110110", "↩
    0000000101110101", "1111111100011000", "0000000010111011", "0000000100000000", "↩
    0000000100001100", "0000000011001110", "0000000101000000", "0000000100011100", "↩
    0000000010111110", "0000000010100110", "0000000100010000", "1111111111001001", "↩
    1111111011010010", "1111111001110111", "1111101110111010", "1111110001001010", "↩
    1111110000101101", "0000000111111100", "0000000010101100", "0000001010001110", "↩
    0000000011101011", "1111111110111001", "0000000100011111", "0000000011010000", "↩
    0000000000110000", "0000000001101000", "0000000000010101", "0000000000101001", "↩
    0000000001100101",
"0000000011000101", "1111111010100110", "1111111100110111", "0000000100111010", "↩
    0000000010010101", "1111111010100110", "1111111101011000", "1111111001010000", "↩
    0000000011001010", "1111111011000101", "0000000001111010", "0000000000011100", "↩
    0000000001010110", "0000000000100110", "0000000001000011", "0000000000001111", "↩
    0000000001101100", "0000000000101011", "0000000001000010", "0000000000100010", "↩
    1111111111100010", "1111111111110110", "1111111111100010", "0000000000110101", "↩
    0000000000010010", "0000000000000010", "1111111001101111", "0000000000001000", "↩
    0000000010111101", "0000000010000001", "0000000001111101", "1111111111111101", "↩
    1111111111011101", "1111111110110101", "1111111111100110", "1111111111010100", "↩
    1111110011011000",
"0000000010001001", "1111111111110011", "0000000001000110", "0000000010111010", "↩
    1111111010110010", "0000000000100101", "0000000010011000", "0000000111110001", "↩
    1111111001110110", "0000001111000000", "1111111111000111", "1111111111101011", "↩
    1111111111111010", "1111111110110010", "1111111111011110", "1111111110111000", "↩
    1111111110100111", "1111111111001001", "1111111111000101", "0000000010011100", "↩
    0000000101101111", "0000001100011001", "0000010000010101", "0000001011001111", "↩
    0000000110100000", "1111111111100101", "1111110011010000", "1111101000101101", "↩
    1111111001100110", "1111111100110010", "1111110110111100", "1111110110000100", "↩
    1111111010001100", "1111111011010110", "0000000000010010", "1111111101110110", "↩
    0000000000000100",
"1111110110100010", "0000000111110101", "0000000010101101", "1111111000110100", "↩
    0000000111110100", "1111111011110101", "0000000000110001", "1111111101101110", "↩
    0000000001010101", "0000000111001001", "0000000000000101", "1111111111110101", "↩
    1111111111110101", "1111111111111000", "1111111111100101", "1111111111101101", "↩
    1111111111111000", "1111111111010111", "1111111111000101", "0000000001111011", "↩
    0000000010111110", "0000000001111110", "1111111011110001", "0000000000000111", "↩
    0000000010001110", "1111111111100010", "1111110001111100", "0000011001000001", "↩
    0000000010111100", "1111111110100010", "1111111111001000", "1111111111000011", "↩
    1111111110001010", "1111111110110000", "1111111011101010", "1111111011011001", "↩
    1111101111011111",
"1111111010100000", "0000001000011001", "1111111101011001", "0000000101001000", "↩
    0000000010000100", "1111111100110100", "1111111111000100", "0000000001111111", "↩
    1111111110010110", "0000001100110101", "1111111101111000", "1111111110100000", "↩
    1111111101110011", "1111111101110011", "1111111101011010", "1111111101101001", "↩
    1111111110000100", "1111111101101010", "1111111011110110", "0000000010100111", "↩
    0000000101011011", "0000000111100011", "0000000000001111", "1111111111000100", "↩
    0000000001001100", "0000000111010011", "1111111110011001", "0000000110000000", "↩
```

```
    1111111011111100", "1111111011010000", "1111111100111010", "1111111111100001", "↩
    0000000100110110", "0000000011100100", "0000000001001101", "0000000000000110", "↩
    1111101110110011",
 "0000000000011100", "1111111010111111", "1111111001100101", "1111111011110111", "↩
    0000000011100001", "1111111111010111", "1111111010010111", "1111111010000000", "↩
    1111111000011100", "1111111111100110", "0000000000011111", "0000000001101010", "↩
    0000000001111010", "0000000000111110", "0000000000111100", "0000000001100010", "↩
    0000000000000111", "0000000000110111", "0000000000000111", "0000000100000100", "↩
    0000000100001110", "1111111111011001", "0000000000000111", "0000000001100111", "↩
    1111111110100100", "0000000010100001", "1111111110110000", "0000000111101111", "↩
    1111111111101111", "1111111010100110", "1111111000000001", "1111111000100100", "↩
    1111111010000110", "1111110011011111", "1111111111011101", "1111111100111110", "↩
    0000000001110101"
   );

TYPE p1Type IS ARRAY (0 to (nInputs1*nNeuronios1)-1) of signed(31 downto 0);
  signal P1: p1Type;

  TYPE W2Type IS ARRAY (0 to (nInputs2*nNeuronios2)-1) of signed(15 downto 0);
  constant W2: W2Type :=(
"0000000110101011", "1111111111010011", "1111111000000001", "0000010011000001", "↩
    1111110011100110", "0000000111000000", "0000001111100010", "0000011010001100", "↩
    1111110111010100", "0000000010001001", "0000000010111010", "1111100100001101", "↩
    1111110101101010", "1111111011001111", "1111011001011000", "1111100110111111",
 "0000000010000001", "1111101111110010", "1111101101100101", "1111011110111011", "↩
    1111110111101111", "0000001101010010", "0000001000100010", "0000000100111011", "↩
    1111111000011010", "1111101110100110", "0000000100001000", "0000011001111101", "↩
    0000011111100001", "0000011011010100", "0000010010111010", "1111101110010110",
 "1111111110000111", "0000010011011001", "1111101110000000", "0000010001110000", "↩
    0000001101101111", "1111110111011101", "1111101111000000", "1111110110110010", "↩
    1111101111111011", "1111101101010101", "1111111101101111", "0000010010001100", "↩
    1111110111111000", "1111101010110111", "1111110001101000", "1111111000111001",
 "1111101100000000", "0000001000010110", "0000010011111101", "1111100100010010", "↩
    0000000011001000", "1111110000010110", "1111100010001101", "1111111101111000", "↩
    0000010001110000", "0000010110101010", "0000000001001000", "1111101011111001", "↩
    1111101110010010", "0000001111110000", "0000010000100101", "1111111001011110"
   );
TYPE p2Type IS ARRAY (0 to (nInputs2*nNeuronios2)-1) of signed(31 downto 0);
  signal P2: p2Type;
-------------------------------------
--0000000100000000 -- 1.0
--0000000010000000 -- 0.5

  TYPE inputType IS ARRAY (0 to nInputs1-1) of signed(15 downto 0);
  signal Input: inputType :=(
"0000000000000000", "0000000000000000", "0000000000000000", "0000000000000000", "↩
    0000000000000000", "0000000000000000", "0000000000000000", "0000000000000000", "↩
    0000000000000000", "0000000000000000", "0000000000000000", "0000000000000000", "↩
    0000000000000000", "0000000000000000", "0000000000000000", "0000000000000000", "↩
    0000000000000000", "0000000000000000", "0000000000000000", "0000000000000000", "↩
    0000000000000000", "0000000000000000", "0000000000000000", "0000000000000000", "↩
    0000000000000000", "0000000000000000", "0000000000000000", "0000000000000000",
  "0000000100000000"); -- a ultima posição ficara com o bias de 1
```

```
  TYPE Sp1Type IS ARRAY (0 to nNeuronios1-1) of signed(31 downto 0);
  signal Sp1: Sp1Type;
  TYPE S1Type IS ARRAY (0 to nNeuronios1) of signed(15 downto 0);
  signal S1: S1Type; -- a ultima posição ficara com o bias de 1

  TYPE Sp2Type IS ARRAY (0 to nNeuronios2-1) of signed(31 downto 0);
  signal Sp2: Sp2Type;
  TYPE S2Type IS ARRAY (0 to nNeuronios2) of signed(15 downto 0);
  signal S2: S2Type;

  signal i : integer:=0;

  constant zero    :   signed  (15 downto 0)   :=  "0000000000000000";
  constant um      :   signed  (15 downto 0)   :=  "0000000100000000";
  constant meio    :   signed  (15 downto 0)   :=  "0000000010000000";

  signal temp32bSigned    :   signed(31 downto 0):="00000000000000000000000000000000";
  signal temp16bSigned    :   signed(15 downto 0);

begin

process (CLOCK)
begin
   if CLOCK='1' and CLOCK'event then
        input(i)<=Entrada;
        if i>=nInputs1 then
            i<=0;
        elsif i<nInputs1 then
            i<=i+1;
        end if;
   end if;
end process;
-- Feedforward ------------------------------------------------
-- CAMADA 1 ---------------------------------------------
  -- MAC
  MAC1lInit: FOR linha IN 0 TO (nNeuronios1-1) GENERATE
          P1(0+linha*nInputs1)<=W1(0+linha*nInputs1)*Input(0);
  END GENERATE;

  MAC1l: FOR linha IN 0 TO (nNeuronios1-1) GENERATE
      MAC1c: FOR coluna IN 1 TO (nInputs1-1) GENERATE
          P1(coluna+linha*nInputs1)<=P1((coluna-1)+linha*nInputs1)+W1(coluna+linha*↩
              nInputs1)*Input(coluna);
      END GENERATE;
  END GENERATE;

  Sp1l: FOR linha IN 0 TO (nNeuronios1-1) GENERATE
          Sp1(linha)<=P1((nInputs1-1)+linha*nInputs1);
  END GENERATE;

  -- Funcao de ativacao
  Camada1ActFunc: FOR linha IN 0 TO (nNeuronios1-1) GENERATE
      S1(linha)<=sigmoid(Sp1(linha));
  END GENERATE;
```

```
  S1(nNeuronios1)<=um; -- Adiciona o bias

-- CAMADA 2 ---------------------------------------------
  -- MAC
  MAC21Init: FOR linha IN 0 TO (nNeuronios2-1) GENERATE
          P2(0+linha*nInputs2)<=W2(0+linha*nInputs2)*S1(0);
  END GENERATE;

  MAC21: FOR linha IN 0 TO (nNeuronios2-1) GENERATE
      MAC2c: FOR coluna IN 1 TO (nInputs2-1) GENERATE
          P2(coluna+linha*nInputs2)<=P2((coluna-1)+linha*nInputs2)+W2(coluna+linha*↩
              nInputs2)*S1(coluna);
      END GENERATE;
  END GENERATE;

  Sp21: FOR linha IN 0 TO (nNeuronios2-1) GENERATE
          Sp2(linha)<=P2((nInputs2-1)+linha*nInputs2);
  END GENERATE;

  -- Funcao de ativacao
  Camada2ActFunc: FOR linha IN 0 TO (nNeuronios2-1) GENERATE
      S2(linha)<=sigmoid(Sp2(linha));
  END GENERATE;
  S2(nNeuronios2)<=um; -- Adiciona o bias

end behavioral;
```

B.2.2 Constantes

```
-------- Package: constantes.vhd ----------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
-------------------------------

PACKAGE constantes IS

    constant    nbitsi   : INTEGER := 16;    -- # of bits per word for inputs
    constant    nbitsw   : INTEGER := 16;    -- # of bits per word for weights
    constant    nbitsprod    : INTEGER := nbitsi+nbitsw; -- # of bits of input*weights
    constant    nInputs1     : INTEGER := 37;    -- # of inputs for layer 1 ( with bias)
    constant    nInputs2     : INTEGER := 16;    -- # of inputs for layer 2 ( with bias)
    constant nNeuronios1     : INTEGER := 15;    -- # of neurons in this layer
    constant nNeuronios2     : INTEGER := 4; -- # of neurons in this layer

END constantes;
```

Apêndice C

C.1 Função de ativação sigmóide reduzida

```
----- SIGMOID FUNCTION: REDUZIDA ------------------------------------------------
FUNCTION sigmoid (soma: signed(31 downto 0)) RETURN signed IS

-- Constantes para a Funcao de ativacao-------------------------------
 constant nbitsTab  : INTEGER :=8;   --# bits para endereço da tabela
 constant nwords  : INTEGER := 256;  --number of entries for the LUT = 2^nbitsTab
-- LUT
 TYPE vector_array IS ARRAY (0 TO nwords-1) OF signed (8 DOWNTO 0);
 CONSTANT LUTmemory: vector_array := ("010000100","010001000","010001100","010010000","↩
     010010100","010011000","010011100","010011111","010100011","010100111","010101010","↩
     010101110","010110001","010110101","010111000","010111011","010111110","011000001","↩
     011000100","011000111","011001010","011001100","011001111","011010001","011010100","↩
     011010110","011011000","011011010","011011100","011011110","011100000","011100001","↩
     011100011","011100101","011100110","011101000","011101001","011101010","011101011","↩
     011101101","011101110","011101111","011110000","011110001","011110001","011110010","↩
     011110011","011110100","011110101","011110101","011110110","011110110","011110111","↩
     011111000","011111000","011111000","011111001","011111001","011111010","011111010","↩
     011111010","011111011","011111011","011111011","011111100","011111100","011111100","↩
     011111100","011111101","011111101","011111101","011111101","011111101","011111110","↩
     011111110","011111110","011111110","011111110","011111110","011111110","011111110","↩
     011111110","011111111","011111111","011111111","011111111","011111111","011111111","↩
     011111111","011111111","011111111","011111111","011111111","011111111","011111111","↩
     011111111","011111111","011111111","011111111","100000000","100000000","100000000","↩
     100000000","100000000","100000000","100000000","100000000","100000000","100000000","↩
     100000000","100000000","100000000","100000000","100000000","100000000","100000000","↩
     100000000","100000000","100000000","100000000","100000000","100000000","100000000","↩
     100000000","100000000","100000000","100000000","100000000","000000000","000000000","↩
     000000000","000000000","000000000","000000000","000000000","000000000","000000000","↩
     000000000","000000000","000000000","000000000","000000000","000000000","000000000","↩
     000000000","000000000","000000000","000000000","000000000","000000000","000000000","↩
     000000000","000000000","000000000","000000000","000000000","000000000","000000001","↩
     000000001","000000001","000000001","000000001","000000001","000000001","000000001","↩
     000000001","000000001","000000001","000000001","000000001","000000001","000000001","↩
     000000001","000000001","000000010","000000010","000000010","000000010","000000010","↩
```

```
    000000010","000000010","000000010","000000010","000000011","000000011","000000011","↩
    000000011","000000011","000000100","000000100","000000100","000000100","000000101","↩
    000000101","000000101","000000110","000000110","000000110","000000111","000000111","↩
    000001000","000001000","000001000","000001001","000001010","000001010","000001011","↩
    000001011","000001100","000001101","000001110","000001111","000001111","000010000","↩
    000010001","000010010","000010011","000010101","000010110","000010111","000011000","↩
    000011010","000011011","000011101","000011111","000100000","000100010","000100100","↩
    000100110","000101000","000101010","000101100","000101111","000110001","000110100","↩
    000110110","000111001","000111100","000111111","001000010","001000101","001001000","↩
    001001011","001001111","001010010","001010110","001011001","001011101","001100001","↩
    001100100","001101000","001101100","001110000","001110100","001111000","001111100");

-- Limites da função de ativação representada na tabela (com margem de segurança para ↩
    evitar as bordas)
 CONSTANT LimiteInferior: signed (15 downto 0) :="1111101000000000"; -- -6.0
 CONSTANT LimiteSuperior: signed (15 downto 0) :="0000011000000000"; -- +6.0
 CONSTANT ZeroPositivo: signed (15 downto 0) :="0000000000010000"; -- +0.0625 ou 2^(-4) a ↩
    mínima resolução
 CONSTANT ZeroNegativo: signed (15 downto 0) :="1111111111110000"; -- -0.0625
 CONSTANT Zero: signed (15 downto 0) :="0000000000000000"; -- 0.0

-- Variaveis para a Funcao de ativacao-----------------------------
 VARIABLE somaTemp:signed (15 downto 0);
 VARIABLE Output: signed (15 downto 0);
  -- Endereço para a entrada da tabela
 BEGIN

 somaTemp:=soma(23 downto 8);

  if (((LimiteInferior < somaTemp) and (somaTemp < ZeroNegativo))or((ZeroPositivo < ↩
      somaTemp) and (somaTemp < LimiteSuperior))) then
   -- Transforma em inteiro para entrar na tabela
   Output(8 downto 0):=LUTmemory(conv_integer(conv_unsigned(soma(19 downto 12), nbitsTab))↩
       );
   Output(15 downto 9):="0000000";
  elsif (somaTemp <= LimiteInferior) then
   Output:="0000000000000000";
  elsif (LimiteSuperior<= somaTemp) then
   Output:="0000000100000000";
  elsif ((ZeroNegativo <= somaTemp) and (somaTemp <= ZeroPositivo)) then
   Output:="0000000010000000";
  else
   Output:="1000000000000000"; -- Error
  end if;

--RETURN signed(soma(15 downto 0)); -- linear
RETURN signed(Output);
END sigmoid;
-----------------------------------------------------------------
```