UNIVERSIDADE DE SÃO PAULO
Escola de Engenharia de São Carlos

IMPLEMENTAÇÃO EM *HARDWARE* DE UMA REDE NEURAL WISARD

Isabela Rodrigues do Prado Rossales

São Carlos, julho de 2012

ISABELA RODRIGUES DO PRADO ROSSALES

IMPLEMENTAÇÃO EM *HARDWARE* DE UMA REDE NEURAL WISARD

Trabalho de Conclusão de Curso apresentado à
Escola de Engenharia de São Carlos, da
Universidade de São Paulo

Curso de Engenharia de Computação com ênfase
em Sistemas Embarcados

ORIENTADOR: Prof. Dr. João Navarro Soares Jr.

São Carlos
2012

AUTORIZO A REPRODUÇÃO E DIVULGAÇÃO TOTAL OU PARCIAL DESTE TRABALHO, POR QUALQUER MEIO CONVENCIONAL OU ELETRÔNICO, PARA FINS DE ESTUDO E PESQUISA, DESDE QUE CITADA A FONTE.

Ficha catalográfica preparada pela Seção de Tratamento
da Informação do Serviço de Biblioteca – EESC/USP

R823i

Rossales, Isabela Rodrigues do Prado
Implementação em *Hardware* de uma rede neural *Wisard* / Isabela Rodrigues do Prado Rossales ; orientador João Navarro Soares Junior. -- São Carlos, 2012.

Monografia (Graduação em Engenharia de Computação com ênfase em Sistemas Embarcados) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2012.

1. *Wisard*. 2. ASIC. 3. CMOS. 4. FPGA. Título.

FOLHA DE APROVAÇÃO

Nome: Isabela Rodrigues do Prado Rossales

Título: "Implementação em Hardware de uma Rede Neural WISARD"

Trabalho de Conclusão de Curso defendido e aprovado em 02/07/2012,

com NOTA 9,5 (nove, cinco), pela comissão julgadora:

Prof. Dr. José Roberto Boffino de Almeida Monteiro - SEL/EESC/USP

Prof. Dr. Maximiliam Luppe - SEL/EESC/USP

Prof. Associado Evandro Luís Linhari Rodrigues
Coordenador pela EESC/USP do
Curso de Engenharia de Computação

Agradecimentos

Agradeço ao meu orientador por sua paciência, conselhos e por ter sido sempre atencioso durante a elaboração deste trabalho. À banca examinadora pelas valiosas sugestões para a versão final desta monografia.

À empresa Kryptus pela oportunidade de estágio, o que proporcionou diversos aprendizados relacionados a alguns tópicos deste trabalho.

Agradeço a Andre L. V. da Cunha, Fernando P. Santos e André M. de L. Curvello pelo seu apoio, amizade e pelas discussões sobre temas tão importantes para nossas vidas. Aos meus demais colegas de classe e amigos por sua constante ajuda nos estudos e nesta monografia, dos quais destaco Carlos A. de M. Massera Filho, Ricardo F. Saidel, Jonas R. Dourado, Bruno K. M. Cesar e Jonatas S. do Espírito Santo.

Agradeço aos meus pais e avós pelo carinho, compreensão, incentivo e pelo exemplo de trabalho, honestidade e fé. Aos meus irmãos Gabriel e Gabriela pelo apoio e ajuda na preparação para a defesa pública deste projeto.

Finalmente, minha gratidão a Deus, Aquele que é poderoso para fazer tudo muito mais abundantemente além daquilo que pedimos ou pensamos.

Resumo

Redes neurais artificiais (RNAs) possuem diversas aplicações, sendo indicadas, principalmente, para a resolução de problemas envolvendo reconhecimento de padrões e generalização. Entretanto, a maior parte de seus modelos envolve unidades de multiplicação, o que aumenta a complexidade de sua implementação em *hardware*. Motivados por esse problema, surgiram diversos estudos envolvendo redes neurais sem peso (RNSP). O modelo WISARD (*Wilkie, Stonham, Aleksander Recognition Device*) é uma RNSP baseada em RAMs (*Random Access Memories*) que, entre outras vantagens, possui um tempo relativamente curto de treinamento e uma estrutura lógica simples. No entanto, existem poucos resultados sobre a implementação em *hardware* desse modelo na literatura. Este trabalho envolve a descrição de uma rede WISARD parametrizada em VHDL (*Very High Speed Integrated Circuits Hardware Description Language*), síntese e desenho do *layout* na tecnologia AMS (*AustriaMicroSystems*) CMOS (*Complementary Metal-Oxide-Semiconductor*) 0,35 μm, validação em FPGA (*Field-programmable Gate Array*) Cyclone II e o desenvolvimento de uma equação relacionando os parâmetros da rede e a área mínima gerada em ASIC (*Application Specific Integrated Circuit*). A frequência máxima de operação do circuito foi de 240 MHz segundo a simulação do *layout* (modelo típico) em ASIC e de 350 MHz na implementação em FPGA. O *layout* completo do ASIC ocupou uma área de 0,329 mm^2, e a síntese para FPGA utilizou 288 células lógicas, das quais 196 possuíam registradores lógicos dedicados e 92 apenas LUTs (*Look-up Tables*). Os resultados da equação que estima a área gerada em ASIC apresentou uma correlação de 0,98 com os valores obtidos na síntese.

Palavras-chave: rede neural, WISARD, ASIC, CMOS, FPGA.

Abstract

Artificial neural networks (ANNs) have many applications and are mainly indicated to solve problems involving pattern recognition and generalization. However, most of its models involve multiplication units which increases the complexity of its implementation in hardware. Motivated by this problem, several studies involving weightless neural networks (WNN) have emerged. The WISARD (Wilkie, Stonham, Aleksander Recognition Device) model is a WNN based on RAMs (Random Access Memories) which, among other advantages, has a relatively short time of training and a simple logical structure. However, there are few results on hardware implementations of this model in the literature. This work involves the description of a parametrized WISARD network in VHDL (Very High Speed Integrated Circuits Hardware Description Language), synthesis and layout design in the AMS (AustriaMicroSystems) CMOS (Complementary Metal-Oxide-Semiconductor) 0.35 μm technology, validation on a Cyclone II FPGA (Field-programmable Gate Array) and the development of an equation relating the parameters of the network and the minimum area generated in ASIC (Application Specific Integrated Circuit). The maximum frequency of operation of the circuit was 240 MHz according to ASIC layout simulations (typical model) and 350 MHz in the FPGA implementation. The complete ASIC layout occupied an area of 0.329 mm^2, and the FPGA synthesis used 288 logical cells, of which 196 had dedicated logic registers and 92 only LUTs (Look-up Tables). The results of the equation that estimates the area generated in ASIC showed a correlation of 0.98 with the values obtained in the synthesis.

Keywords: neural network, WISARD, ASIC, CMOS, FPGA.

Lista de Figuras

2.1 Neurônio biológico. Fonte: ALTERS; ALTERS, 1999, p. 263. 4
2.2 Aprendizagem da letra I segundo o método das n-tuplas (apenas 2 n-tuplas são mostradas). Fonte: BLEDSOE; BROWNING, 1959, p. 226. 5
2.3 Matriz de memória após a aprendizagem dos caracteres B, G e 5, onde o caractere G foi apresentado duas vezes como padrão. Fonte: BLEDSOE; BROWNING, 1959, p. 227. 6
2.4 a)Estado inicial do discriminador correspondente à classe 1; b)discriminador após o primeiro treinamento; c)discriminador após o segundo treinamento. 7
2.5 Teste da classe 1 após os treinamentos das classes 1 e 2. 8

3.1 Utilização das ferramentas de auxílio a projeto. 11
3.2 Placa *Starter Development Board* da Altera. Fonte: TERASIC, 2012. 12
3.3 Interface do neurônio descrito em VHDL. 13
3.4 Interface do discriminador descrito em VHDL. 14
3.5 Interface do *top-level* descrito em VHDL. 15
3.6 Máquina de estados de Mealy correspondente à descrição do *top-level (wisard_top)*. . 15
3.7 Imagens de 16 *pixels* correspondentes às entradas no treinamento e nos testes executados por classe. 16
3.8 Imagens de 100 *pixels* utilizadas para treinamento (linha superior) e para teste (linha inferior) da classe 2. 16
3.9 Máquina de estados de Mealy do *testbench* sintetizável. 17
3.10 Diagrama de blocos do circuito sintetizado no FPGA. 18

4.1 Simulação no ModelSim para 16 entradas, 8 neurônios por discriminador e 3 classes. 19
4.2 Saída dos discriminadores para 9 testes (3 com cada classe) utilizando-se 50 neurônios por discriminador e 100 entradas. 21
4.3 Confiança relativa do resultado da rede WISARD *versus* número de neurônios por discriminador para imagens de 100 *pixels*. 22
4.4 Exemplo de treinamento e teste para uma imagem de 4 *pixels* e 1 neurônio por discriminador. 23
4.5 Exemplo de treinamento e teste para uma imagem de 4 *pixels* e 4 neurônios por discriminador. 23
4.6 Esquemático do neurônio gerado pelo LeonardoSpectrum. 24
4.7 Esquemático do discriminador gerado pelo LeonardoSpectrum. 25
4.8 Esquemático da rede WISARD gerado pelo LeonardoSpectrum. 25

4.9 *Layout* do neurônio (60 μm x 68 μm). 26
4.10 *Layout* do discriminador (267 μm x 272 μm). 27
4.11 *Layout* do *top-level* (565 μm x 582 μm). 28
4.12 Sinais de controle e *clock* para o modelo típico. 30
4.13 Saída do discriminador correspondente à classe 0 para o modelo típico. 30
4.14 Saída do discriminador correspondente à classe 1 para o modelo típico. 31
4.15 Saída do discriminador correspondente à classe 2 para o modelo típico. 31
4.16 Gráfico de dispersão em pares, onde *input* é a quantidade de *pixels*, *neuron* o número de neurônios, *address* o tamanho do endereço, *classes* o número de classes, *estim* o tamanho estimado de acordo com a equação 4.1, *size* o tamanho apresentado pelo LeonardoSpectrum, *inst* o número de instâncias e *delay* o atraso do caminho crítico. 34
4.17 Resíduos *versus* valores preditos. 36
4.18 Tamanho estimado (μm^2) *versus* número de neurônios por discriminador para um número variável de classes com uma imagem de 90 *pixels*. 36
4.19 Tamanho estimado (μm^2) *versus* número de *pixels* para um número variável de classes com 4 neurônios por discriminador. 37
4.20 Áreas estimadas segundo a Equação 4.2 e apresentadas pelo LeonardoSpectrum (μm^2) para os testes realizados. 38

Lista de Tabelas

3.1 Constantes definidas no pacote *wisard_pkg*. 13

4.1 Resposta dos discriminadores para simulação do ModelSim com 16 entradas, 8 neurônios por discriminador e 3 classes. 19

4.2 Resposta dos discriminadores para testes com imagens de 100 *pixels* e diversos números de neurônios por discriminador (N/D), onde D0, D1 e D2 são os discriminadores treinados para reconhecer as classes 0, 1 e 2, respectivamente. 20

4.3 Frequência máxima de operação e área mínima estimadas e número de instâncias geradas para diversas otimizações do LeonardoSpectrum no circuito a ser implementado em ASIC. 24

4.4 Áreas mínimas da síntese pelo LeonardoSpectrum (sem considerar as conexões), áreas finais do *layout* implementado para cada módulo e a razão entre as mesmas (*layout*/LeonardoSpectrum). 29

4.5 Correspondênia entre os sinais da simulação no Eldo e sua descrição em VHDL. . . . 29

4.6 Frequência máxima e consumo de potência do *top-level* implementado na tecnologia AMS 0,35 μm para vários modelos. 32

4.7 Correlação entre os diversos parâmetros de entrada da rede WISARD, estimativas de células de um *bit* e dados do relatório apresentados pelo LeonardoSpectrum. . . . 33

Lista de Abreviaturas

AMS	*AustriaMicroSystems*
ANN	*Artificial Neural Network*
ASIC	*Application Specific Integrated Circuit*
CI	Circuito Integrado
CMOS	*Complementary Metal-Oxide-Semiconductor*
DRC	*Design Rule Check*
FPGA	*Field-programmable Gate Array*
HDL	*Hardware Description Language*
LUT	*Look-up Table*
LVS	*Layout versus Schematic*
PLL	*Phase-locked loop*
PWL	*Piece-Wise Linear*
RAM	*Random Access Memory*
RNA	Rede Neural Artificial
RNSP	Rede Neural Sem Peso
SDL	*Schematic-driven Layout*
SOPC	*System-on-a-Programmable-Chip*
TCL	*Tool Command Language*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuits*
WISARD	*Wilkie, Stonham, Aleksander Recognition Device*
WNN	*Weightless Neural Network*

Sumário

Resumo iii

Abstract v

Lista de Figuras vii

Lista de Tabelas ix

Lista de Abreviaturas xi

1 Introdução 1

1.1 Contextualização e Motivação . . . 1
1.2 Objetivos . . . 2
1.3 Organização do Trabalho . . . 2

2 Fundamentos Teóricos 3

2.1 Redes Neurais Artificiais . . . 3
2.1.1 Introdução . . . 3
2.1.2 Redes Neurais Sem Peso . . . 5
2.1.3 WISARD . . . 6

3 Metodologia e Implementação 9

3.1 Ferramentas utilizadas . . . 9
3.2 Considerações de projeto e observações . . . 10
3.3 Implementação . . . 10
3.3.1 Python . . . 10
3.3.2 VHDL . . . 12

4 Resultados e Discussão 19

4.1 ModelSim - simulação (16 entradas) . . . 19
4.2 ModelSim - simulação (100 entradas) . . . 20
4.3 LeonardoSpectrum - síntese . . . 24
4.4 *IC-Station - Layout* . . . 26
4.5 Eldo - simulação . . . 29
4.6 Quartus II - síntese . . . 32
4.7 LeonardoSpectrum, R - análises . . . 33
4.7.1 Estimativa de células de um *bit* . . . 33

4.7.2 Correlações 33
4.7.3 Atraso do caminho crítico 33
4.7.4 Estimativa de área 35
4.7.5 Previsões de área 35

5 Conclusões **39**

Bibliografia **42**

APÊNDICE A – Código Fonte em Python **43**

APÊNDICE B – Código Fonte em VHDL **47**

APÊNDICE C – R^2-ajustado **61**

Capítulo 1

Introdução

1.1 Contextualização e Motivação

Problemas de certas categorias não podem ser facilmente resolvidos por algoritmos. Normalmente são problemas dependentes de um grande número de variáveis inter-relacionadas como, por exemplo, o problema de reconhecimento de imagens e sua classificação em grupos. Para a resolução dessa classe de problemas, a estrutura do cérebro humano demonstra ser mais apropriada, pois trabalha de modo paralelizado, além de ser capaz de aprender através de exemplos. Com o intuito de automatizar a resolução dessa classe de problemas, que funciona bem em sistemas biológicos, foram criadas as redes neurais artificiais (RNAs). Seu modelo computacional compartilha dos mesmos paralelismo e capacidade de aprendizagem não algorítimica do seu correspondente biológico. Essa capacidade possibilita generalização e associação de dados em RNAs, ou seja, a partir de um conjunto de treinamento, uma rede neural pode encontrar soluções para entradas similares, porém ainda não apresentadas, o que implica em um alto grau de tolerância a falhas quando os dados de entrada possuem certa margem de ruído (KRIESEL, 2005). Apesar do abandono do estudo de RNAs com a publicação de um livro mostrando algumas dificuldades das redes neurais modelo Perceptron por Minsky e Papert (1969[1] apud KROSE; SMAGT, 1996), a partir do final dos anos 80 o interesse pelo tema foi retomado. Atualmente existem diversos grupos de pesquisa trabalhando na área (KROSE; SMAGT, 1996) e variadas aplicações, tais como reconhecimento de imagens (LINES *et al.*, 2001), identificação e classificação de ondas cerebrais (CECOTTI; GRASER, 2011), análise de séries temporais financeiras (OLIVEIRA, 2007) etc.

Ao longo do tempo foram propostos diversos modelos de redes neurais. Em muitos deles é necessário utilizar multiplicação, o que dificulta sua implementação em *hardware*. Motivados por essa dificuldade, estudos envolvendo redes neurais sem peso (RNSP) surgiram na década de 60 (BRAGA *et al.*, 2000). A rede WISARD (*Wilkie, Stonham & Aleksander's Recognition Device*) é uma RNSP formada por discriminadores baseados em RAMs (*Random Access Memories*) que, entre outras vantagens, possui um tempo relativamente curto de treinamento e uma estrutura lógica simples (PATTICHIS *et al.*, 1994).

Linguagens de descrição de *hardware* (*Hardware Description Languages* - HDLs) são linguagens computacionais com a finalidade de descrição formal de circuitos digitais, representando o *hardware* independentemente da tecnologia alvo a ser utilizada para sua síntese. Entre as principais vantagens da utilização de HDLs estão a facilidade de descrição e simulação de circuitos complexos. VHDL (*Very High Speed Integrated Circuits* HDL) é, junto com Verilog, uma das HDLs mais empregadas na indústria e no meio acadêmico (GODSE; GODSE, 2009).

A tecnologia CMOS (*Complementary Metal-Oxide-Semiconductor*) é bastante utilizada em circuitos integrados (CIs), principalmente para o projeto de circuitos digitais, devido ao seu baixo consumo de potência, facilidade de projeto e alto nível de integração. Segundo BREWER (1998), cerca de 98% da produção de semicondutores era baseada em silício, sendo mais de 75% dos circuitos

[1]MINSKY, M.; PAPER, S. **Perceptrons: An Introduction to Computational Geometry.** The MIT Press, 1969.

feitos em CMOS (dados de 1998). O domínio da tecnologia CMOS continua atualmente: no último trimestre de 2011, a produção semanal de *wafers* em tecnologia MOS atingiu 90% da produção total (*wafers* incluindo circuitos discretos e integrados) (SIA, 2012).

Assim, a popularidade e aplicabilidade de redes neurais, HDLs e circuitos com tecnologia CMOS foram os principais fatores motivadores para a elaboração desse projeto. Outra motivação é que, embora a rede WISARD tenha sido criada com o intuito de implementação em *hardware*, a maior parte dos trabalhos encontrados na literatura a seu respeito foi feita em *software*, como os projetos de SOUZA (2011) e LINES *et al.* (2001). Além disso, as implementações em *hardware* encontradas (AZHAR; DIMOND, 2002, WILLIAMS; YORK, 1999) não mostraram resultados de área ocupada e frequência máxima de operação do circuito.

1.2 Objetivos

Esse trabalho tem como objetivos:

- a descrição parametrizada de uma rede neural modelo WISARD em linguagem VHDL;
- síntese de um exemplo da rede neural WISARD para ASIC (*Application Specific Integrated Circuit*) na tecnologia CMOS 0,35 μm da AMS (AUSTRIAMICROSYSTEMS, 2003, AUSTRIAMICROSYSTEMS, 2012);
- a realização de simulações do ASIC para verificação da frequência máxima de operação e do consumo de potência do circuito;
- a validação da descrição em VHDL através de FPGA (*Field-Programmable Gate Array*).

Através dos dados obtidos, procura-se estabelecer, adicionalmente, relações entre os parâmetros da descrição da rede e a área mínima (excetuando-se conexões de metal) do circuito implementado em ASIC.

1.3 Organização do Trabalho

O restante dessa monografia está divido como segue: o Capítulo 2 apresenta os conceitos teóricos aprendidos durante o desenvolvimento do projeto; o Capítulo 3 descreve as ferramentas utilizadas, considerações de projeto e a implementação da rede neural e dos testes; o Capítulo 4 mostra os resultados obtidos e os discute; e o Capítulo 5 apresenta a conclusão do trabalho.

Capítulo 2

Fundamentos Teóricos

Neste capítulo é apresentado um breve resumo de redes neurais artificiais, focando-se em redes neurais sem peso e, mais especificamente, na descrição da rede WISARD.

2.1 Redes Neurais Artificiais

2.1.1 Introdução

Redes neurais artificiais (RNAs) são sistemas computacionais que lembram a estrutura do cérebro humano, executando tarefas de forma não algorítmica (BRAGA *et al.*, 2000). Elas possuem em geral duas fases de operação: a fase de aprendizado ou **treinamento**, quando os parâmetros da rede são ajustados de acordo com as entradas apresentadas, e a fase de uso ou **teste**, na qual a rede é utilizada para executar alguma tarefa. Desse modo, a composição final de uma rede não é pré-determinada, sendo uma função de suas entradas de treinamento. A sua estrutura é formada por nós, também chamados de neurônios, dispostos em uma ou mais camadas interligadas por conexões denominadas axônios. Na maioria das RNAs, essas conexões possuem pesos, de forma que ponderam as entradas recebidas pelos neurônios.

As principais caracerísticas que levam à solução de problemas através de RNAs são suas capacidades de aprendizagem com um número reduzido de exemplos, e de posterior generalização, responsável por a rede responder conforme o esperado a entradas desconhecidas.

A fim de se compreender a estrutura e o funcionamento de uma rede neural artificial, será feita uma breve explicação sobre seu sistema biológico equivalente sem, entretanto, entrar em detalhes que fogem ao escopo do projeto.

Os neurônios biológicos possuem três componentes básicos: o corpo do neurônio, os dentritos e o axônio (BRAGA *et al.*, 2000), conforme mostrado na Figura 2.1. Através dos dentritos a informação chega ao corpo celular, é processada e gera uma saída que será conduzida pelo axônio até os dentritos dos próximos neurônios. A conexão entre dentritos e axônios é denominada sinapse. No estado de repouso, a ação de bombas de sódio e potássio dentro do neurônio, que enviam 3 íons de sódio (Na^+) para fora da célula a cada 2 íons de potássio (K^+) que entram, cria uma diferença de potencial de aproximadamente - 70 mV em relação ao exterior. Para que a célula produza um impulso nervoso e dispare para as células seguintes é necessário que a combinação de impulsos inibitórios e excitatórios de entrada aumentem a diferença de potencial do neurônio para - 50 mV. Nesse momento, canais da célula controlados por tensão e responsáveis pelo transporte de sódio se abrem, de forma que seus íons adentram o neurônio elevando a tensão para 30 mV. Assim, moléculas neurotransmissoras são geradas, determinando a polarização ou despolarização das próximas células. Após a geração de um impulso, o neurônio entra em um período de refração, no qual sua diferença de potencial retorna à do estado de repouso devido ao fechamento dos canais de sódio (ALTERS; ALTERS, 1999).

O primeiro modelo matemático de um neurônio foi descrito por MCCULLOCH; PITTS (1943) e é denominado MCP, sendo utilizado como base para a maioria das implementações de redes neurais. Este modelo apresenta n entradas representando os dentritos, $(x_1, ..., x_n)$, e uma saída y

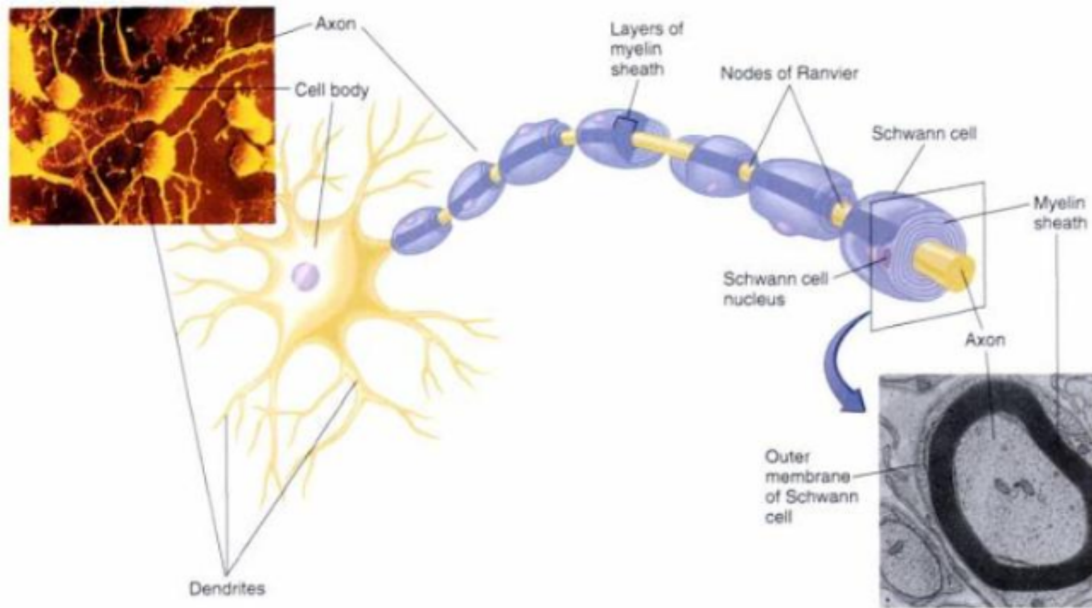

Figura 2.1: Neurônio biológico. Fonte: ALTERS; ALTERS, 1999, p. 263.

binária, representado o axônio. As sinapses são modeladas por conexões entre neurônios com pesos acoplados $(w_1, ..., w_n)$. No modelo MCP, o disparo do neurônio ocorre quando a soma ponderada de suas entradas atinge um limiar θ, conforme a Equação 2.1.

$$\sum_{i=1}^{n} x_i w_i \geq \theta \tag{2.1}$$

Além disso, no modelo MCP são feitas as seguintes simplificações (MCCULLOCH; PITTS, 1943):

- a atividade do neurônio é um processo binário;
- um número fixo de sinapses deve ser excitado dentro do período de repouso a fim de excitar um neurônio em qualquer tempo, e esse número é independente da atividade anterior e posição do neurônio;
- o único atraso significativo no sistema nervoso é o atraso sináptico;
- a atividade de qualquer sinapse inibitória previne completamente a excitação do neurônio naquele período;
- a estrutura da rede não muda com o tempo.

Entretanto, no artigo em que o modelo MCP é apresentado não há preocupação com técnicas de aprendizado, de forma que as diferenças entre o sistema biológico e o modelo MCP, principalmente a assunção de que os pesos acoplados não são ajustáveis, tornaram o modelo original bastante limitado.

Nas diversas variações do modelo MCP original, a saída não necessariamente é binária, mas representada por uma função qualquer, como rampa, degrau etc. Além disso, o número de camadas da rede (quantidade de neurônios percorridos por um dado entre qualquer entrada e saída da rede), a quantidade de neurônios por camada e o tipo de conexão entre eles (com realimentação ou não, conexão parcial ou completa) podem ser alterados. Estes parâmetros permitem a divisão das RNAs em diversas classes, como retroalimentadas, completamente conectadas etc.

Outra classificação de RNAs é baseada no modo de aprendizado, isto é, no algoritmo através do qual parâmentros da rede neural, como os pesos, são ajustados a fim de executar a tarefa desejada.

Baseado nessa classificação, as redes neurais são divididas em dois grupos: redes de aprendizado supervisionado (quando informações sobre a saída esperada são passadas para a rede através de um supervisor) e não-supervisionado (quando não há supervisor, o aprendizado depende da redundância da entrada).

2.1.2 Redes Neurais Sem Peso

Devido à dificuldade de implementação em *hardware* dos pesos das conexões entre os neurônios, uma nova classe de RNAs passou a ser estudada: as redes neurais sem peso (RNSP). O primeiro trabalho relacionado a RNSPs foi o método das n-tuplas de BLEDSOE; BROWNING (1959), cuja aplicação era o reconhecimento de caracteres alfanuméricos. O projeto consistia em um mosaico de 10x15 fotocélulas aleatoriamente agrupadas em 75 pares (n-tuplas, onde n=2) de forma exclusiva (cada fotocélula relacionada a apenas um elemento de uma n-tupla). Nesse mosaico eram projetados os caracteres, de modo que um valor binário era associado a cada fotocélula de acordo com a iluminação da mesma (Figura 2.2). Além disso, foi utilizada uma matriz de memória com palavras de 36 *bits* cujas linhas representavam todos os valores que as n-tuplas poderiam assumir, totalizando 300 (4 combinações para cada uma das 75 n-tuplas). Desse modo, cada par de fotocélulas era responsável pelo endereçamento de 4 palavras da matriz de memória. Os *bits* das palavras representavam os caracteres alfanuméricos, totalizando 36 (26 letras, 9 números e um ponto). Quando um caractere era apresentado como padrão, o nível lógico '1' era colocado na matriz, na coluna correspondente a este caractere (Figura 2.3) e em apenas uma das quatro linhas correspondentes a cada n-tupla (justamente nas linhas associadas aos valores correspondentes à iluminação das fotocélulas). Com a apresentação de outro caractere, outras linhas das n-tuplas poderiam ser também escritas seguindo as mesmas regras descritas anteriormente.

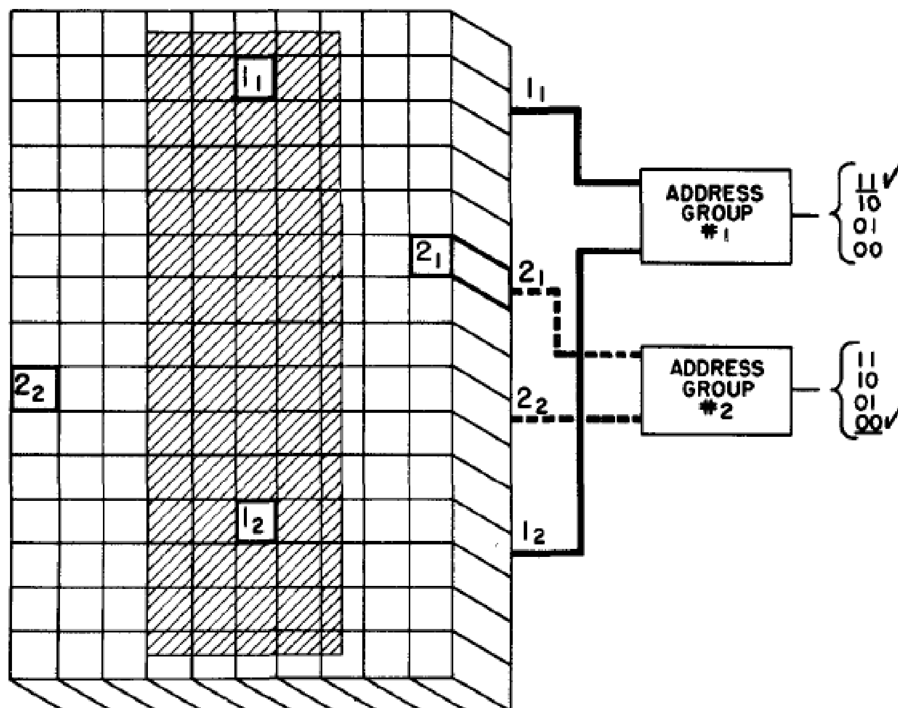

Figura 2.2: Aprendizagem da letra I segundo o método das n-tuplas (apenas 2 n-tuplas são mostradas). Fonte: BLEDSOE; BROWNING, 1959, p. 226.

A principal diferença entre as RNSPs e as demais redes neurais consiste na localização da informação aprendida. Enquanto nas RNAs convencionais a mesma se encontra nos pesos das conexões, em RNSPs a informação é armazenada em tabelas-verdade. Além disso, em redes sem peso as entradas são sempre discretas e seus neurônios são capazes de computar todas as funções booleanas

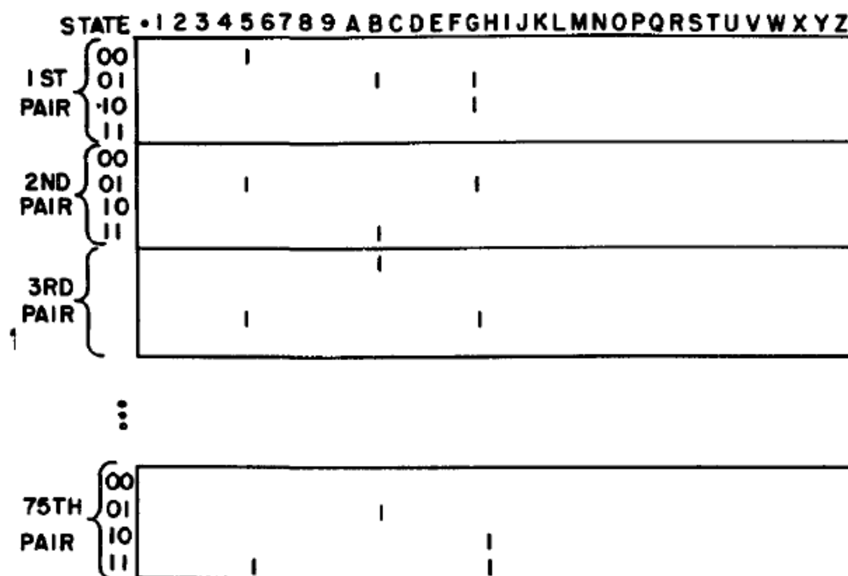

Figura 2.3: Matriz de memória após a aprendizagem dos caracteres B, G e 5, onde o caractere G foi apresentado duas vezes como padrão. Fonte: BLEDSOE; BROWNING, 1959, p. 227.

de suas entradas digitalizadas. Já as redes neurais com peso, que geralmente utilizam o modelo de neurônio *threshold neuron* com entradas e pesos assumindo qualquer valor real, possuem neurônios que computam apenas problemas com padrões de entrada formando conjuntos linearmente separáveis[1].

2.1.3 WISARD

WISARD (*Wilkie, Stonham, Aleksander Recognition Device*) é uma implementação em *hardware* ou *software* do método das n-tuplas de BLEDSOE; BROWNING (1959). Entre as suas principais vantagens estão a alta velocidade de treinamento (nas demais redes neurais normalmente o ajuste de pesos é um processo bastante demorado), a simplicidade de sua estrutura, permitindo implementação rápida e ajustável ao problema apresentado, além da redução natural dos dados de entrada devido à necessidade de quantização dos mesmos (PATTICHIS *et al.*, 1994).

A estrutura do sistema WISARD consiste em nós RAM e diversos discriminadores, cada um responsável pelo reconhecimento de um padrão.

Um **neurônio** é representado, na rede WISARD, pelo nó RAM, uma estrutura com n entradas e uma saída binárias. As entradas endereçam uma memória de tamanho 2^n com palavras de um *bit.* Durante a fase de treinamento a memória é escrita, sendo lida na fase de teste.

Cada **discriminador** representa um padrão a ser reconhecido, ou seja, corresponde a uma classe de entradas. Ele é formado por um conjunto de k neurônios RAM com n entradas cada. Portanto, o tamanho da entrada de um discriminador é de $k * n$ *bits.* Cada neurônio é conectado a n entradas dentre todas as do discriminador, de modo que o mesmo aprende apenas parte do padrão apresentado. A saída de um discriminador é dada pela soma das saídas dos seus neurônios. Em uma rede WISARD, cada discriminador pode ter um número diferente de neurônios, já que cada um representa um padrão distinto.

Para a separação de imagens em classes, que é a principal aplicação desta rede, o mapeamento entre os *pixels* da imagem e as entradas dos neurônios pode ser feito de modo aleatório. Durante a

[1] Dois subconjuntos X e Y de $\mathbb{R}^d$ são linearmente separáveis se existe um hiperplano tal que os elementos de X e Y estão em lados opostos do mesmo (ELIZONDO, 2006). No modelo de neurônio *threshold neuron*, similar ao MCP original, sua saída é dada por uma função $\phi(z)$, que apresenta resultado binário dependendo de um limiar em z, onde $z = w_0 + w_1x_1 + ... + w_nx_n$ determina a equação de um hiperplano no espaço n-dimensional (AIZENBERG, 2011). Como poucos subconjuntos de $\mathbb{R}^d$ podem ser separados por um hiperplano formado a partir de w reais, apenas um *threshold neuron* com pesos complexos pode computar problemas não linearmente separáveis.

fase de teste, o discriminador que responder com a maior soma dos resultados dos neurônios terá sua classe atribuída à imagem de entrada. Uma estimativa da qualidade dessa resposta pode ser obtida através da confiança relativa, dada pela seguinte equação:

$$C = \frac{D}{R_{jmax}} \tag{2.2}$$

onde C é a confiança relativa, R_j a resposta do j-ésimo discriminador, D a diferença entre os dois $R_j s$ mais altos e R_{jmax} o maior dos $R_j s$ (BRAGA *et al.*, 2000).

A Figura 2.4 mostra um exemplo de treinamento para um discriminador da rede WISARD com 8 neurônios por discriminador e 16 entradas. O padrão a ser reconhecido é uma imagem de 4x4 *pixels*. Inicialmente as RAMs dos neurônios possuem todas as palavras em nível lógico '0'. A cada nova entrada apresentada como pertencente à classe relacionada com esse discriminador, a memória de cada neurônio é escrita com '1' na posição endereçada pelo conjunto de entradas conectados a ele.

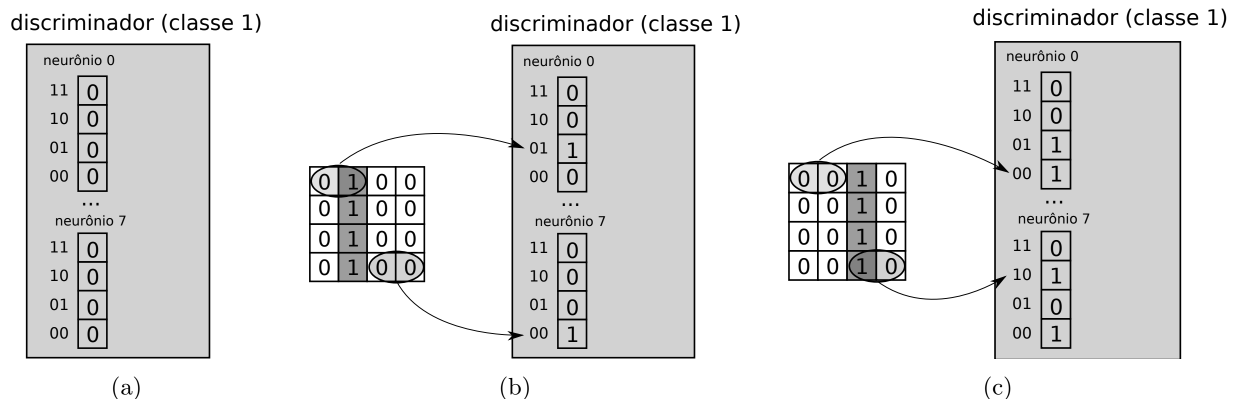

Figura 2.4: a)Estado inicial do discriminador correspondente à classe 1; b)discriminador após o primeiro treinamento; c)discriminador após o segundo treinamento.

Considerando a rede com apenas dois discriminadores (classes), o teste para o reconhecimento da classe 1 após o treinamento de ambas encontra-se na Figura 2.5. Para cada discriminador, o conteúdo da posição de memória de cada neurônio que for endereçada pelas entradas do teste é somado e apresentado como saída do mesmo. Um comparador é então usado para escolher o discriminador que apresenta a maior soma. A classe correspondente à esse discriminador é, então, atribuída à imagem de entrada.

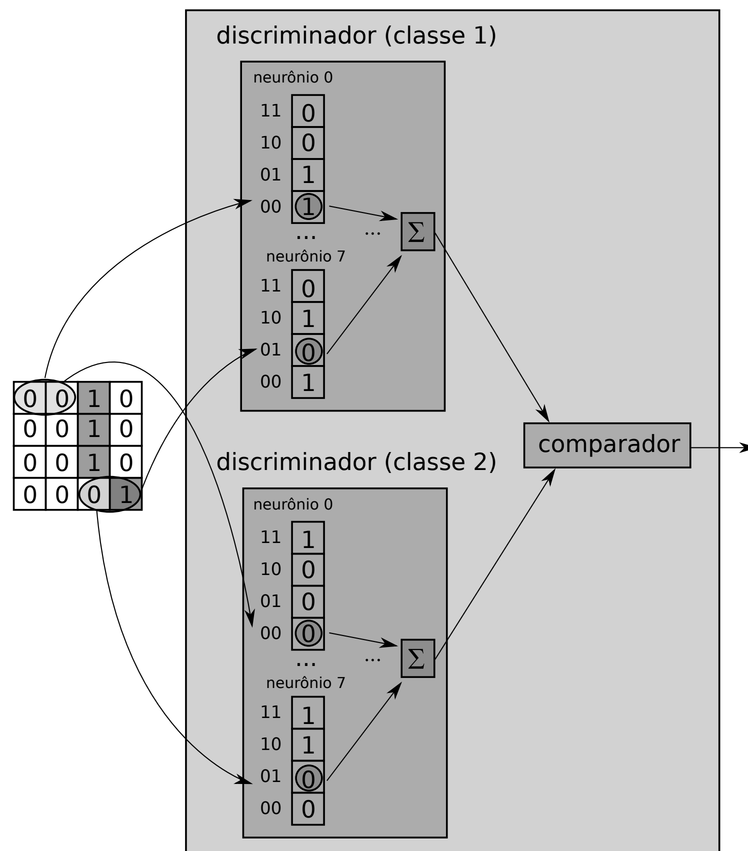

Figura 2.5: Teste da classe 1 após os treinamentos das classes 1 e 2.

Capítulo 3

Metodologia e Implementação

Neste capítulo são descritas as ferramentas de projeto utilizadas, considerações de projeto e as implementações em Python e VHDL.

3.1 Ferramentas utilizadas

Neste trabalho foram utilizadas algumas ferramentas de auxílio a projeto, descritas a seguir:

- ModelSim é uma ferramenta utilizada para simulação de linguagens de descrição de *hardware*, com suporte para as linguagens VHDL e Verilog, além de permitir o uso de *scripts* TCL (*Tool Command Language*) (MENTOR GRAPHICS, 2012).

- O *software* Quartus II provê um ambiente de desenvolvimento para *system-on-a-programmable-chip* (SOPC), incluindo *design* baseado em diagrama de blocos ou HDL, síntese, *routing* e programação de FPGA (ALTERA, 2010).

- LeonardoSpectrum é uma suíte de ferramentas de *design* para FPGA e ASIC, podendo ter entradas em VHDL ou Verilog e oferecendo síntese lógica de circuitos com análise de temporização e otimização baseada em restrições (MENTOR GRAPHICS, 1999).

- A suíte *IC Station* é composta por três pacotes de aplicação: *ICgraph Basic*, *IC Station Schematic-driven Layout (SDL)* e *ICassemble* (MENTOR GRAPHICS, 2009). O *ICgraph Basic* consiste em um conjunto de funções básicas de edição de polígonos para o desenvolvimento manual de *layouts* de circuitos integrados (CIs). Já o *IC Station SDL* permite a criação automática do *layout* através da utilização das informações de conectividade no esquemático, aumentando significativamente a produtividade em relação ao *design* manual. Finalmente, o *ICassemble* possui um conjunto de funções para o planejamento da localização das células (*floorplanning*) e *routing* interativo.

- Calibre LVS (*Layout Versus Schematic*) e DRC (*Design Rule Check*) são ferramentas utilizadas na verificação física do *layout*, isto é, se o mesmo se conforma com o esquemático do circuito (LVS) e com as regras de fabricação (DRC) (MENTOR GRAPHICS, 2005).

- *Design Architect - IC* é um aplicativo de captura de esquemático com um ambiente direcionado à otimização da criação de circuitos integrados. A ferramenta proporciona captura e verificação de esquemáticos, descrição integrada de conexões, instâncias e atributos (*Netlist*) SPICE, suporta atalhos definidos pelo usuário, permite a criação de *designs* hierárquicos utilizando metodologia *bottom-up* ou *top-down* e a edição de arquivos nas linguagens VHDL, VHDL-AMS, Verilog, Verilog-A, SPICE, HSPICE e EldoSPICE. As ferramentas *IC Station* podem também ser invocadas diretamente do *Design Architect* (MENTOR GRAPHICS, 2003).

- Eldo é um simulador de circuitos integrados baseado em SPICE. Possui tecnologia *multi-threading* e é suportado por todas as principais *foundries* do mundo, cobrindo os modelos mais importantes de dispositivos (MENTOR GRAPHICS, 2011a).
- O *software* EZwave é um visualizador de forma de onda avançado, com suporte nativo para diversos simuladores, dentre eles Eldo Classic e Questa. Ele permite a análise de vários tipos de formas de onda nos domínios da frequência e tempo, como diagrama de olho, carta de Smith e histogramas (MENTOR GRAPHICS, 2011b).
- R é uma linguagem e um ambiente para computação estatística que possui diversas facilidades gráficas e testes clássicos de análise de dados, podendo ser facilmente estentido através de pacotes (R-PROJECT, 2012).

A Figura 3.1 mostra como foram utilizadas as ferramentas descritas durante a execução deste projeto.

3.2 Considerações de projeto e observações

Para que o funcionamento da rede fosse testado, foi necessário encontrar algum problema que pudesse ser resolvido por ela. O reconhecimento de imagens de caracteres escritos à mão é uma aplicação clássica da rede WISARD e, portanto, foi escolhido como aplicação da rede implementada neste projeto.

O número de *pixels* da imagem escolhido (16) para a implementação em ASIC foi o mínimo possível de modo que ainda permitisse uma variação nas imagens de entrada mas não deixasse o *layout* muito grande. Para facilitar a descrição da rede, todos os discriminadores possuem o mesmo número de neurônios. O número de neurônios por discriminador (8) é o que gerou a menor quantidade de instâncias na síntese do LeonardoSpectrum. Como o número de *pixels* é pequeno, o reconhecimento de muitos caracteres se tornaria inviável, pois poderia haver sobreposição dos contornos dos mesmos. Portanto, foram escolhidas apenas três classes para serem reconhecidas, representando os caracteres 0, 1 e 2.

A partir desta seção, quando for citado o número de entradas da rede, subentende-se que o mesmo exclui os sinais de controle, *clock* e *reset*, representando apenas os sinais correspondentes aos *pixels* da imagem.

O FPGA utilizado para validação da rede neural foi o Cyclone II (tecnologia 90 nm), devido à disponibilidade da placa *Starter Development Board* (Figura 3.2), da Altera, que contém o mesmo. A implementação em ASIC foi feita na tecnologia AMS CMOS 0,35 μm por ser essa a tecnologia aprendida e aplicada nas disciplinas da graduação. Todas as simulações no Eldo (típico, *worst power* e *worst speed*) utilizaram o modelo BSIM3v3 (*Berkley Short-channel IGFET Model*).

3.3 Implementação

3.3.1 Python

A seguir encontra-se uma descrição dos códigos implementados em Python, disponíveis no Apêndice A.

Gerador de *testbenches* em VHDL (*tb_gen.py*)

De modo a facilitar a geração de *testbenches* em VHDL para um número grande de entradas da rede neural, foi implementado em Python um programa que lê um arquivo de imagem *bmp* em tons de cinza com 10x10 *pixels* e imprime, na saída padrão, uma descrição de sinais de forma equivalente a um arquivo de *testbench* em VHDL. Para que as entradas da rede neural fossem binárias, cada *pixel* da imagem lida é convertido em '0' ou '1' de acordo com um limiar no nível de intensidade do mesmo. As imagens a serem utilizadas nos treinamentos e nos testes podem ser

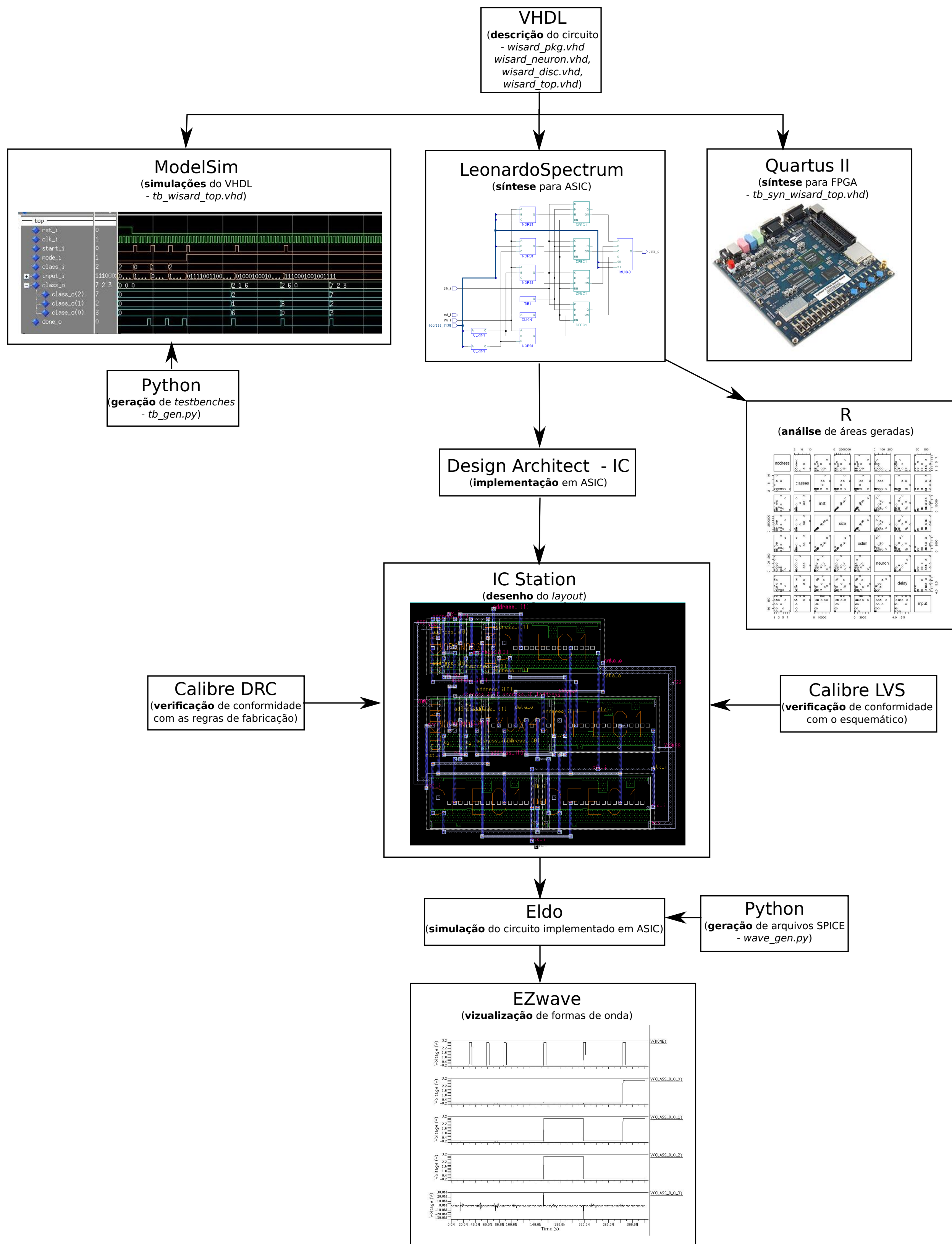

Figura 3.1: Utilização das ferramentas de auxílio a projeto.

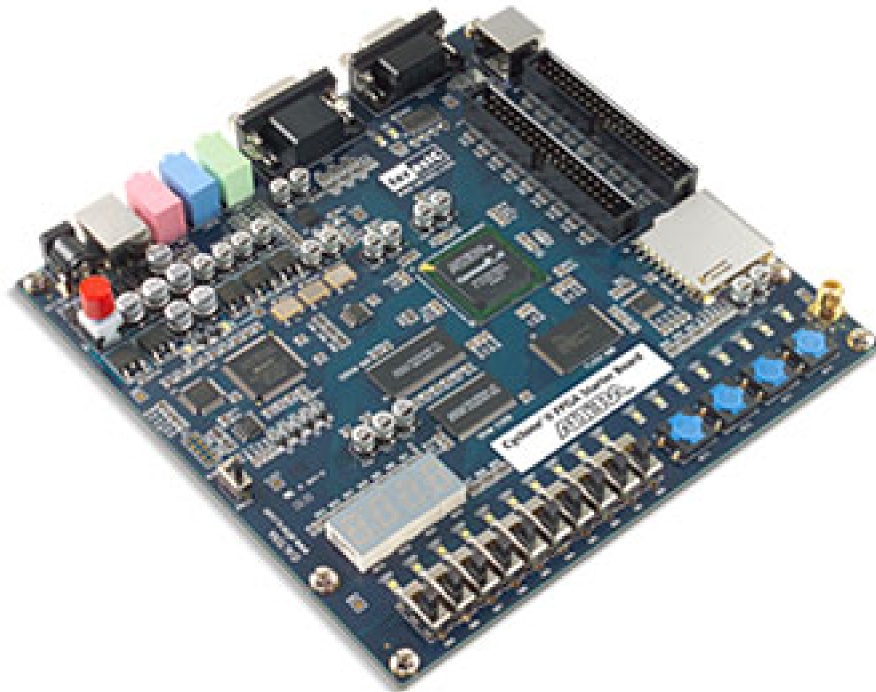

Figura 3.2: Placa *Starter Development Board* da Altera. Fonte: TERASIC, 2012.

alteradas facilmente na função principal. Uma descrição das funções mais importantes que compõem o programa encontra-se a seguir:

- **read_file:** recebe como parâmetro o nome do arquivo de imagens *bmp* que então é lido e retorna um vetor de zeros e uns de acordo com um limiar no nível de intensidade do *pixel*;
- **train:** recebe como parâmetros o nome de um arquivo e uma classe, chama a função *read_file* e imprime sinais de entrada em VHDL relativos à classe especificada e aos *pixels* do arquivo lido;
- **test:** recebe como parâmetro o nome de um arquivo, chama a função *read_file* e imprime os sinais de entrada da rede em VHDL baseados no arquivo lido, colocando-a no modo de teste.

Gerador de arquivo SPICE (*wave_gen.py*)

Como a simulação do circuito implementado em ASIC envolvia vários sinais de entrada que eram alterados em instantes de tempo diferentes, um gerador de arquivos SPICE (extensão *.cir*) em Python foi criado a fim de se evitar erros na criação manual do mesmo, erros estes que só seriam descobertos ao fim de uma longa simulação.

A principal função utilizada nesse programa é a *gen_wave*, que recebe como parâmetros o nome do sinal, uma lista contendo em quais pulsos de *clock* o sinal deve ser alterado e outra com os valores de tensão nesses pulsos. Esta função gera, assim, uma onda do tipo PWL (*Piece-Wise Linear*).

3.3.2 VHDL

Os módulos descritos em VHDL (Apêndice B) seguiram uma estrutura hierárquica, com *wisard_neuron* sendo o módulo de nível mais baixo, seguido de *wisard_disc* e *wisard_top*. A fim de simular a operação do *top-level*, foi criado o *testbench tb_wisard_top*. Para os testes em FPGA, foi também descrito um módulo de *testbench* sintetizável, o *tb_syn_wisard_top*. Além disso, um pacote (*package*) foi definido de modo que a rede neural fosse completamente parametrizada. A seguir, encontra-se uma descrição desse pacote e dos demais módulos.

Pacote wisard_pkg

As constantes que foram criadas nesse pacote estão descritas na Tabela 3.1 e são utilizadas como parâmetro nos demais módulos.

Tabela 3.1: Constantes definidas no pacote *wisard_pkg*.

Constante	Significado	Valor usado para a implementação em ASIC
CLASS_NUMBER	Número de classes	3
INPUTS_NUMBER	Número de entradas	16
NEURONS_NUMBER	Número de neurônios	8
SUM_SIZE	Número mínimo de *bits* para descrever NEURONS_NUMBER	4
ADD_SIZE	Tamanho do endereço dos neurônios: $\frac{INPUTS_NUMBER}{NEURONS_NUMBER}$	2
WORDS_NUMBER	Tamanho da RAM de cada neurônio: 2^{ADD_SIZE}	4

Módulo wisard_neuron

Este módulo descreve um neurônio como uma memória com endereçamento de ADD_SIZE *bits* e tamanho de WORDS_NUMBER. A Figura 3.3 mostra a interface do neurônio descrito. Os sinais de entrada e saída do bloco são:

- *clk_i:* sinal de entrada de sincronismo (*clock*);
- *rst_i:* sinal de controle *reset* (nível alto);
- *rw_i:* sinal de controle de leitura (nível alto) e escrita (nível baixo);
- *address_i:* sinais de entrada que fornecem o endereço da memória;
- *data_o:* sinal de saída de dados.

Neste bloco a escrita é síncrona, sendo que o dado a ser escrito é sempre o nível lógico '1'; a leitura de dados e o *reset*, que zera a memória inteira, são assíncronos.

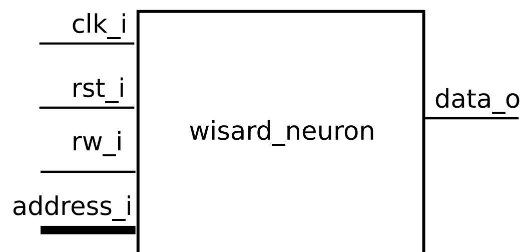

Figura 3.3: Interface do neurônio descrito em VHDL.

Módulo wisard_disc

Este módulo consiste na descrição de um discriminador com número de neurônios e entradas parametrizado, conforme descrito na Tabela 3.1. A interface do discriminador é apresentada na Figura 3.4. Os sinais de entrada e saída do bloco são:

- *clk_ i:* sinal de entrada de sincronismo (*clock*);
- *rst_ i:* sinal de controle *reset* (nível alto);
- *start_ i:* sinal que inicia o teste ou treinamento do discriminador (nível alto);
- *mode_ i:* sinal de controle de teste (nível alto) ou treinamento (nível baixo);
- *input_ i:* sinais de entrada da rede representando os *pixels* de uma imagem;
- *done_ o:* sinal que indica o término da operação de teste ou treinamento;
- *sum_ o:* sinais que representam a soma das saídas dos neurônios quando em teste.

O discriminador é baseado em uma máquina de estados, de forma que caso o sinal *start_ i* esteja em nível alto, o estado é alterado de acordo com o modo da rede em treinamento (*mode_ i* = '0') ou em teste (*mode_ i* = '1').

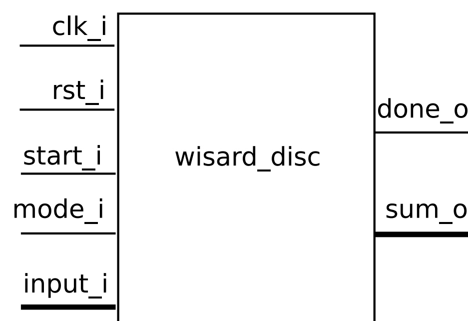

Figura 3.4: Interface do discriminador descrito em VHDL.

Módulo wisard_ top

Este módulo consiste na descrição da entidade *top-level* da rede neural WISARD, utilizando-se dos parâmetros descritos na Tabela 3.1. A Figura 3.5 mostra a interface do bloco criado. Os sinais de entrada e saída do bloco são:

- *clk_ i:* sinal de entrada de sincronismo (*clock*);
- *rst_ i:* sinal de controle *reset* (nível alto);
- *start_ i:* sinal que inicia o teste ou treinamento da rede (nível alto);
- *mode_ i:* sinal de controle de teste (nível alto) ou treinamento (nível baixo);
- *input_ i:* sinais de entrada da rede representando os *pixels* de uma imagem;
- *class_ i:* sinais de entrada representando a classe a ser treinada;
- *done_ o:* sinal que indica o término da operação de teste ou treinamento;
- *class_ o:* sinais de saída indicando os resultados de cada discriminador.

A máquina de estados correspondente à implementação de *wisard_ top* encontra-se na Figura 3.6. Cada transição mostrada ocorre durante a borda de subida do *clock*. Durante o *reset*, assíncrono, a máquina retorna ao estado IDLE. Dependendo do modo da rede, o próximo estado será o de treinamento (S1) ou de teste (S2). O sinal de saída *class_ o* é um vetor de inteiros que mostra o resultado de cada discriminador para a entrada apresentada. Quando em modo de treinamento, deve ser apresentada a classe correspondente à entrada através do barramento *class_ i*.

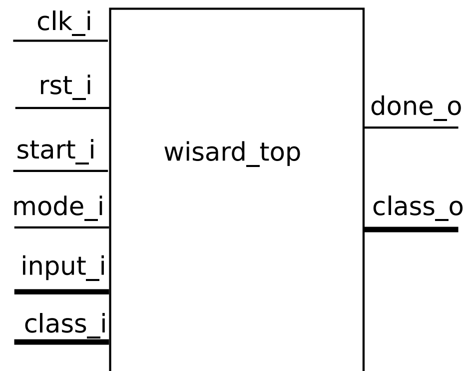

Figura 3.5: Interface do *top-level* descrito em VHDL.

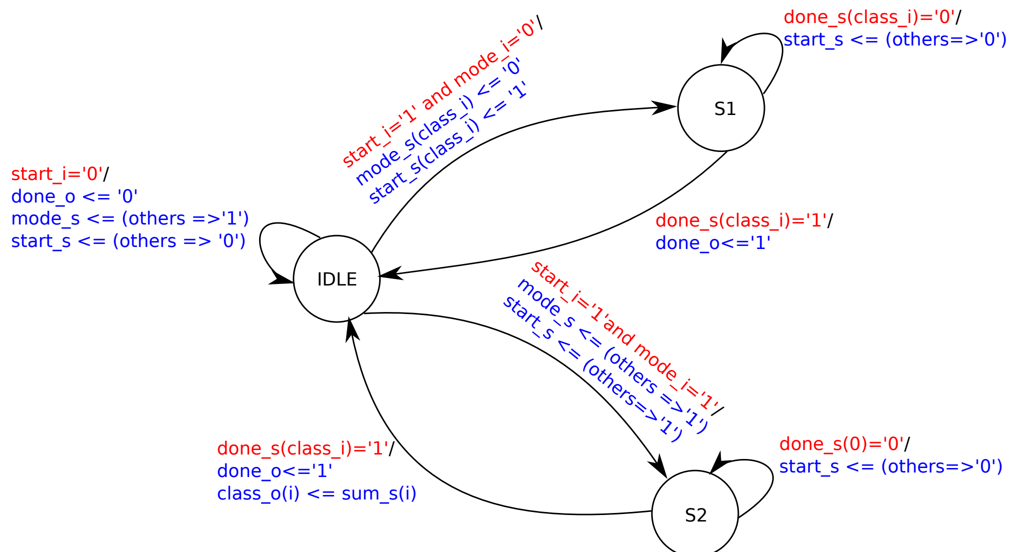

Figura 3.6: Máquina de estados de Mealy correspondente à descrição do *top-level (wisard_ top)*.

Módulo tb_wisard_top

O módulo *tb_wisard_top* foi criado com o intuito de simular o funcionamento da entidade *top-level* (*wisard_top*) da rede WISARD. Esse módulo não é sintetizável e foi utilizado apenas no ModelSim. Ele não possui portas, apenas sinais internos correspondentes aos sinais de entrada e saída da rede neural instanciada descritos na seção 3.3.2.

Com o módulo *tb_wisard_top* foram realizadas diversas simulações: algumas utilizando os mesmos parâmetros do circuito que seria implementado em ASIC e outras com uma RNA de 100 entradas.

Na simulação do circuito que seria implementado em ASIC, três classes são aprendidas com um treinamento cada e as mesmas são testadas posteriormente com entradas não utilizadas no treinamento. Cada novo teste ou treinamento é realizado após o sinal de concluído (*done_s*) estar em nível alto. A Figura 3.7 mostra as imagens correspondentes às entradas de treinamento e testes. Para estas entradas, teremos que se o primeiro *pixel* for branco, *input_s(0)* <= '0', se for preto, *input_s(0)* <= '1' e assim sucessivamente.

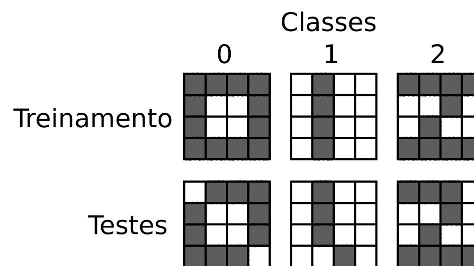

Figura 3.7: Imagens de 16 *pixels* correspondentes às entradas no treinamento e nos testes executados por classe.

A fim de se verificar o funcionamento do circuito para outros parâmetros, novos testes foram criados através do gerador de *testbenches* implementado em Python. O número de entradas foi alterado de 16 para 100 e diversos números de neurônios por discriminador foram escolhidos. Um exemplo das imagens correspondentes às entradas de *wisard_top* encontra-se na Figura 3.8. Conforme descrito na seção 3.3.1 um limiar foi utilizado para que tons de cinza mais escuro fossem convertidos em nível lógico '1' e os mais claros em '0' através do *software tb_gen.py*.

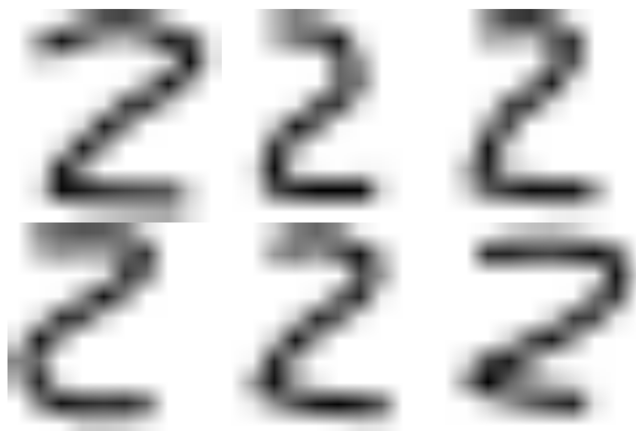

Figura 3.8: Imagens de 100 *pixels* utilizadas para treinamento (linha superior) e para teste (linha inferior) da classe 2.

Módulo tb_syn_wisard_top

Este módulo foi criado a fim de ser sintetizado em FPGA, de modo que o correto funcionamento do circuito pudesse ser testado e sua frequência máxima obtida. As entradas utilizadas foram as

mesmas para o teste do *layout* em ASIC. Os sinais de entrada, de saída e internos desse bloco são:

- *clk_i:* sinal de entrada de sincronismo (*clock*);
- *rst_i:* sinal de controle *reset* (nível alto);
- *led1_o:* sinal de saída indicando que a classe 1 foi reconhecida corretamente (nível alto);
- *led2_o:* sinal de saída indicando que a classe 2 foi reconhecida corretamente (nível alto);
- *led3_o:* sinal de saída indicando que a classe 3 foi reconhecida corretamente (nível alto);
- *start_s:* sinal que inicia o teste ou treinamento da rede (nível alto);
- *mode_s:* sinal de controle de teste (nível alto) ou treinamento (nível baixo);
- *input_s:* sinais de entrada da rede representando os *pixels* de uma imagem;
- *class_i_s:* sinais de entrada representando a classe a ser treinada;
- *done_s:* sinal que indica o término da operação de teste ou treinamento;
- *class_o_s:* sinais de saída indicando os resultados de cada discriminador;
- *results_s:* sinais correspondentes aos esperados para *class_o_s*.

O módulo *tb_syn_wisard_top* é baseado na máquina de estados mostrada na Figura 3.9. Durante o *reset*, a máquina é colocada no estado IDLE, o sinal para o início da rede (*start_s*) e os sinais *ledX_o* são colocados em nível baixo e seu modo em treinamento (*mode_s=‘0’*). Cada transição representada ocorre durante a borda de subida do *clock*. A fim de simplificar o diagrama, os sinais de entrada da rede correspondentes aos *pixels* das imagens a serem treinadas ou testadas não estão descritos no mesmo.

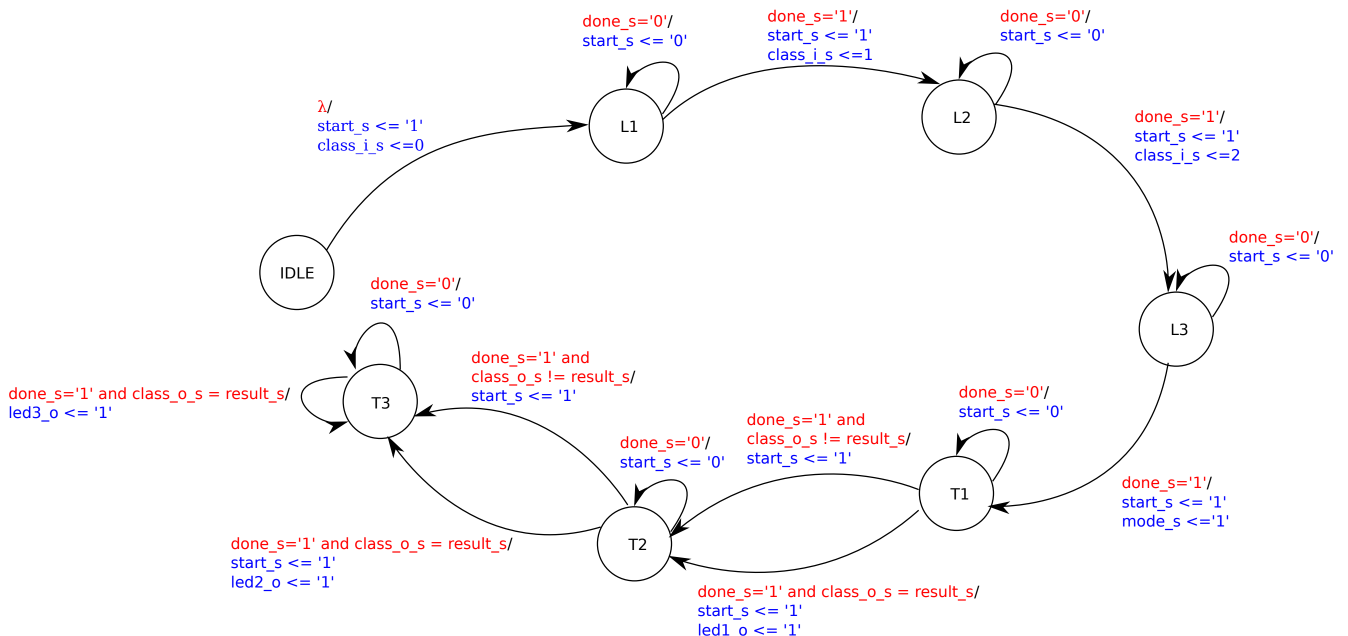

Figura 3.9: Máquina de estados de Mealy do *testbench* sintetizável.

Nos estados IDLE, L1 e L2 são iniciados os treinamentos das classes 0, 1 e 2 respectivamente. No estado L3 o último treinamento é concluído e o teste da classe 0 é iniciado. Durante os estados T1, T2 e T3, caso o resultado dos testes seja igual ao esperado (que está armazenado em um vetor), os sinais *led1_o*, *led2_o* e *led3_o* são, respectivamente, colocados em nível alto. As portas *clk_i* e

rst_i foram conectadas à saída de *clock* do bloco PLL (*Phase-locked Loop*) da Altera *ALTPLL* e de um *switch* da placa utilizada. Os sinais *ledX_o* foram conectados a três LEDs. O diagrama de blocos do circuito sintetizado com esse módulo se encontra na Figura 3.10.

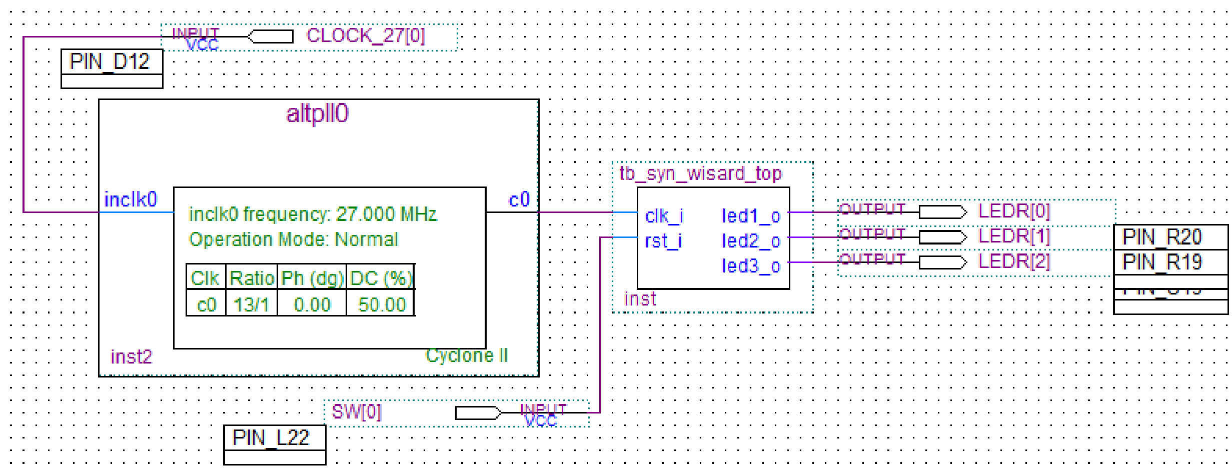

Figura 3.10: Diagrama de blocos do circuito sintetizado no FPGA.

Capítulo 4

Resultados e Discussão

Neste capítulo são apresentados e discutidos os resultados do projeto.

4.1 ModelSim - simulação (16 entradas)

Utilizando-se do *testbench* *tb_wisard_top*, foi simulado no ModelSim o funcionamento da RNA WISARD descrita em VHDL com os mesmos parâmetros que foram utilizados na síntese para ASIC, isto é, 16 entradas, 3 discriminadores e 8 neurônios por classe. Inicialmente as 3 classes são aprendidas (sinais $class_i$ = 0, 1 e 2; sinal $mode_i$ = '0') e posteriormente testadas (sinal $mode_i$ = '1'). Os resultados dos discriminadores para cada classe encontram-se nos sinais $class_o$.

Conforme esperado, durante o teste com entradas de sua respectiva classe, cada discriminador respondeu com o maior valor (Figura 4.1 e Tabela 4.1).

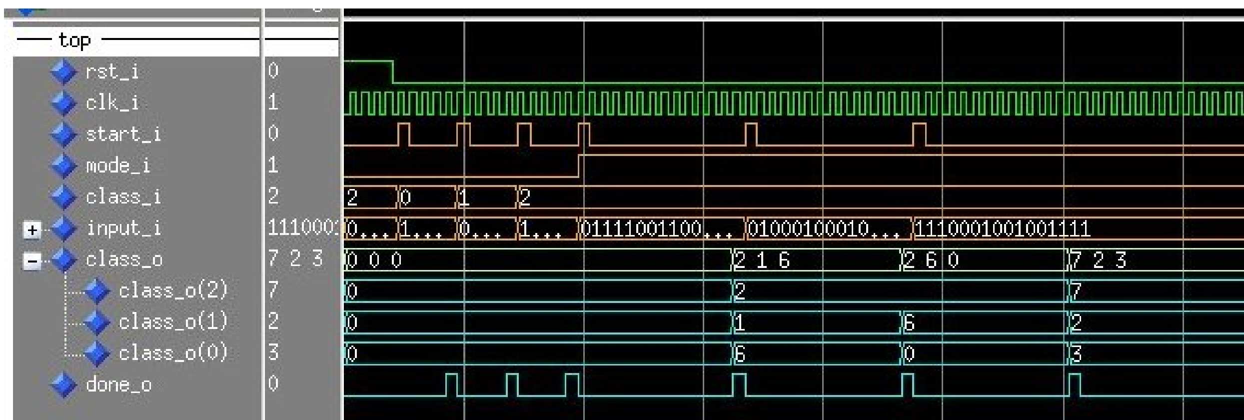

Figura 4.1: Simulação no ModelSim para 16 entradas, 8 neurônios por discriminador e 3 classes.

Tabela 4.1: Resposta dos discriminadores para simulação do ModelSim com 16 entradas, 8 neurônios por discriminador e 3 classes.

Discriminante	Teste - classe 0	Teste - classe 1	Teste - classe 2
2	2	2	**7**
1	1	**6**	2
0	**6**	0	3

4.2 ModelSim - simulação (100 entradas)

Novas simulações no ModelSim foram realizadas para três redes WISARD com 100 entradas, 3 discriminadores e 10, 25 e 50 neurônios por discriminador. Ao todo foram usadas 18 imagens, 6 para cada classe (3 para treinamento e 3 para teste), como entradas do programa *tb_gen.py*, responsável pela criação do *testbench* em VHDL (*tb_wisard_top*) utilizado nestas simulações. A alteração do número de neurônios para cada simulação foi feita mudando-se apenas os parâmetros descritos no pacote *wisard_pkg*. Portanto, não foi necessária a alteração do *testbench* gerado, sendo utilizadas as mesmas imagens nas três simulações.

Os resultados das simulações bem como a confiança relativa dos mesmos (Equação 2.2) encontram-se na Tabela 4.2 e nas Figuras 4.2 e 4.3. A Figura 4.2 mostra as formas de onda geradas para a simulação com 50 neurônios por discriminador.

Tabela 4.2: Resposta dos discriminadores para testes com imagens de 100 *pixels* e diversos números de neurônios por discriminador (N/D), onde D0, D1 e D2 são os discriminadores treinados para reconhecer as classes 0, 1 e 2, respectivamente.

N/D	Classes testadas (número do teste)	D0	D1	D2	Confiança relativa
10	0 (1)	2	0	0	1,00
	0 (2)	4	3	2	0,25
	0 (3)	3	0	1	0,67
	1 (1)	4	10	3	0,60
	1 (2)	2	10	1	0,80
	1 (3)	4	10	2	0,60
	2 (1)	1	2	3	0,33
	2 (2)	1	1	6	0,83
	2 (3)	2	2	3	0,33
25	0 (1)	17	2	7	0,59
	0 (2)	18	8	9	0,50
	0 (3)	17	3	8	0,53
	1 (1)	11	25	17	0,32
	1 (2)	9	25	15	0,40
	1 (3)	11	25	16	0,36
	2 (1)	13	12	17	0,24
	2 (2)	11	11	23	0,52
	2 (3)	12	10	17	0,29
50	0 (1)	42	21	26	0,38
	0 (2)	45	28	33	0,27
	0 (3)	44	22	27	0,39
	1 (1)	40	50	42	0,16
	1 (2)	38	50	40	0,20
	1 (3)	40	50	42	0,16
	2 (1)	39	34	40	0,03
	2 (2)	34	33	48	0,29
	2 (3)	39	32	43	0,09

Novamente, cada discriminador teve uma resposta maior quando apresentada uma entrada da classe para a qual foi treinado reconhecer.

Figura 4.2: Saída dos discriminadores para 9 testes (3 com cada classe) utilizando-se 50 neurônios por discriminador e 100 entradas.

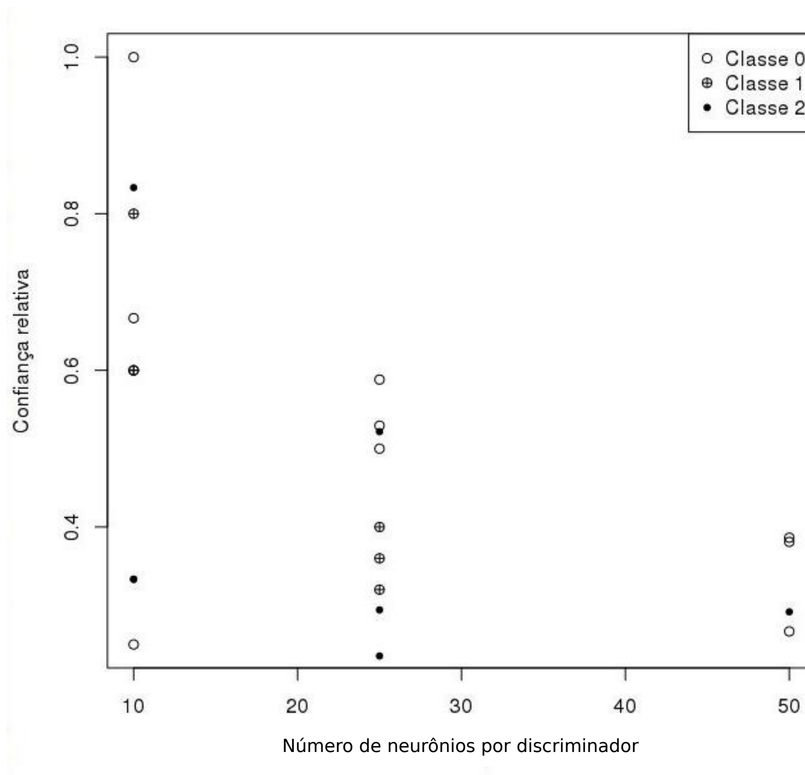

Figura 4.3: Confiança relativa do resultado da rede WISARD *versus* número de neurônios por discriminador para imagens de 100 *pixels*.

Conforme se pode observar, a confiança relativa diminui à medida que o número de neurônios por discriminador se aproxima do número de entradas da rede. Esse resultado era esperado, dado que o aumento do número de neurônios faz com que cada um deles seja responsável pelo reconhecimento de uma quantidade menor de *pixels* da imagem. Dessa forma, a probabilidade de um neurônio responder com '1' aumenta, pois o tamanho da RAM diminui.

As Figuras 4.4 e 4.5 foram criadas com o intuito de exemplificar esse efeito. Considerando-se que o número de entradas da rede é 4, essas figuras mostram um treinamento e um teste realizados para uma classe. Para uma rede com apenas um neurônio (Figura 4.4), a saída do discriminador é a mesma do neurônio, 0. Para uma rede com 4 neurônios (Figura 4.5), a saída é a soma das saídas dos neurônios, 3. Pode-se ver, então, que o discriminador com apenas um neurônio só reconheceria a imagem de teste caso fosse igual à entrada apresentada como treinamento, enquanto que o discriminador com mais neurônios tende a reconhecer mais imagens como pertencendo à sua classe. Portanto, quando o número de neurônios por discriminador aumenta, cada discriminador perde especificidade.

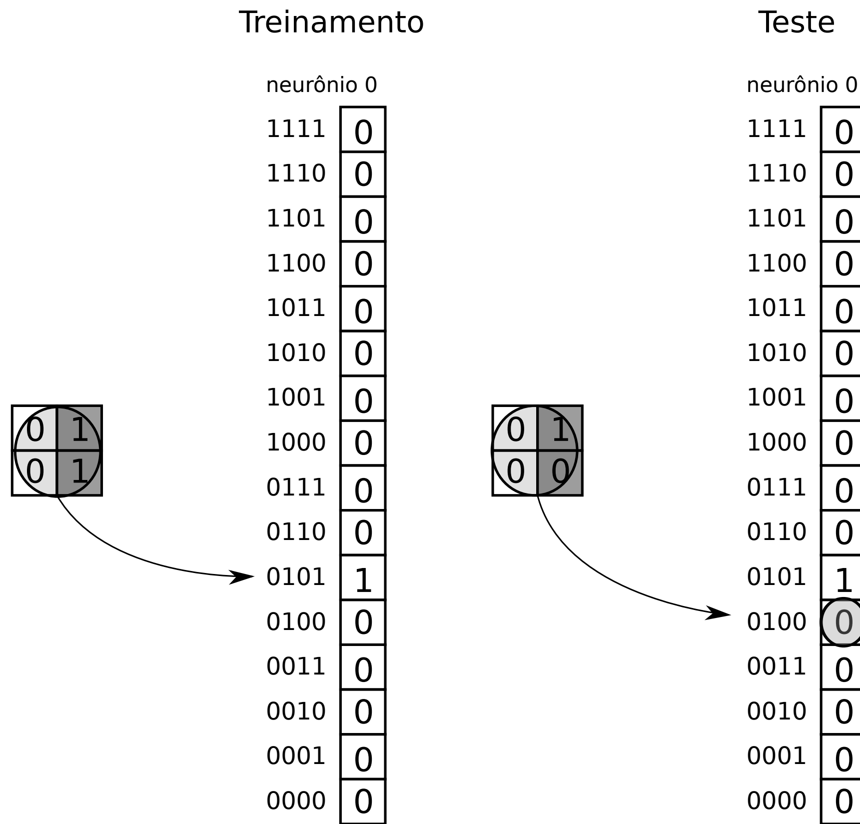

Figura 4.4: Exemplo de treinamento e teste para uma imagem de 4 *pixels* e 1 neurônio por discriminador.

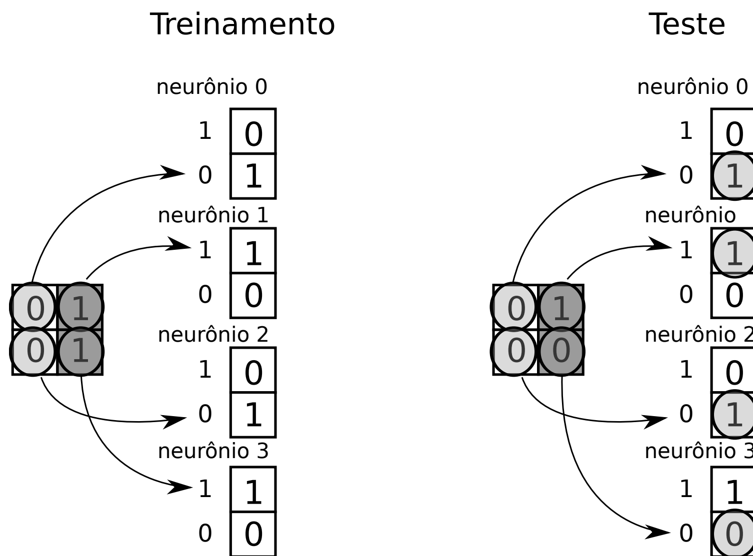

Figura 4.5: Exemplo de treinamento e teste para uma imagem de 4 *pixels* e 4 neurônios por discriminador.

4.3 LeonardoSpectrum - síntese

A Tabela 4.3 mostra o número de instâncias geradas para diversos tipos de otimizações do LeonardoSpectrum, bem como a frequência máxima estimada e a área mínima, calculada apenas somando as áreas das instâncias geradas. Como se pode observar, a frequência máxima de operação do circuito aumenta quando a síntese é feita sem a obrigação de manter a hierarquia (*flatten*), principalmente na otimização por atraso. A área gerada é menor nas sínteses não hierárquicas devido à maior liberdade de otimização do LeonardoSpectrum.

Tabela 4.3: Frequência máxima de operação e área mínima estimadas e número de instâncias geradas para diversas otimizações do LeonardoSpectrum no circuito a ser implementado em ASIC.

Otimização	Frequência máxima (MHz)	Área (μm^2)	Instâncias
por área (mantendo hierarquia)	220	99554	583
por área (sem hierarquia)	319	97807	523
por atraso (mantendo hierarquia)	274	101065	599
por atraso (sem hierarquia)	442	97934	492

Os esquemáticos gerados pelo LeonardoSpectrum com otimização para área de modo hierárquico encontram-se nas Figuras 4.6, 4.7 e 4.8.

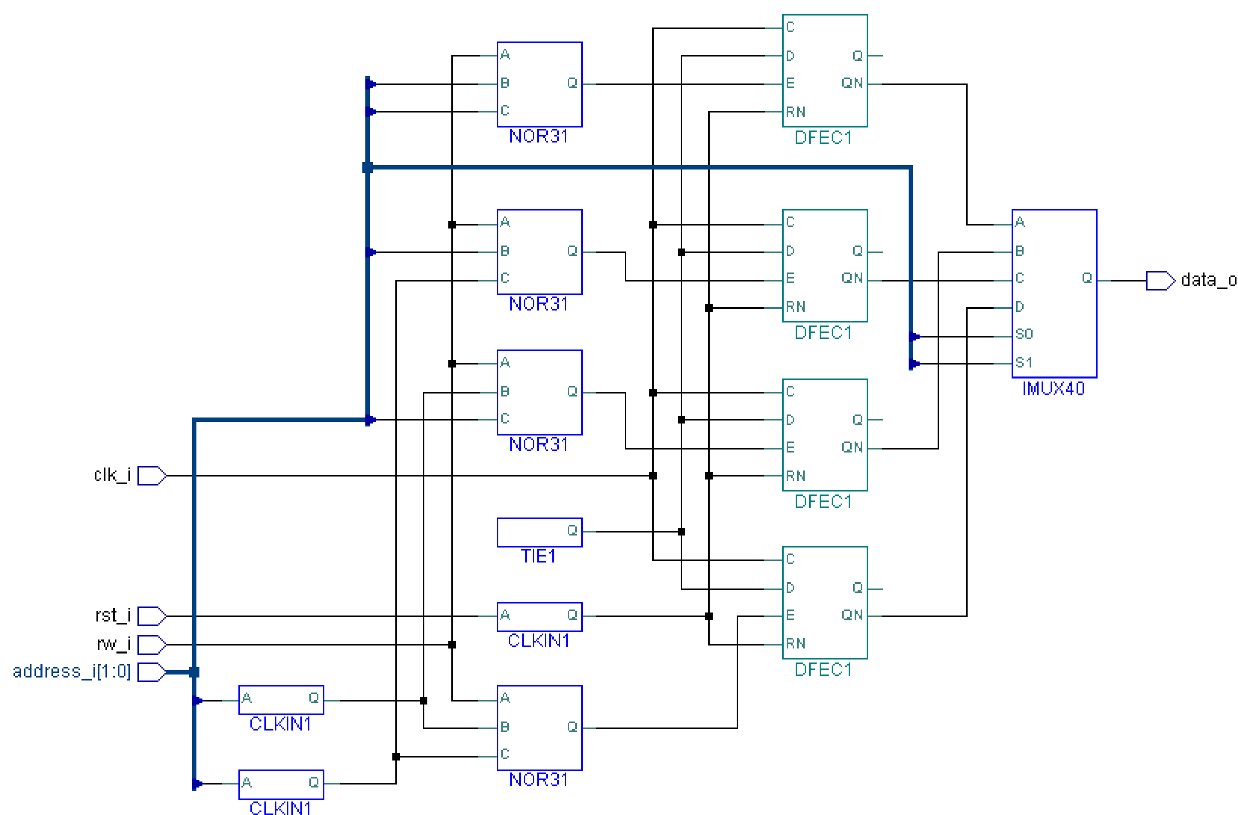

Figura 4.6: Esquemático do neurônio gerado pelo LeonardoSpectrum.

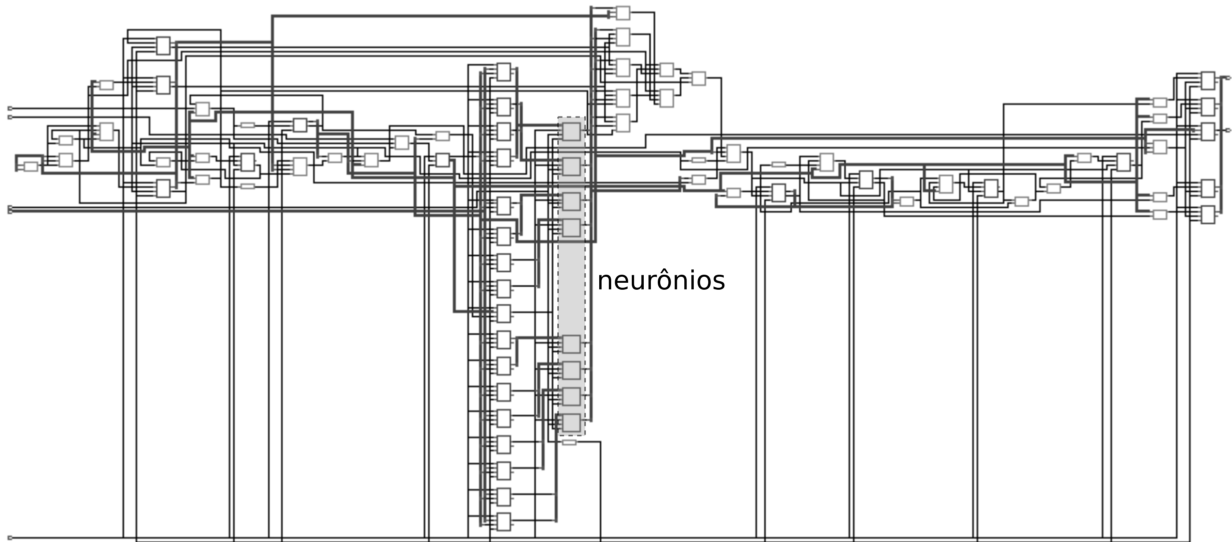

Figura 4.7: Esquemático do discriminador gerado pelo LeonardoSpectrum.

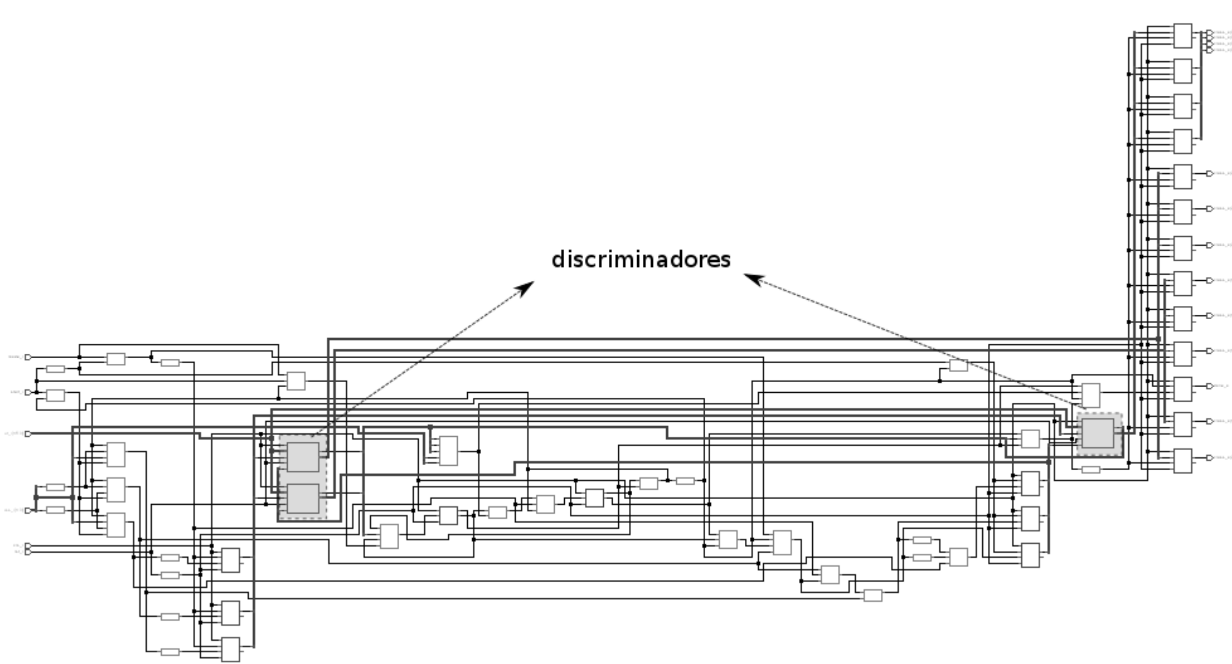

Figura 4.8: Esquemático da rede WISARD gerado pelo LeonardoSpectrum.

4.4 *IC-Station - Layout*

A fim de facilitar o desenho do *layout*, a síntese do circuito para ASIC foi otimizada para área mantendo a hierarquia dos módulos.

A partir da síntese da descrição em VHDL pelo LeonardoSpectrum foi gerado um arquivo em Verilog com as instâncias da tecnologia alvo (AMS CMOS 0,35 μm) e suas conexões. O arquivo foi importado e seu esquemático gerado através das ferramentas da *Mentor Graphics* de forma automatizada. A partir do esquemático foram instanciadas as *standard cells* da tecnologia no *IC Station* para que o desenho do *layout* fosse concluído. A localização (*placement*) das células foi feita de modo automático no nível mais baixo da hierarquia (neurônio) e parcialmente de forma manual nos níveis mais altos devido aos blocos dos neurônios serem maiores que uma célula padrão. O *routing* também foi feito de modo semi-automático, embora as intervenções manuais que eram necessárias provocassem um grande atraso no desenho do *layout*, que devia não apenas estar corretamente conectado conforme o esquemático (*Calibre LVS*), mas conformar-se com as regras de fabricação (*Calibre DRC*). Foi também tomado o cuidado de engrossar as linhas de alimentação (V_{DD} e V_{SS}), o que diminui a resistência das mesmas e gera capacitâncias parasitas capazes de diminuir o ruído causado pelas portas lógicas, estabilizando o circuito e possibilitando maiores frequências de operação. As Figuras 4.9, 4.10 e 4.11 mostram os *layouts* do neurônio, discriminador e *top-level* respectivamente.

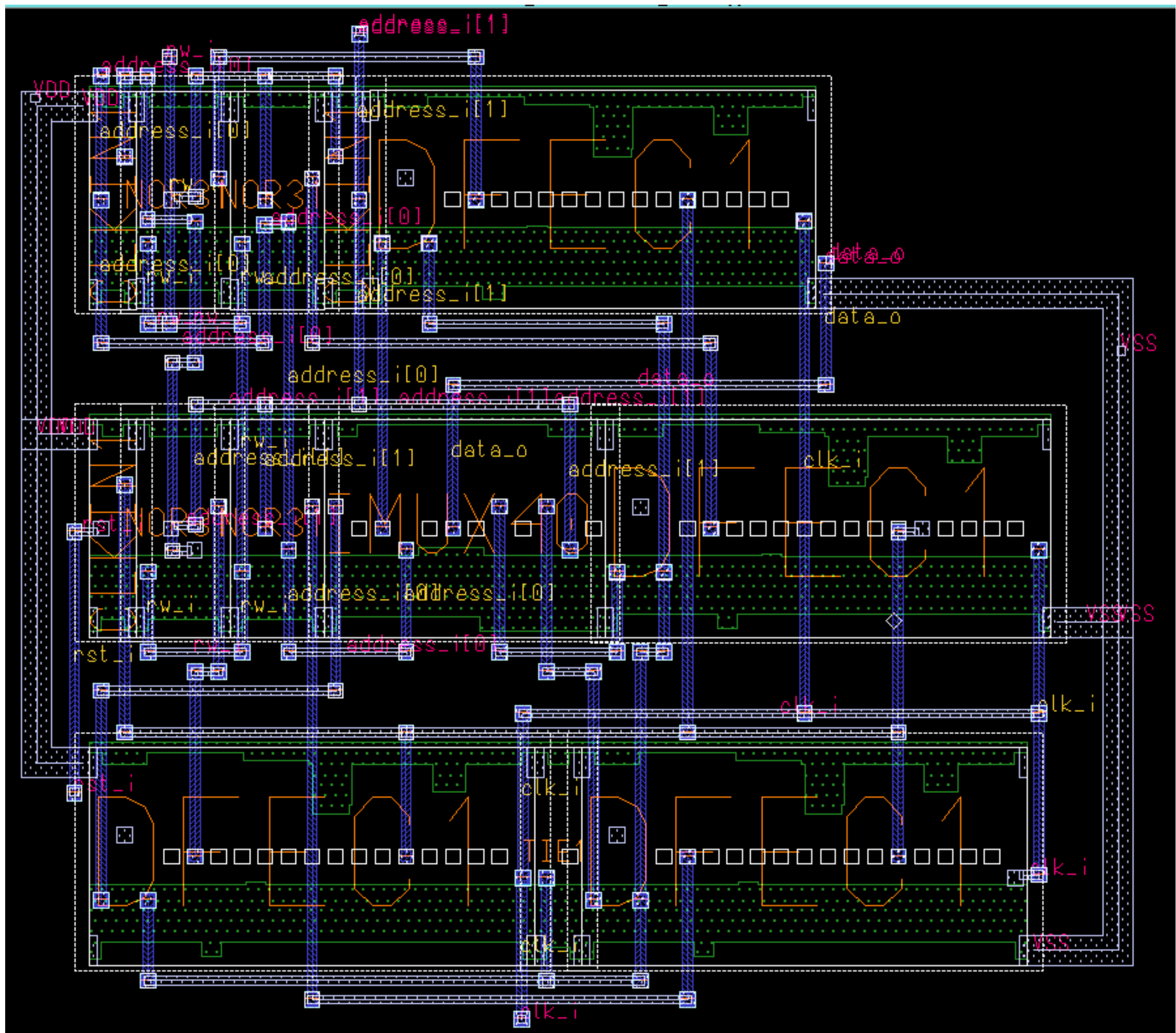

Figura 4.9: *Layout* do neurônio (60 μm x 68 μm).

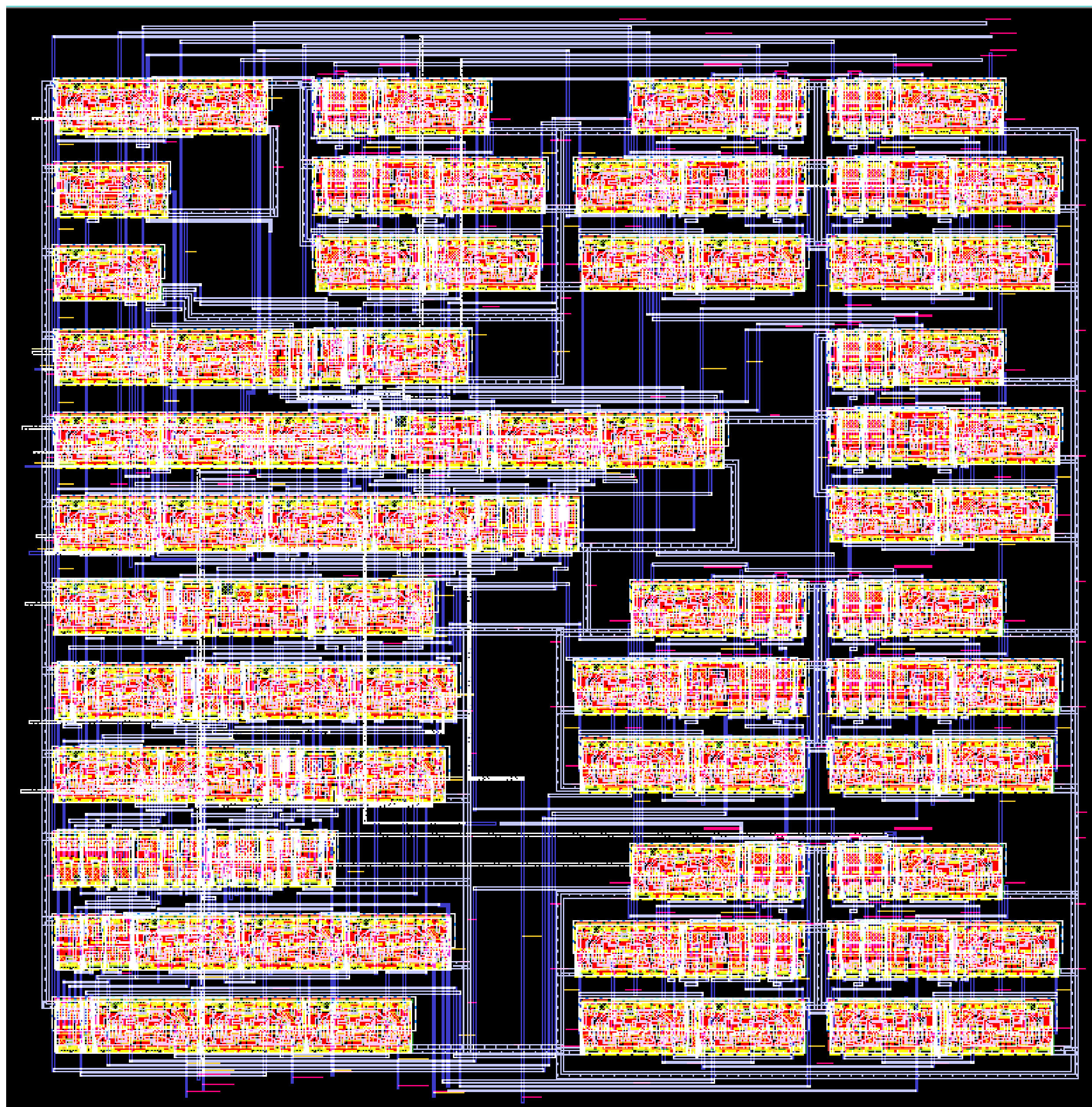

Figura 4.10: *Layout* do discriminador (267 μm x 272 μm).

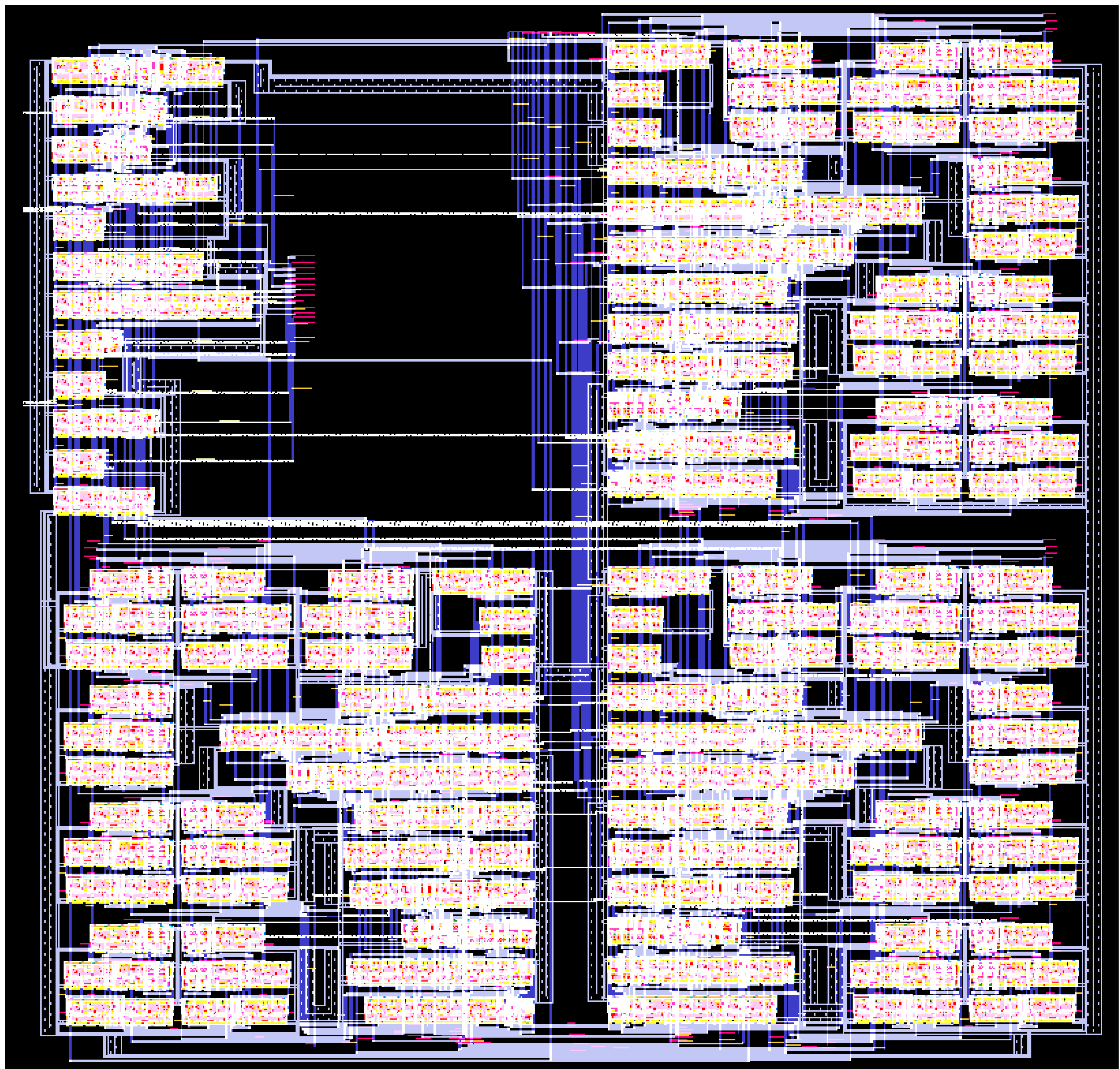

Figura 4.11: *Layout* do *top-level* (565 μm x 582 μm).

A área do *layout* de cada módulo encontra-se na Tabela 4.4. Como a área apresentada no relatório de síntese do LeonardoSpectrum não considera as conexões de metal, apenas soma todas as áreas das instâncias geradas, e a síntese hierárquica tende a aumentar o desperdício de espaço, o *layout* ficou maior, principalmente para o *top-level*.

Tabela 4.4: Áreas mínimas da síntese pelo LeonardoSpectrum (sem considerar as conexões), áreas finais do *layout* implementado para cada módulo e a razão entre as mesmas (*layout*/LeonardoSpectrum).

Módulo	LeonardoSpectrum (μm^2)	*Layout* (μm^2)	Razão
wisard_neuron	2038	4106	2,0
wisard_disc	30139	72865	2,4
wisard_top	99554	329053	3,3

4.5 Eldo - simulação

A partir do *layout* da RNA WISARD com 16 entradas, 3 classes e 8 neurônios por classe (Figura 4.11) extraiu-se um arquivo para realizar a simulação elétrica (Eldo). As simulações foram feitas com os modelos típico, *worst power* e *worst speed* e são similares à executada no ModelSim para 16 entradas (Seção 4.1): 3 classes foram aprendidas e depois testadas.

Os resultados da simulação para o modelo típico encontram-se nas Figuras 4.12, 4.13, 4.14 e 4.15. A correspondência entre os sinais apresentados nessas figuras e os utilizados na descrição em VHDL de *wisard_top* é mostrada na Tabela 4.5.

Tabela 4.5: Correspondência entre os sinais da simulação no Eldo e sua descrição em VHDL.

Eldo	VHDL (*wisard_top*)
V(CLK)	*clk_i*
V(RST)	*rst_i*
V(START)	*start_i*
V(MODE)	*mode_i*
V(DONE)	*done_o*
V(CLASS_O_0_0) V(CLASS_O_0_1) V(CLASS_O_0_2) V(CLASS_O_0_3)	*class_o(0)*
V(CLASS_O_1_0) V(CLASS_O_1_1) V(CLASS_O_1_2) V(CLASS_O_1_3)	*class_o(1)*
V(CLASS_O_2_0) V(CLASS_O_2_1) V(CLASS_O_2_2) V(CLASS_O_2_3)	*class_o(2)*

Os três primeiros pulsos de V(DONE) correspondem ao fim do treinamento das classes 0, 1 e 2, e os três últimos ao término dos testes. O sinal V(MODE) (Figura 4.12) indica o período de treinamento (nível lógico '0') e de teste (nível lógico '1'). Os sinais de saída do discriminador 0 (Figura 4.13) foram 6, 0 e 3; os do discriminador 1 (Figura 4.14) 1, 6 e 2; e os do discriminador 2 (Figura 4.15) 2, 2 e 7 quando as entradas do teste pertenciam às classes 0, 1 e 2, respectivamente.

Comparando-se os resultados com a simulação no ModelSim (Figura 4.1) e com seus resultados (Tabela 4.1), pode-se ver que as saídas foram as esperadas, ou seja, para uma entrada da classe 0,

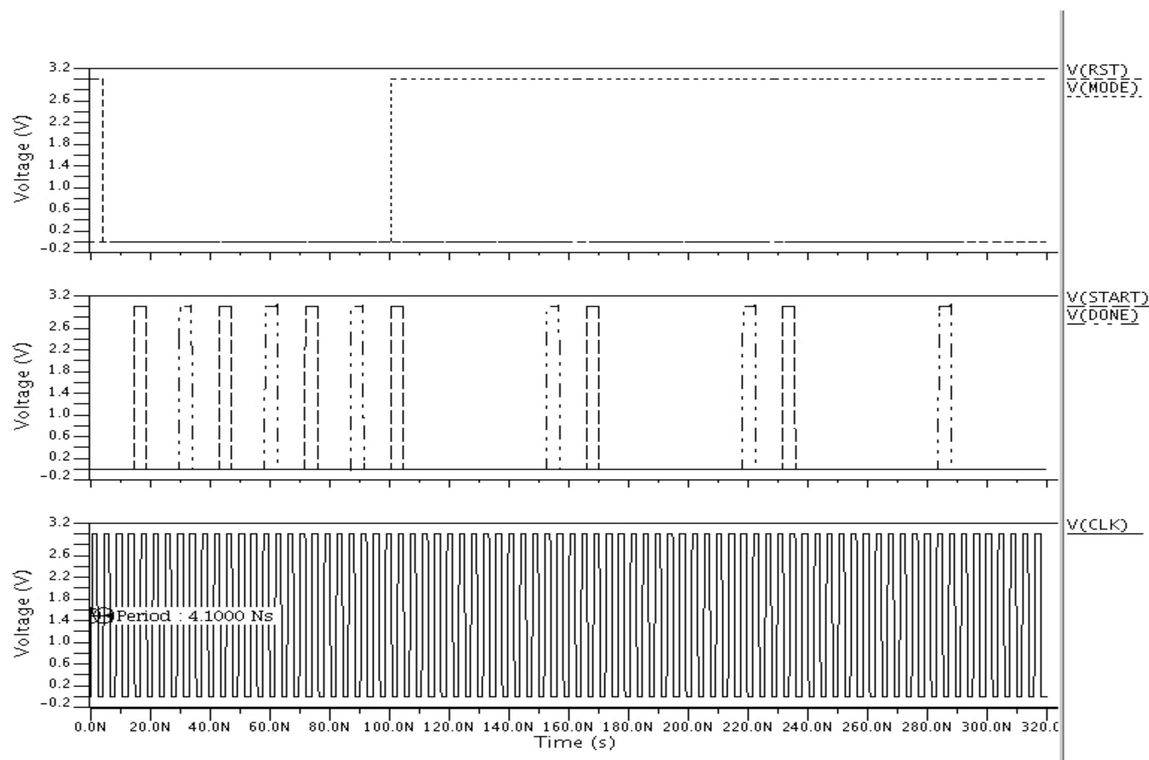

Figura 4.12: Sinais de controle e *clock* para o modelo típico.

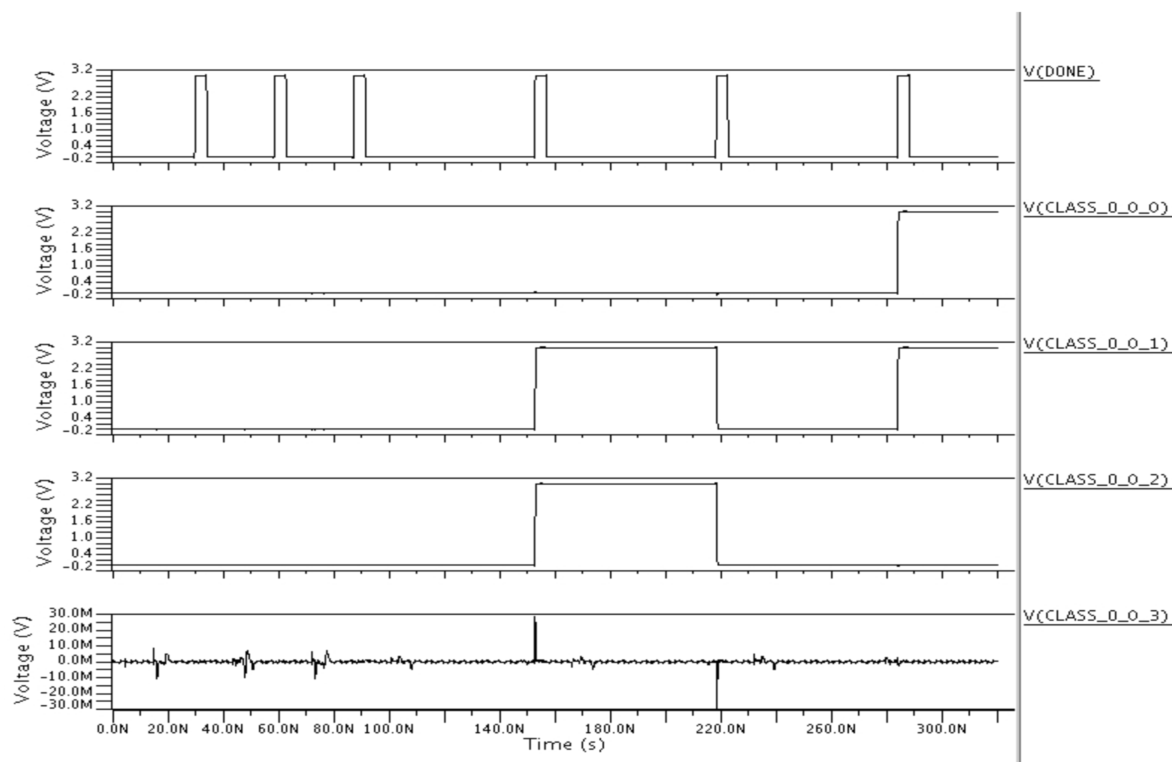

Figura 4.13: Saída do discriminador correspondente à classe 0 para o modelo típico.

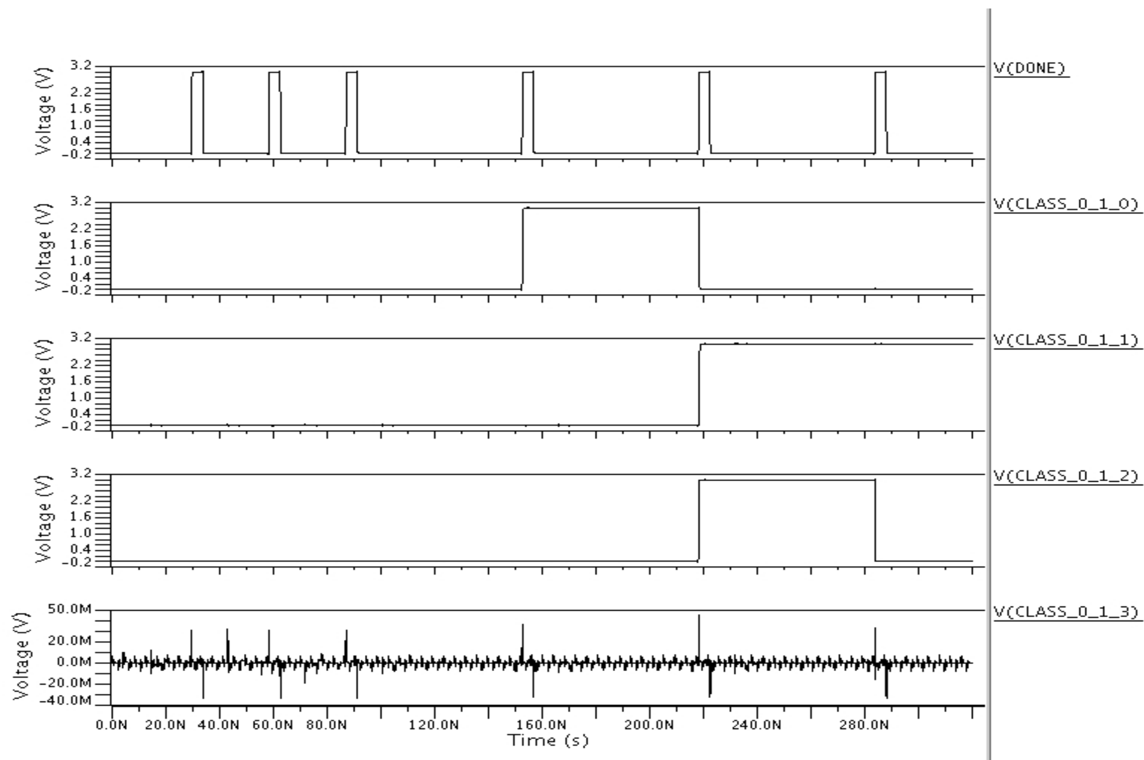

Figura 4.14: Saída do discriminador correspondente à classe 1 para o modelo típico.

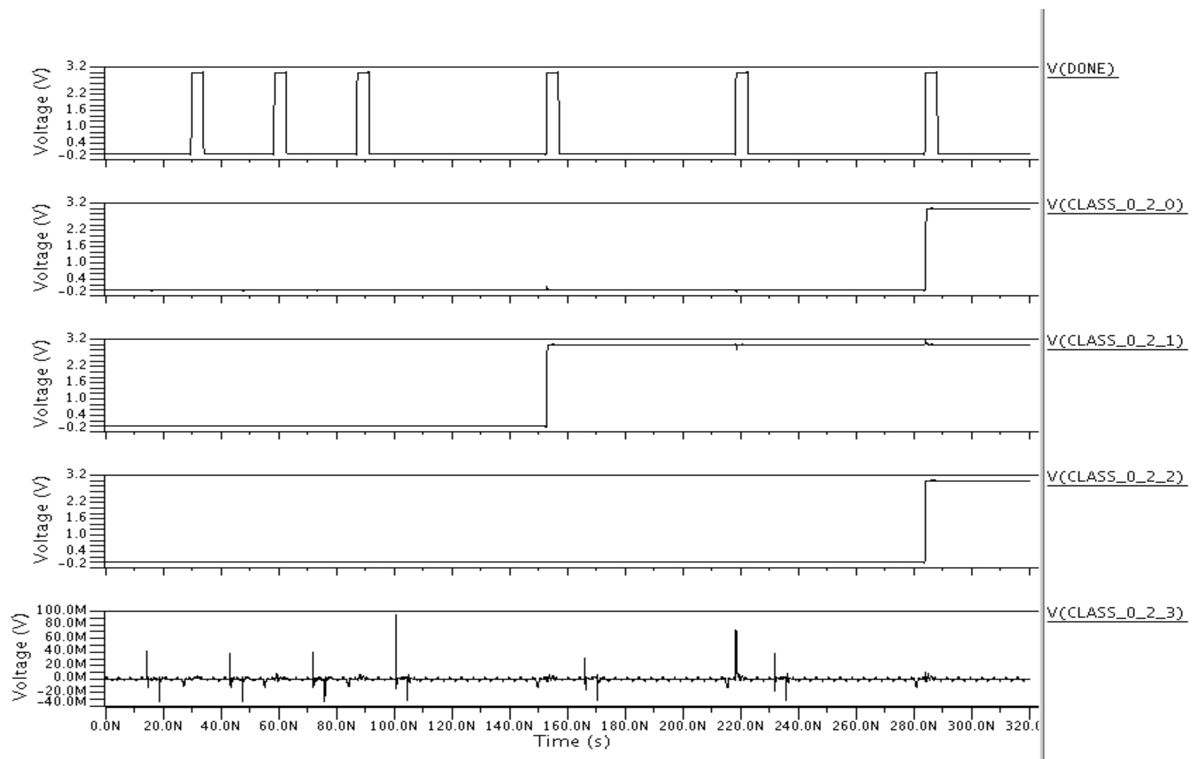

Figura 4.15: Saída do discriminador correspondente à classe 2 para o modelo típico.

seu discriminador correspondente respondeu com 6 (6 neurônios dispararam) e assim por diante.

A frequência máxima de operação e a potência consumida encontradas para cada modelo segundo a simulação no Eldo encontra-se na Tabela 4.6. O resultado para frequência máxima, modelo típico, (240 MHz) foi um pouco maior que a estimativa do LeonardoSpectrum (220 MHz) pois este possivelmente sobre-estimou as capacitâncias parasitas do circuito.

Tabela 4.6: Frequência máxima e consumo de potência do *top-level* implementado na tecnologia AMS 0,35 μm para vários modelos.

Modelo	Frequência máxima (MHz)	Consumo de potência (mW/GHz)
Worst Power	370	86,4
Tipical Model	240	83,6
Worst Speed	170	82,5

4.6 Quartus II - síntese

Foi sintetizada uma RNA WISARD descrita em VHDL com os mesmos parâmetros usados no ASIC (16 entradas, 3 classes e 8 neurônios por discriminador) para um FPGA Cyclone II utilizando-se do *testbench tb_ syn_ wisard_ top* e do bloco ALTPLL da Altera (Figura 3.10). Com as configurações padrão de otimização foram geradas, excetuando-se o *testbench* sintetizável e a PLL, 288 células lógicas, das quais 196 possuíam registradores lógicos dedicados e 92 apenas LUTs (*Look-up Tables*).

A partir de alterações dos parâmetros da PLL da Altera e da utilização do *testbench* sintetizável, pôde-se verificar que a frequência máxima de operação do circuito, semelhante ao implementado em ASIC, exceto pelo nível mais alto da hirarquia (*tb_ syn_ wisard_ top*), foi de 350 MHz. Comparando-se este valor com a frequência máxima encontrada para a simulação do circuito em ASIC (240 MHz no caso típico), temos que a implementação em FPGA foi cerca de 45% mais rápida.

Para que fosse possível a comparação de resultados de frequência máxima de operação entre implementações diferentes (ASIC com *standard cells* e FPGA) em tecnologias diferentes (AMS 0,35 μm e Cyclone II 90 nm), alguns resultados da literatura foram úteis:

- Segundo KUON; ROSE (2007), a relação média entre as frequências máximas de implementações em ASIC com *standard cells* (tecnologia STMicroeletronics 90 nm) e em FPGA (Stratix II - 90 nm) é de 3,2.

- Apesar de o FPGA utilizado (Cyclone II) também ser feito em uma tecnologia 90 nm, analisando diversas implementações (HELIUM TECHNOLOGY, 2008, VISENGI, 2011, JOP, 2008, IPCORES, 2008), o mesmo possui uma frequência máxima de operação entre 1,3 e 1,6 vezes menor que o Stratix II.

- Implementações de um comparador binário em *standard cells* nas tecnologias STMicroeletronics 90 nm e AMS 0,35 μm (PERRI; CORSONELLO, 2008) mostram uma razão de 4,7 entre as suas frequências máximas de operação.

- Ainda, de acordo com (PERRI; CORSONELLO, 2011), que implementa um banco de memória, a razão dos atrasos da memória (tempo de acesso aos dados, *setup* de endereço etc.) entre as implementações em tecnologias AMS 0,35 μm e STMicroeletronics 90 nm varia entre 4,2 e 8,7.

Assim, dado que a frequência máxima do circuito em FPGA é de 350 MHz, a partir das informações apresentadas nos itens anteriores, era de se esperar que a frequência máxima de uma implementação ASIC na tecnologia AMS 0,35 μm estivesse no intervalo de 170 MHz a 430 MHz. Como a frequência máxima obtida da rede WISARD foi de 240 MHz para o modelo típico (podendo variar de 170 MHz a 370 MHz nos demais modelos), os resultados estão dentro do esperado.

4.7 LeonardoSpectrum, R - análises

Com o intuito de estabelecer relações entre a área total das instâncias geradas na implementação para ASIC e os parâmetros da rede neural descrita em VHDL, foram executadas 60 sínteses no LeonardoSpectrum otimizadas para área mantendo hierarquia com diversos valores de número de entradas, número de classes e número de neurônios por discriminador. Esses parâmetros, juntamente com a estimativa de células de um *bit* (descrita na subseção seguinte) foram relacionados com a área e o atraso do caminho crítico obtidos nas sínteses realizadas.

4.7.1 Estimativa de células de um *bit*

Analisando a estrutura da rede WISARD, pode-se concluir que há uma relação entre o número de células de 1 *bit* necessárias e o número de neurônios por discriminador, o número de classes e a quantidade de *pixels*, expressa pela seguinte equação:

$$Estim = N * C * 2^{\frac{I}{N}} \quad (4.1)$$

onde N é o número de neurônios por discriminador, C o número de classes, I o número de *pixels* da imagem e Estim a estimativa do número de células de 1 *bit*.

4.7.2 Correlações

A Tabela 4.7 e a Figura 4.16 mostram, respectivamente, as correlações dos dados extraídos e um gráfico de dispersão em pares. As correlações foram calculadas através do *software* R segundo o método de Pearson e as mais importantes serão discutidas nas próximas subseções.

Tabela 4.7: Correlação entre os diversos parâmetros de entrada da rede WISARD, estimativas de células de um *bit* e dados do relatório apresentados pelo LeonardoSpectrum.

	Área do circuito	**Nº Instâncias**	**Atraso**
Nº Pixels	0,74	0,74	0,53
Neurônios/Discriminador	0,45	0,46	0,70
Tamanho do endereço	0,16	0,16	-0,07
Nº Classes	0,50	0,50	-0,03
Estimativa de células de um *bit*	0,98	0,99	0,41

4.7.3 Atraso do caminho crítico

A correlação de 0,70 entre o atraso do caminho crítico e o número de neurônios por discriminador pode ser explicada pelas seguintes observações:

- Na maioria das sínteses executadas, o caminho crítico encontrava-se dentro do discriminador.
- O discriminador possui um somador que acrescenta no máximo um ao seu valor anterior, ou seja, ele pode ser considerado um contador síncrono com *enable*.
- O discriminador possui um comparador relacionado a esse somador.

Devido às entradas do somador terem sido descritas em VHDL como inteiros de tamanho máximo igual ao número de neurônios por discriminador, a área e o atraso correspondentes a ele crescem linearmente com o teto do logaritmo na base dois desse número. Além disso, o comparador relacionado a esse somador possui um crescimento de área e atraso também logarítmicos. Um outro modo de descrever o somador seria com um *shift register*. Nesse caso o caminho crítico quase não aumentaria, porém a área cresceria linearmente com o número de neurônios por discriminador. Por outro lado, o comparador seria reduzido ao último *bit* do *shift register*.

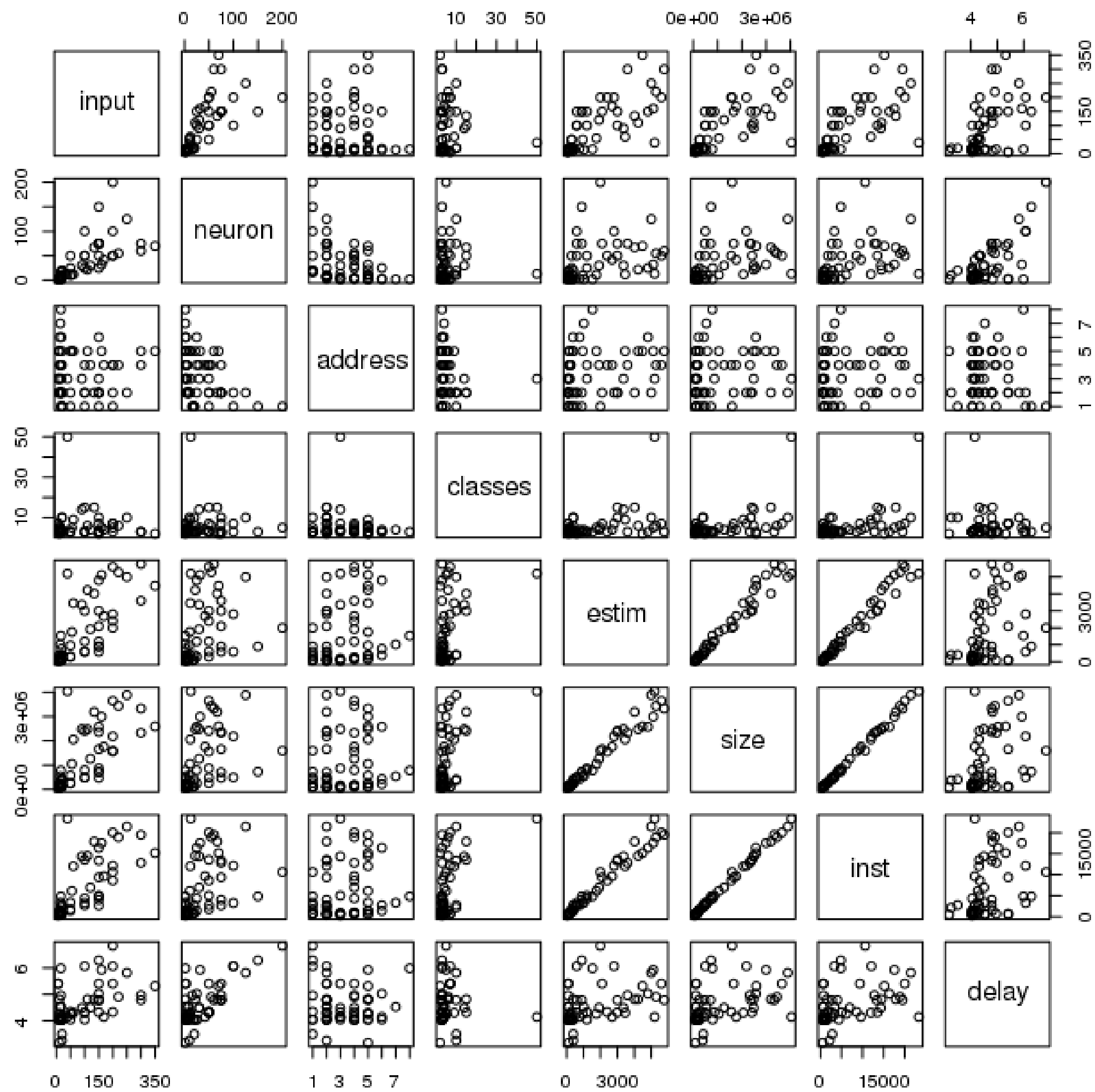

Figura 4.16: Gráfico de dispersão em pares, onde *input* é a quantidade de *pixels*, *neuron* o número de neurônios, *address* o tamanho do endereço, *classes* o número de classes, *estim* o tamanho estimado de acordo com a equação 4.1, *size* o tamanho apresentado pelo LeonardoSpectrum, *inst* o número de instâncias e *delay* o atraso do caminho crítico.

4.7.4 Estimativa de área

A alta correlação entre a estimativa de células de 1 *bit* e a área mínima apresentada pelo LeonardoSpectrum (0,98), bem como o gráfico apresentado na Figura 4.16, indicam uma relação linear entre as mesmas. Portanto, foi utilizada a função de regressão linear disponível no R (*lm()*), chegando-se à seguinte fórmula de cálculo da área:

$$Estim = N * C * 2^{\frac{I}{N}} * 650.2 + 51668.7 \tag{4.2}$$

onde N é o número de neurônios por discriminador, C o número de classes, I o número de *pixels* da imagem e Estim o tamanho mínimo do circuito, excetuando-se as conexões, em μm^2. O ajuste entre a estivativa de número de células de *bit* e a área mínima dada pelo LeonardoSpectrum foi alto, isto é, o valor de R^2-ajustado (Apêndice C) foi de 0,97.

Entretanto, existem alguns pressupostos para que a regressão linear possa ser utilizada:

1. A relação entre a covariável *número de células de um bit* e a variável de resposta *área do circuito* deve ser linear.
2. O *número de células de um bit* não é uma variável aleatória.
3. O *número de células de um bit* possui uma variância não nula.
4. A covariância entre o erro e o *número de células de um bit* é nula.
5. A variância do erro é constante (homogênea).
6. Os erros das variáveis de resposta são independentes.
7. O erro possui uma distribuição normal.

Apesar do ajuste da reta ter sido bom, a variância do erro não é constante, ou seja, ele aumenta com o aumento do tamanho do circuito, conforme pode ser visto na Figura 4.17. Caso a variância fosse constante, o gráfico de resíduos por valores preditos deveria ter uma distribuição homogênea de pontos, não a forma triangular encontrada. Portanto, a Equação 4.2 deve ser usada com cautela para um circuito maior.

4.7.5 Previsões de área

Com a Equação 4.2, foram feitos gráficos de previsões do tamanho do circuito para diversos números de *pixels* e neurônios por discriminador com o intuito de se observar o efeito dos mesmos na área final. Assim, foram obtidos os gráficos das Figuras 4.18 e 4.19. Conforme esperado, a área aumenta exponencialmente com a diminuição do número de neurônios. Além disso, para um número fixo de neurônios por discriminador, a área também aumenta exponencialmente com o aumento do número de *pixels* de entrada.

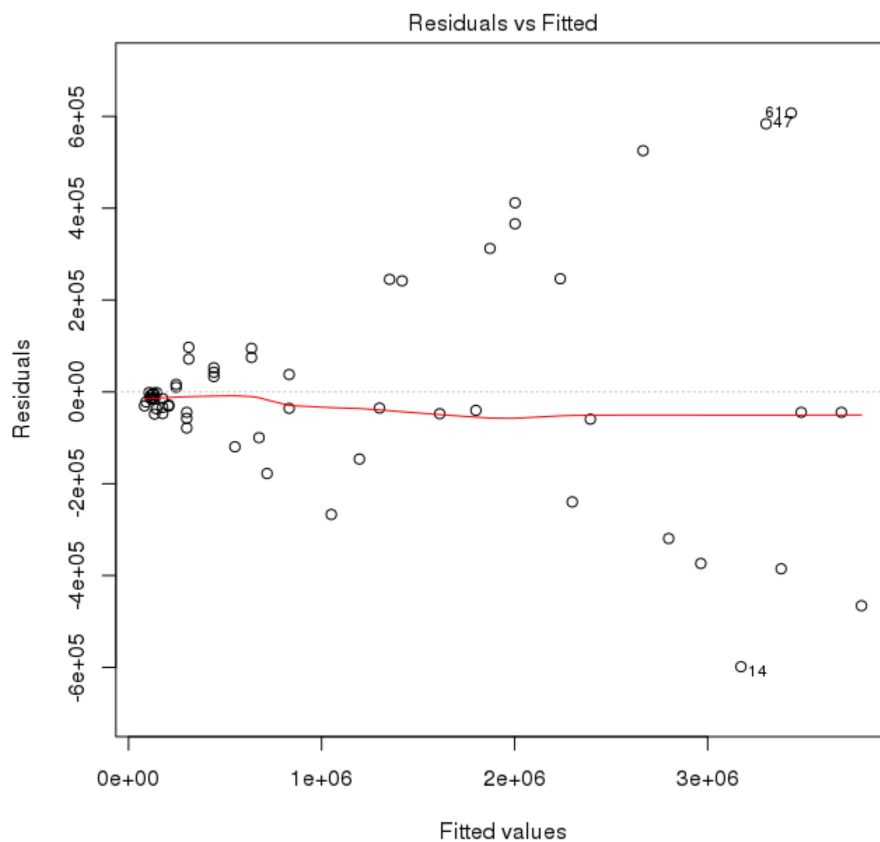

Figura 4.17: Resíduos *versus* valores preditos.

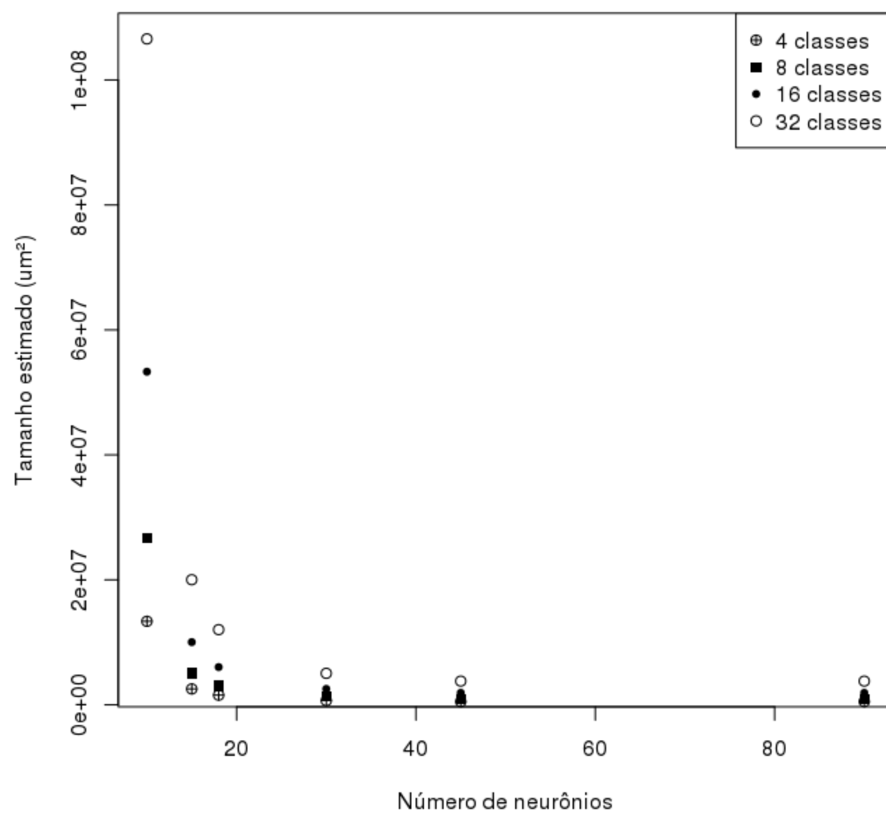

Figura 4.18: Tamanho estimado (μm^2) *versus* número de neurônios por discriminador para um número variável de classes com uma imagem de 90 *pixels*.

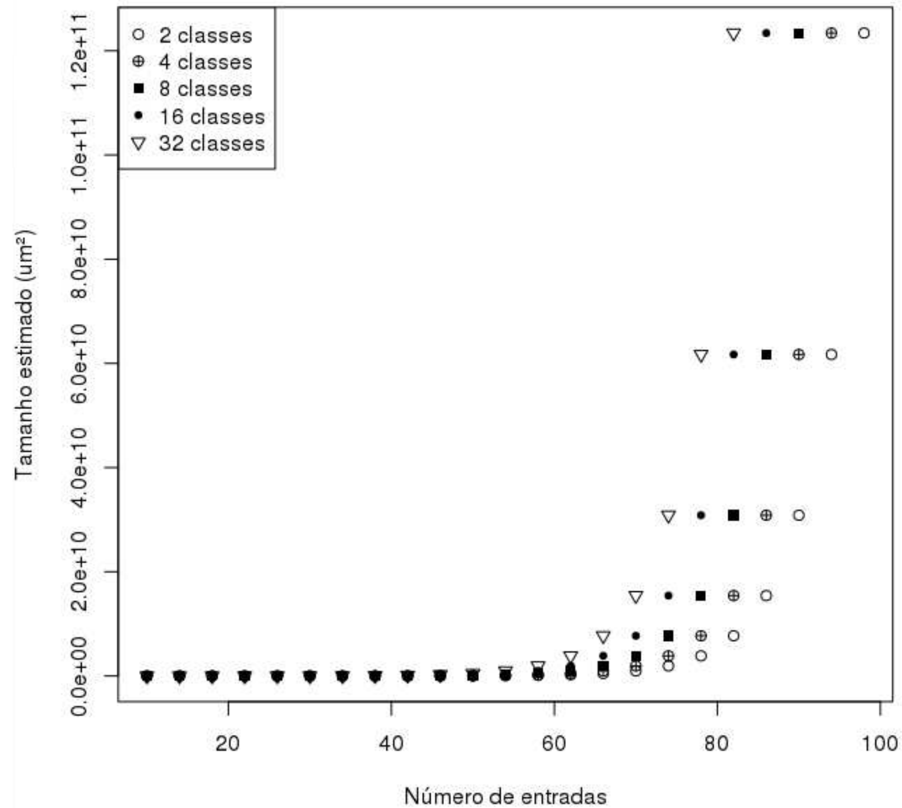

Figura 4.19: Tamanho estimado (μm^2) *versus* número de *pixels* para um número variável de classes com 4 neurônios por discriminador.

A fim de se verificar a validade da estimativa em relação às áreas apresentadas pelo LeonardoSpectrum, foi gerado um gráfico com esses valores para os casos testados (Figura 4.20).

Para um número maior de *pixels* na entrada não foi possível obter os dados do LeonardoSpectrum para uma quantidade baixa de neurônios pois, neste caso, devido ao tamanho do circuito aumentar exponencialmente, o programa deixou de responder.

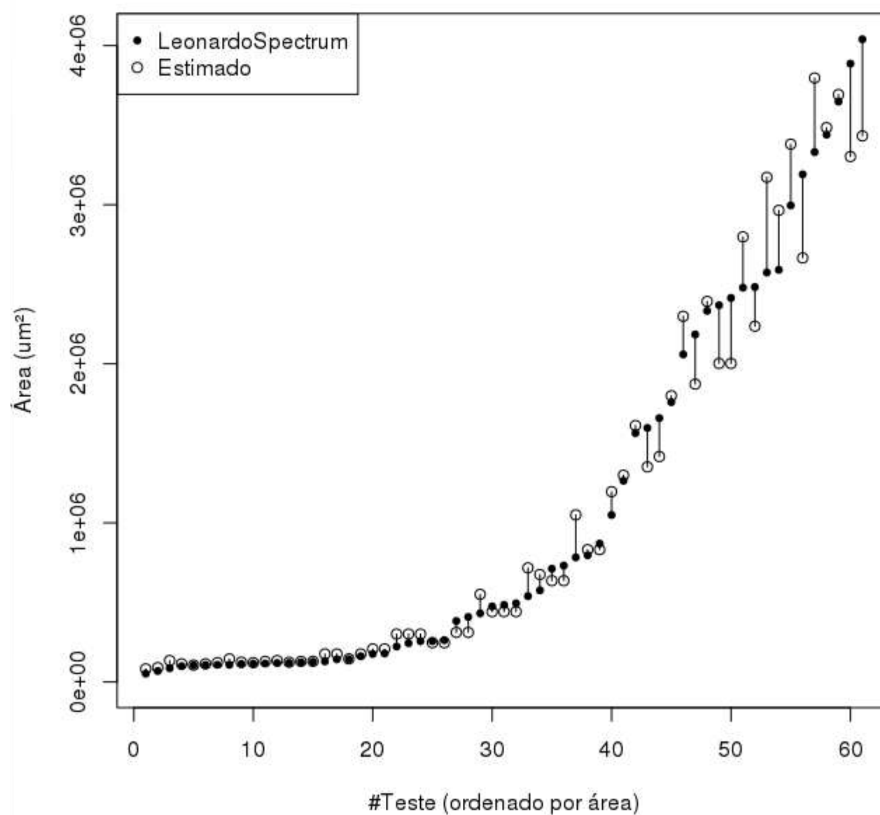

Figura 4.20: Áreas estimadas segundo a Equação 4.2 e apresentadas pelo LeonardoSpectrum (μm^2) para os testes realizados.

Capítulo 5

Conclusões

Nesse trabalho foi descrita uma rede neural WISARD em VHDL e simulada com o *software* ModelSim. Um exemplo dessa descrição, uma rede com 16 entradas, 3 classes e 8 neurônios por discriminador, foi sintetizado para ASIC. Seu *layout* foi desenvolvido e o circuito extraído a partir do mesmo foi simulado no Eldo. A descrição em VHDL da rede com os mesmos parâmetros foi também validada em FPGA. Além disso, foi implementado um gerador de *testbenches* em VHDL e um gerador de arquivos SPICE para esse problema específico utilizando-se da linguagem Python.

As velocidades máximas de operação obtidas para a rede WISARD sintetizada com 16 entradas, 3 classes e 8 neurônios por discriminador foram de 350 MHz para o FPGA Cyclone II, da Altera, e de 240 MHz para a tecnologia AMS 0,35 μm, modelo típico. Conforme discutido na Seção 4.6, esses resultados estão coerentes de acordo com a análise de outras implementações encontradas na literatura.

A área mínima do *layout* apresentada pelo LeonardoSpectrum foi cerca de 3,3 vezes menor que a área final do circuito (329053 μm^2) na tecnologia AMS 0,35 μm. Na síntese para o Cyclone II, foram utilizadas 288 células lógicas (196 com registradores lógicos dedicados e 92 com apenas LUTs).

Foi criada uma equação a fim de estimar a área mínima do circuito implementado em ASIC (AMS 0,35 μm) que é calculada a partir da soma das áreas das instâncias sintetizadas pelo LeonardoSpectrum. A estimativa de área fornecida pela equação apresentou uma correlação de 98% com os dados reais obtidos.

Esse projeto permitiu o aprendizado sobre redes neurais, que não haviam sido estudadas durante a graduação, o aprimoramento nas habilidades de descrição de *hardware* em VHDL direcionado à síntese e a utilização de conceitos de estatística para gerar estimativas de área. Além disso, esse trabalho compreendeu desde o uso de uma linguagem de alto nível, como Python, até o desenvolvimento de esquemáticos e *layouts* de circuitos, envolvendo aplicação de diversos conceitos aprendidos durante a graduação em um mesmo projeto.

Como sugestões para trabalhos futuros estão: otimizações do código em VHDL com o intuito de se reduzir a área ou aumentar a velocidade máxima de operação; mudanças em cada discriminador a fim de otimizar a rede para a resolução de algum problema específico; implementações dessa rede em outras tecnologias ASIC a fim de se comparar área e atraso do caminho crítico; e comparação da rede WISARD com outras redes neurais a serem implementadas em *hardware*.

Bibliografia

AIZENBERG, I. N. **Complex-Valued Neural Networks with Multi-Valued Neurons**. Springer, 2011.

ALTERA. **Introduction to the Quartus II Software**. 2010.

ALTERS, S.; ALTERS, B. **Biology: Understanding Life**. Jones & Bartlett Publishers, 1999.

AUSTRIAMICROSYSTEMS. **0.35 μm CMOS C35 Design Rules**. 2003.

AUSTRIAMICROSYSTEMS. **0.35 μm CMOS Technology Selection Guide**, 2012. http://www.ams.com/eng/Products/Full-Service-Foundry/Process-Technology/CMOS/0.35-m-CMOS-Technology-Selection-Guide>. Acesso em 12/06/2012.

AZHAR, M. H. B.; DIMOND, K. Design of an FPGA based adaptive neural controller for intelligent robot navigation. In: EUROMICRO SYMPOSIUM ON DIGITAL SYSTEM DESIGN, 2002. **Proceedings...** p. 283–290, 2002.

BLEDSOE, W. W.; BROWNING, I. Pattern recognition and reading by machine. In: EASTERN JOINT IRE-AIEE-ACM COMPUTER CONFERENCE, 1959, Boston, Massachusetts. **Proceedings...** Boston, Massachusetts, p. 225–232, 1959.

BRAGA, A. P., LUDERMIR, T. B.; CARVALHO, A. C. P. d. L. F. **Redes Neurais Artificiais: Teoria e Aplicações**. LTC, 2000.

BREWER, J. A new and improved roadmap. **Circuits and Devices Magazine, IEEE**, v. 14, n. 2, p. 13–18, 1998.

CECOTTI, H.; GRASER, A. Convolutional Neural Networks for P300 Detection with Application to Brain-Computer Interfaces. **Pattern Analysis and Machine Intelligence, IEEE Transactions on**, v. 33, n. 3, p. 433–445, 2011.

ELIZONDO, D. The Linear Separability Problem: Some Testing Methods. **IEEE Transactions on Neural Networks**, v. 17, n. 2, p. 330–344, 2006.

GODSE, A.; GODSE, D. **Fundamentals of HDL**. Technical Publications, 2009.

HELIUM TECHNOLOGY. **DVB LSA Altera Core Datasheet**. 2008.

IPCORES. **Elliptic Curve Point Multiply and Verify Core**, 2008. Disponível em: <http://www.ipcores.com/elliptic_curve_crypto_ip_core.htm>. Acesso em 30/05/2012.

JOP. **Jop Maximum Frequency**, 2008. Disponível em: <http://www.jopwiki.com/Maximum_frequency>. Acesso em 30/05/2012.

KRIESEL, D. **A Brief Introduction to Neural Networks**, 2005.

KROSE, B.; SMAGT, P. van der. **An introduction to Neural Networks**, 1996.

KUON, I.; ROSE, J. Measuring the gap between FPGAs and ASICs. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 26, n. 2, p. 203–215, 2007.

LINES, J. A. *et al.* An automatic image-based system for estimating the mass of free-swimming fish. **Computers and Eletronics in Agriculture**, v. 31, n. 2, p. 151–168, 2001.

MCCULLOCH, W. S.; PITTS, W. H. A logical calculus of the ideas immanent in nervous activity. **Bulletin of Mathematical Biology**, v. 5, n. 4, p. 115–133, 1943.

MENTOR GRAPHICS. **LeonardoSpectrum User's Guide**. 1999.

MENTOR GRAPHICS. **Mentor Graphics Design Architect-IC User's Manual**. 2003.

MENTOR GRAPHICS. **Calibre LVS Datasheet**. 2005.

MENTOR GRAPHICS. **Mentor Graphics IC Station Datasheet**. 2009.

MENTOR GRAPHICS. **Eldo Classic Datasheet**. 2011.

MENTOR GRAPHICS. **EZwave Datasheet**. 2011.

MENTOR GRAPHICS. **ModelSim PE Student Edition - HDL Simulation**, 2012. Disponível em: <http://model.com/content/modelsim-pe-student-edition-hdl-simulation>. Acesso em 15/05/2012.

OLIVEIRA, M. A. d. **Aplicação de redes neurais artificiais na análise de séries temporais econômico-financeiras**. Tese (Doutorado), FEA-USP, 2007.

PATTICHIS, C. S. *et al.* A hybrid neural network electromyographic system: incorporating the WISARD net. In: IEEE INTERNATIONAL CONFERENCE ON NEURAL NETWORKS. IEEE WORLD CONGRESS ON COMPUTATIONAL INTELLIGENCE, 1994. **Proceedings...** New York, NY, USA, p. 3478–83, 1994.

PERRI, S.; CORSONELLO, P. Fast Low-Cost Implementation of Single-Clock-Cycle Binary Comparator. **IEEE Transactions on Circuits and Systems II: Express Briefs**, v. 55, n. 12, p. 1239–1243, 2008.

PERRI, S.; CORSONELLO, P. Efficient memory architecture for image processing. **International Journal of Circuit Theory and Applications**, Chichester, UK, v. 39, n. 3, p. 351–356, 2011.

R-PROJECT. **What is R?**, 2012. Disponível em: <http://www.r-project.org/>. Acesso em 29/05/2012.

RUPPERT, D. **Statistics and data analysis for financial engineering**. Springer, 2011.

SIA. **Semiconductor Capacity Utilization (SICAS) Reports**, 2012. Disponível em: <http://www.sia-online.org/industry-statistics/semiconductor-capacity-utilization-sicas-reports/>. Acesso em 02/06/2012.

SOUZA, C. R. **Redes neurais sem peso aplicadas na categorização de subtipos do HIV-1**. Dissertação (Mestrado), COPPE-UFRJ, 2011.

TERASIC. **Altera Cyclone II Starter Development Kit**, 2012. Disponível em: <http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=56&No=121>. Acesso em 14/06/2012.

VISENGI. **JPEG Decoder**, 2011. Disponível em: <http://www.visengi.com/products/jpeg_hardware_decoder>. Acesso em 30/05/2012.

WILLIAMS, P.; YORK, T. Hardware implementation of RAM-based neural networks for tomographic data processing. **Computers and Digital Techniques, IEE Proceedings**, v. 146, n. 2, p. 114–118, 1999.

APÊNDICE A – Código Fonte em Python

Gerador de *testbenches* - tb_gen.py

```
1 import struct
2 import sys
3
4 def read_file(filename):
5     x = []
6     f = open(filename, "rb")
7     i = 1
8     try:
9         byte = f.read(54)
10        byte = f.read(1)
11        while byte != "":
12
13            if i == 10:
14                byte = f.read(1)
15                i = 0
16            else:
17                a = (struct.unpack('B', byte)[0])
18                byte = f.read(1)
19                b = (struct.unpack('B', byte)[0])
20                byte = f.read(1)
21                c = (struct.unpack('B', byte)[0])
22                if a>128:
23                    x.append(0)
24                else:
25                    x.append(1)
26                i += 1
27            byte = f.read(1)
28    finally:
29        f.close()
30    return x
31
32 def print_bin(x):
33     sys.stdout.write("\"")
34     for i in range(0,len(x)):
35         sys.stdout.write(str(x[i]))
36     sys.stdout.write("\"")
37
38 def print_setup():
39     print "input_s <= (others=>'0');"
40     print "start_s <= '0';"
41     print "mode_s <= '0';"
42     print "wait for RST_TIME;"
43     print "wait until rising_edge(clk_s);"
44
45 def print_input(input_s):
46     print "input_s <= ",
47     print_bin(input_s)
48     print ";"
```

```
49
50 def print_class(class_s):
51     print "class_i_s <= ",
52     sys.stdout.write(str(class_s))
53     print ";"
54
55 def print_start_pulse():
56     print "start_s <= '1';"
57     print "wait until rising_edge(clk_s);"
58     print "start_s <= '0';"
59
60 def print_wait_for_done():
61     print "wait until done_s = '1';"
62     print "wait until rising_edge(clk_s);"
63
64 def train(filename, class_s):
65     file_read = read_file(filename)
66     print_input(file_read)
67     print_class(class_s)
68     print_start_pulse()
69     print_wait_for_done()
70     print
71
72 def print_test_setup():
73     print "mode_s <= '1';"
74
75 def test(filename):
76     print_input(read_file(filename))
77     print_start_pulse()
78     print_wait_for_done()
79
80
81 if __name__ == "__main__":
82     print_setup()
83     train("images/0_1.bmp",0)
84     train("images/0_2.bmp",0)
85     train("images/0_3.bmp",0)
86     train("images/1_1.bmp",1)
87     train("images/1_2.bmp",1)
88     train("images/1_3.bmp",1)
89     train("images/2_1.bmp",2)
90     train("images/2_2.bmp",2)
91     train("images/2_3.bmp",2)
92     print_test_setup()
93     test("images/0_4.bmp")
94     test("images/0_5.bmp")
95     test("images/0_6.bmp")
96     test("images/1_4.bmp")
97     test("images/1_5.bmp")
98     test("images/1_6.bmp")
99     test("images/2_4.bmp")
100    test("images/2_5.bmp")
101    test("images/2_6.bmp")
```

Gerador de arquivos SPICE - wave_gen.py

```
1 import sys
2 from sys import stdout
3
4 def gen_wave(name,times,value, desl=True):
5   print name,
6   sys.stdout.write('PWL(')
7   print times[0], value[0], '',
```

```
8
9     if desl:
10      for i in xrange(1, len(times)):
11        print '',
12        sys.stdout.write('\'')
13        sys.stdout.write(str(times[i]))
14        print '*T1+DESL-INF\'', value[i-1], '',
15        sys.stdout.write('\'')
16        sys.stdout.write(str(times[i]))
17        print '*T1+DESL\'', value[i],
18    else:
19      for i in xrange(1, len(times)):
20        print '',
21        sys.stdout.write('\'')
22        sys.stdout.write(str(times[i]))
23        print '*T1-INF\'', value[i-1], '',
24        sys.stdout.write('\'')
25        sys.stdout.write(str(times[i]))
26        print '*T1\'', value[i],
27
28    sys.stdout.write(')')
29
30  if __name__ == "__main__":
31    VDD = 3
32    START1 = 3
33    START2 = 10
34    START3 = 17
35    START4 = 24
36    START5 = 40
37    START6 = 56
38
39    print '\n'
40    times = [0, 1]
41    value = [VDD, 0]
42    gen_wave('V_rst RST 0 ', times, value, False)
43    print '\n'
44    times = [0, START4]
45    value = [0, VDD]
46    gen_wave('V_mode MODE 0 ', times, value)
47    print '\n'
48    times = [0, START1, START1+1, START2, START2+1, START3, START3+1, START4,
        START4+1, START5, START5+1, START6, START6+1]
49    value = [0,     VDD,       0,     VDD,       0,     VDD,       0,     VDD,
              0,     VDD,       0,     VDD,       0]
50    gen_wave('V_start START 0 ', times, value)
51    print '\n'
52    times = [0, START1, START2, START3, START4, START5, START6]
53    value = [0,     VDD,      0,    VDD,      0,      0,    VDD]
54    gen_wave('V_i00 INPUT_0 0 ', times, value)
55    print '\n'
56    times = [0, START1, START2, START3, START4, START5, START6]
57    value = [0,     VDD,      0,    VDD,    VDD,    VDD,    VDD]
58    gen_wave('V_i01 INPUT_1 0 ', times, value)
59    print '\n'
60    times = [0, START1, START2, START3, START4, START5, START6]
61    value = [0,     VDD,    VDD,    VDD,    VDD,      0,    VDD]
62    gen_wave('V_i02 INPUT_2 0 ', times, value)
63    print '\n'
64    times = [0, START1, START2, START3, START4, START5, START6]
65    value = [0,     VDD,      0,    VDD,    VDD,      0,    VDD]
66    gen_wave('V_i03 INPUT_3 0 ', times, value)
67    print '\n'
68    times = [0, START1, START2, START3, START4, START5, START6]
69    value = [0,     VDD,      0,      0,    VDD,      0,      0]
70    gen_wave('V_i04 INPUT_4 0 ', times, value)
```

```
71     print '\n'
72     times = [0, START1, START2, START3, START4, START5, START6]
73     value = [0,      0,      0,      0,      0,      0,      0]
74     gen_wave('V_i05 INPUT_5 0 ', times, value)
75     print '\n'
76     times = [0, START1, START2, START3, START4, START5, START6]
77     value = [0,      0,    VDD,    VDD,      0,    VDD,    VDD]
78     gen_wave('V_i06 INPUT_6 0 ', times, value)
79     print '\n'
80     times = [0, START1, START2, START3, START4, START5, START6]
81     value = [0,    VDD,      0,      0,    VDD,      0,      0]
82     gen_wave('V_i07 INPUT_7 0 ', times, value)
83     print '\n'
84     times = [0, START1, START2, START3, START4, START5, START6]
85     value = [0,    VDD,      0,      0,    VDD,      0,      0]
86     gen_wave('V_i08 INPUT_8 0 ', times, value)
87     print '\n'
88     times = [0, START1, START2, START3, START4, START5, START6]
89     value = [0,      0,      0,    VDD,      0,      0,    VDD]
90     gen_wave('V_i09 INPUT_9 0 ', times, value)
91     print '\n'
92     times = [0, START1, START2, START3, START4, START5, START6]
93     value = [0,      0,    VDD,      0,      0,    VDD,      0]
94     gen_wave('V_i10 INPUT_10 0 ', times, value)
95     print '\n'
96     times = [0, START1, START2, START3, START4, START5, START6]
97     value = [0,    VDD,      0,      0,    VDD,      0,      0]
98     gen_wave('V_i11 INPUT_11 0 ', times, value)
99     print '\n'
100    times = [0, START1, START2, START3, START4, START5, START6]
101    value = [0,    VDD,      0,    VDD,    VDD,      0,      0]
102    gen_wave('V_i12 INPUT_12 0 ', times, value)
103    print '\n'
104    times = [0, START1, START2, START3, START4, START5, START6]
105    value = [0,    VDD,      0,    VDD,    VDD,      0,    VDD]
106    gen_wave('V_i13 INPUT_13 0 ', times, value)
107    print '\n'
108    times = [0, START1, START2, START3, START4, START5, START6]
109    value = [0,    VDD,    VDD,    VDD,    VDD,    VDD,    VDD]
110    gen_wave('V_i14 INPUT_14 0 ', times, value)
111    print '\n'
112    times = [0, START1, START2, START3, START4, START5, START6]
113    value = [0,    VDD,      0,    VDD,      0,      0,    VDD]
114    gen_wave('V_i15 INPUT_15 0 ', times, value)
115    print '\n'
```

APÊNDICE B – Código Fonte em VHDL

Pacote wisard_pkg

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 package wisard_pkg is
 6
 7     constant CLASS_NUMBER     : integer := 3;
 8
 9     constant INPUTS_NUMBER    : integer := 16;
10     constant NEURONS_NUMBER   : integer := 8;
11
12             -- minimum number of bits to describe neurons_number
13     constant SUM_SIZE         : integer := 4;
14
15         -- inputs_number/neurons_number
16     constant ADD_SIZE         : integer := INPUTS_NUMBER/NEURONS_NUMBER;
17
18         -- 2^ADD_SIZE
19     constant WORDS_NUMBER     : integer := 2**ADD_SIZE;
20
21     type class_vector_t is array(CLASS_NUMBER-1 downto 0) of
22         integer range NEURONS_NUMBER downto 0;
23
24     component wisard_top is
25         port (
26             rst_i       : in std_logic;
27             clk_i       : in std_logic;
28
29             start_i     : in std_logic;
30             done_o      : out std_logic;
31
32             input_i     : in std_logic_vector(INPUTS_NUMBER-1 downto 0);
33             class_i     : in integer range CLASS_NUMBER-1 downto 0;
34
35             class_o     : out class_vector_t;
36
37             -- '1' = test, '0' = learn
38             mode_i      : in std_logic
39         );
40     end component;
41
42     component wisard_disc is
43         port (
44             rst_i       : in std_logic;
45             clk_i       : in std_logic;
46
47             start_i     : in std_logic;
48             done_o      : out std_logic;
```

```
49
50              input_i      : in std_logic_vector(INPUTS_NUMBER-1 downto 0);
51
52              -- '1' = test, '0' = learn
53              mode_i       : in std_logic;
54
55              sum_o        : out integer range NEURONS_NUMBER downto 0
56          );
57      end component;
58
59      component wisard_neuron is
60
61          port (
62              rst_i        : in std_logic;
63              clk_i        : in std_logic;
64
65              -- '1'  = read, '0' = write
66              rw_i         : in std_logic;
67              address_i    : in integer range WORDS_NUMBER-1 downto 0;
68
69              data_o       : out std_logic
70          );
71      end component;
72
73 end wisard_pkg;
```

Módulo wisard_neuron

```
1 -- Module        : wisard_neuron.vhd
2 -- Description   : memory of 1 bit words
3
4 library ieee;
5 use ieee.std_logic_1164.all;
6 use ieee.numeric_std.all;
7
8 library wisard;
9 use wisard.wisard_pkg.all;
10
11 entity wisard_neuron is
12
13     port (
14         rst_i        : in std_logic;
15         clk_i        : in std_logic;
16
17         -- '1'  = read, '0' = write
18         rw_i         : in std_logic;
19         address_i    : in integer range WORDS_NUMBER-1 downto 0;
20
21         data_o       : out std_logic
22     );
23 end wisard_neuron;
24
25 architecture rtl of wisard_neuron is
26     signal memory_s : std_logic_vector(WORDS_NUMBER-1 downto 0);
27 begin
28
29     data_o <= memory_s(address_i);
30
31     process(clk_i, rst_i)
32     begin
33         if (rst_i = '1') then
34             memory_s <= (others => '0');
35         elsif (rising_edge(clk_i)) then
```

```
36              if (rw_i = '0') then
37                  memory_s(address_i) <= '1';
38              end if;
39          end if;
40      end process;
41
42 end rtl;
```

Módulo wisard_disc

```
1 -- Module         : wisard_disc
2 -- Description    : Wisard neural network discriminator
3 --                  Each discriminator represents one class and
4 --                  has one or more neurons
5
6 library ieee;
7 use ieee.std_logic_1164.all;
8 use ieee.numeric_std.all;
9
10 library wisard;
11 use wisard.wisard_pkg.all;
12
13 entity wisard_disc is
14     port (
15         rst_i       : in std_logic;
16         clk_i       : in std_logic;
17
18         start_i     : in std_logic;
19         done_o      : out std_logic;
20
21         input_i     : in std_logic_vector(INPUTS_NUMBER-1 downto 0);
22
23         -- '1' = test, '0' = learn
24         mode_i      : in std_logic;
25
26         sum_o       : out integer range NEURONS_NUMBER downto 0
27     );
28 end wisard_disc;
29
30 architecture rtl of wisard_disc is
31
32     type state_t is (IDLE, S1, S2, S3);
33
34     signal state_fsm, nstate_fsm : state_t;
35
36     signal i_cnt              : integer range NEURONS_NUMBER downto 0;
37     signal result_s           : integer range NEURONS_NUMBER downto 0;
38     signal rw_neuron_s        : std_logic;
39     signal data_neuron_s      : std_logic_vector(NEURONS_NUMBER-1 downto 0);
40
41     type address_t is array(NEURONS_NUMBER-1 downto 0)
42         of integer range WORDS_NUMBER-1 downto 0;
43
44     signal address_neuron_s : address_t;
45 begin
46
47     neurons_gen: for i in 0 to NEURONS_NUMBER-1 generate
48         neuron_u : wisard_neuron
49             port map (
50                 rst_i       => rst_i,
51                 clk_i       => clk_i,
52                 rw_i        => rw_neuron_s,
53                 address_i   => address_neuron_s(i),
```

```
54                  data_o       => data_neuron_s(i)
55              );
56      end generate;
57
58      process(clk_i, rst_i)
59      begin
60          if rst_i = '1' then
61              state_fsm <= IDLE;
62          elsif rising_edge(clk_i) then
63              state_fsm <= nstate_fsm;
64          end if;
65      end process;
66
67      process(state_fsm, start_i, i_cnt, mode_i)
68      begin
69          nstate_fsm <= state_fsm;
70
71          case state_fsm is
72              when IDLE =>
73                  if start_i = '1' then
74                      if mode_i = '1' then
75                          nstate_fsm <= S1;
76                      else
77                          nstate_fsm <= S3;
78                      end if;
79                  end if;
80              when S1 => -- testing mode
81                  nstate_fsm <= S2;
82              when S2 =>
83                  if i_cnt = NEURONS_NUMBER then
84                      nstate_fsm <= IDLE;
85                  end if;
86              when S3 => -- learn mode
87                  nstate_fsm <= IDLE;
88          end case;
89      end process;
90
91      process(clk_i, rst_i)
92      begin
93          if rst_i = '1' then
94              done_o <= '0';
95              rw_neuron_s <= '1' ;
96              sum_o <= 0;
97        i_cnt <= 0;
98        result_s <= 0;
99              for i in NEURONS_NUMBER-1 downto 0 loop
100                 address_neuron_s(i) <= 0;
101             end loop;
102         elsif rising_edge(clk_i) then
103             case state_fsm is
104                 when IDLE =>
105                     rw_neuron_s <= '1';
106                     done_o <= '0';
107                     i_cnt <= 0;
108                     result_s <= 0;
109                     sum_o <= 0;
110                 -- test
111                 when S1 =>
112                     for i in NEURONS_NUMBER-1 downto 0 loop
113                         address_neuron_s(i) <= to_integer(unsigned(input_i((i+1)
                                *ADD_SIZE-1 downto i*ADD_SIZE)));
114                     end loop;
115
116                     rw_neuron_s <= '1';
117
```

```
118                 when S2 =>
119                     if i_cnt = NEURONS_NUMBER then
120                         done_o <= '1';
121                         sum_o <= result_s;
122                     else
123                         if data_neuron_s(i_cnt) = '1' then
124                             result_s <= result_s + 1;
125                         end if;
126                         i_cnt <= i_cnt + 1;
127                     end if;
128 
129                 -- train
130                 when S3 =>
131                     for i in NEURONS_NUMBER-1 downto 0 loop
132                         address_neuron_s(i) <= to_integer(unsigned(input_i((i+1)
                                *ADD_SIZE-1 downto i*ADD_SIZE)));
133                     end loop;
134                     rw_neuron_s <= '0';
135                     done_o <= '1';
136             end case;
137         end if;
138     end process;
139 
140 end rtl;
```

Módulo wisard_top

```
1 -- Module       : wisard_top.vhd
2 -- Description  : top level entity of wisard neura network.
3 --                Each discriminator is responsible for one class
4 
5 library ieee;
6 use ieee.std_logic_1164.all;
7 use ieee.numeric_std.all;
8 
9 library wisard;
10 use wisard.wisard_pkg.all;
11 
12 entity wisard_top is
13     port (
14         rst_i       : in std_logic;
15         clk_i       : in std_logic;
16 
17         start_i     : in std_logic;
18         done_o      : out std_logic;
19 
20         input_i     : in std_logic_vector(INPUTS_NUMBER-1 downto 0);
21         class_i     : in integer range CLASS_NUMBER-1 downto 0;
22 
23         class_o     : out class_vector_t;
24 
25         -- '1' = test, '0' = learn
26         mode_i      : in std_logic
27     );
28 end wisard_top;
29 
30 architecture rtl of wisard_top is
31 
32     type state_t is (IDLE, S1, S2, S3);
33 
34     signal state_fsm, nstate_fsm : state_t;
35 
36     signal start_s        : std_logic_vector(CLASS_NUMBER-1 downto 0);
```

```
37     signal done_s       : std_logic_vector(CLASS_NUMBER-1 downto 0);
38     signal mode_s       : std_logic_vector(CLASS_NUMBER-1 downto 0);
39 
40     signal sum_s        : class_vector_t;
41 
42 begin
43 
44     disc_gen: for i in CLASS_NUMBER-1 downto 0 generate
45         disc_u: wisard_disc
46             port map (
47                 rst_i   => rst_i,
48                 clk_i   => clk_i,
49                 start_i => start_s(i),
50                 done_o  => done_s(i),
51                 input_i => input_i,
52                 mode_i  => mode_s(i),
53                 sum_o   => sum_s(i)
54             );
55     end generate;
56 
57     process(clk_i, rst_i)
58     begin
59         if rst_i='1' then
60             done_o <= '0';
61             class_o <= (others => 0);
62             start_s <= (others => '0');
63             mode_s <= (others => '1');
64         elsif rising_edge(clk_i) then
65             case state_fsm is
66                 when IDLE =>
67                     done_o <= '0';
68                     if start_i = '1' then
69                         mode_s(class_i) <= mode_i;
70                         if mode_i='0' then
71                             start_s(class_i) <= '1';
72                         else
73                             start_s <= (others => '1');
74                         end if;
75                     else
76                         mode_s <= (others => '1');
77                         start_s <= (others => '0');
78                     end if;
79                 when S1 => -- learn
80                     start_s <= (others => '0');
81 
82                     if done_s(class_i) = '1' then
83                         done_o <= '1';
84                     end if;
85                 when S2 => -- test
86                     start_s <= (others => '0');
87 
88                     if done_s(0) = '1' then
89                         done_o <= '1';
90                         for i in CLASS_NUMBER-1 downto 0 loop
91                             class_o(i) <= sum_s(i);
92                         end loop;
93                     end if;
94                 when others =>
95             end case;
96         end if;
97     end process;
98 
99     process(clk_i, rst_i)
100    begin
101        if rst_i = '1' then
```

```
102                 state_fsm <= IDLE;
103             elsif rising_edge(clk_i) then
104                 state_fsm <= nstate_fsm;
105             end if;
106         end process;
107 
108         process(state_fsm, start_i, done_s, mode_i)
109         begin
110             nstate_fsm <= state_fsm;
111 
112             case state_fsm is
113                 when IDLE =>
114                     if start_i = '1' then
115                         if mode_i = '1' then
116                             nstate_fsm <= S2;
117                         else
118                             nstate_fsm <= S1;
119                         end if;
120                     end if;
121                 when S1 => -- learn mode
122                     if done_s(class_i) = '1' then
123                         nstate_fsm <= IDLE;
124                     end if;
125                 when S2 => -- test mode
126                     if done_s(0) = '1' then
127                         nstate_fsm <= IDLE;
128                     end if;
129                 when others =>
130                     nstate_fsm <= IDLE;
131             end case;
132         end process;
133 
134 end rtl;
```

Testbench tb_wisard_disc

```
 1 -- Module        : tb_wisard_disc
 2 -- Description   : Wisard neural network discriminator testbench
 3 
 4 library ieee;
 5 use ieee.std_logic_1164.all;
 6 use ieee.numeric_std.all;
 7 
 8 use work.wisard_pkg.all;
 9 
10 entity tb_wisard_disc is
11 end tb_wisard_disc;
12 
13 architecture behavioral of tb_wisard_disc is
14     signal clk_s     : std_logic;
15     signal rst_s     : std_logic;
16     signal start_s   : std_logic;
17     signal done_s    : std_logic;
18     signal input_s   : std_logic_vector(INPUTS_NUMBER-1 downto 0);
19     signal mode_s    : std_logic;
20     signal sum_s     : integer range NEURONS_NUMBER downto 0;
21 
22     constant CLK_PERIOD : time := 10 ns;
23     constant RST_TIME   : time := 40 ns;
24 
25 begin
26 
27     disc_u : wisard_disc
```

```
28         port map (
29             rst_i    => rst_s,
30             clk_i    => clk_s,
31             start_i  => start_s,
32             done_o   => done_s,
33             input_i  => input_s,
34             mode_i   => mode_s,
35             sum_o    => sum_s
36         );
37
38     process
39     begin
40         clk_s <= '0';
41         wait for CLK_PERIOD/2;
42         clk_s <= '1';
43         wait for CLK_PERIOD/2;
44     end process;
45
46     process
47     begin
48         rst_s <= '1';
49         wait for RST_TIME;
50         rst_s <= '0';
51         wait;
52     end process;
53
54     process
55     begin
56         start_s <= '0';
57         mode_s <= '0';
58         wait for RST_TIME;
59         wait until rising_edge(clk_s);
60         start_s <= '1';
61         wait until rising_edge(clk_s);
62         start_s <= '0';
63         wait until done_s = '1';
64         wait until rising_edge(clk_s);
65         mode_s <= '1';
66         start_s <= '1';
67         wait until rising_edge(clk_s);
68         start_s <= '0';
69         wait;
70     end process;
71
72     process
73     begin
74         input_s <= "1111100110011111";
75                 -- 1111
76                 -- 1001
77                 -- 1001
78                 -- 1111
79         wait;
80     end process;
81
82 end behavioral;
```

Testbench tb_wisard_top

```
1 -- Module       : tb_wisard_top
2 -- Description  : Wisard top level testbench
3
4 library ieee;
5 use ieee.std_logic_1164.all;
```

```
6 use ieee.numeric_std.all;
7
8 library wisard;
9 use wisard.wisard_pkg.all;
10
11 entity tb_wisard_top is
12 end tb_wisard_top;
13
14 architecture behavioral of tb_wisard_top is
15     signal clk_s         : std_logic;
16     signal rst_s         : std_logic;
17     signal start_s       : std_logic;
18     signal done_s        : std_logic;
19     signal input_s       : std_logic_vector(INPUTS_NUMBER-1 downto 0);
20     signal class_i_s     : integer range CLASS_NUMBER-1 downto 0;
21     signal class_o_s     : class_vector_t;
22     signal mode_s        : std_logic;
23
24     constant CLK_PERIOD : time := 10 ns;
25     constant RST_TIME : time := 40 ns;
26
27 begin
28
29     top_u: wisard_top
30         port map (
31             rst_i    => rst_s,
32             clk_i    => clk_s,
33             start_i => start_s,
34             done_o   => done_s,
35             input_i => input_s,
36             class_i => class_i_s,
37             class_o => class_o_s,
38             mode_i   => mode_s
39         );
40
41     process
42     begin
43         clk_s <= '0';
44         wait for CLK_PERIOD/2;
45         clk_s <= '1';
46         wait for CLK_PERIOD/2;
47     end process;
48
49     process
50     begin
51         rst_s <= '1';
52         wait for RST_TIME;
53         rst_s <= '0';
54         wait;
55     end process;
56
57     process
58     begin
59         input_s <= "0000000000000000";
60         start_s <= '0';
61         mode_s <= '0'; -- learn class 0
62         wait for RST_TIME;
63         wait until rising_edge(clk_s);
64         input_s <= "1111100110011111";
65                 -- 1111
66                 -- 1001
67                 -- 1001
68                 -- 1111
69         start_s <= '1';
70         class_i_s <= 0;
```

```
71          wait until rising_edge(clk_s);
72          start_s <= '0';
73          wait until done_s = '1';
74          wait until rising_edge(clk_s); -- learn class 1
75          start_s <= '1';
76          class_i_s <= 1;
77          input_s <= "0100010001000100";
78                  -- 0100
79                  -- 0100
80                  -- 0100
81                  -- 0100
82          wait until rising_edge(clk_s);
83          start_s <= '0';
84          wait until done_s = '1';
85          wait until rising_edge(clk_s); -- learn class 2
86          input_s <= "1111001001001111";
87                  -- 1111
88                  -- 0010
89                  -- 0100
90                  -- 1111
91          start_s <= '1';
92          class_i_s <= 2;
93          wait until rising_edge(clk_s);
94          start_s <= '0';
95          wait until done_s = '1';
96          wait until rising_edge(clk_s); -- test class 0
97          mode_s <= '1';
98          start_s <= '1';
99          input_s <= "0111100110011110";
100                 -- 0111
101                 -- 1001
102                 -- 1001
103                 -- 1110
104         wait until rising_edge(clk_s);
105         start_s <= '0';
106         wait until done_s = '1';
107         wait until rising_edge(clk_s); -- test class 1
108         input_s <= "0100010001000010";
109                 -- 0100
110                 -- 0100
111                 -- 0100
112                 -- 0010
113         start_s <= '1';
114         wait until rising_edge(clk_s);
115         start_s <= '0';
116         wait until done_s = '1';
117         wait until rising_edge(clk_s); -- test class 2
118         input_s <= "1110001001001111";
119                 -- 1110
120                 -- 0010
121                 -- 0100
122                 -- 1111
123         start_s <= '1';
124         wait until rising_edge(clk_s);
125         start_s <= '0';
126         wait;
127     end process;
128
129 end behavioral;
```

Testbench tb_syn_wisard_top

```
1 -- Module          : tb_syn_wisard_top.vhd
```

```
2 -- Description  : Wisard synthesizable testbench
3
4 library ieee;
5 use ieee.std_logic_1164.all;
6 use ieee.numeric_std.all;
7
8 library wisard;
9 use wisard.wisard_pkg.all;
10
11 entity tb_syn_wisard_top is
12     port (
13         clk_i   : in std_logic;
14         rst_i   : in std_logic;
15
16         led1_o  : out std_logic;
17         led2_o  : out std_logic;
18         led3_o  : out std_logic
19     );
20 end tb_syn_wisard_top;
21
22 architecture rtl of tb_syn_wisard_top is
23     type results_t is array((CLASS_NUMBER*3-1) downto 0) of
24         integer range NEURONS_NUMBER downto 0;
25     signal results_s    : results_t;
26     signal start_s      : std_logic;
27     signal done_s       : std_logic;
28     signal input_s      : std_logic_vector(INPUTS_NUMBER-1 downto 0);
29     signal class_i_s    : integer range CLASS_NUMBER-1 downto 0;
30     signal class_o_s    : class_vector_t;
31     signal mode_s       : std_logic;
32
33     type state_t is (IDLE, L1, L2, L3, T1, T2, T3);
34
35     signal state_fsm, nstate_fsm : state_t;
36 begin
37     results_s(0) <= 6;
38     results_s(1) <= 1;
39     results_s(2) <= 2;
40     results_s(3) <= 0;
41     results_s(4) <= 6;
42     results_s(5) <= 2;
43     results_s(6) <= 3;
44     results_s(7) <= 2;
45     results_s(8) <= 7;
46
47     top_u: wisard_top
48         port map (
49             rst_i   => rst_i,
50             clk_i   => clk_i,
51             start_i => start_s,
52             done_o  => done_s,
53             input_i => input_s,
54             class_i => class_i_s,
55             class_o => class_o_s,
56             mode_i  => mode_s
57         );
58
59     process(clk_i, rst_i)
60     begin
61         if rst_i = '1' then
62             state_fsm <= IDLE;
63         elsif rising_edge(clk_i) then
64             state_fsm <= nstate_fsm;
65         end if;
66     end process;
```

```
67
68
69     process(clk_i, rst_i)
70     begin
71         if rst_i = '1' then
72             start_s <= '0';
73             input_s <= (others => '0');
74             class_i_s <= 0;
75             mode_s <= '0';
76             led1_o <= '0';
77             led2_o <= '0';
78             led3_o <= '0';
79         elsif rising_edge(clk_i) then
80             case state_fsm is
81                 when IDLE =>
82                     start_s <= '1';
83                     input_s <= "1111100110011111";
84                     class_i_s <= 0;
85                 when L1 =>
86                     start_s <= '0';
87                     if done_s = '1' then
88                         start_s <= '1';
89                         class_i_s <= 1;
90                         input_s <= "0100010001000100";
91                     end if;
92                 when L2 =>
93                     start_s <= '0';
94                     if done_s = '1' then
95                         start_s <= '1';
96                         class_i_s <= 2;
97                         input_s <= "1111001001001111";
98                     end if;
99                 when L3 =>
100                    start_s <= '0';
101                    mode_s <= '0';
102                    if done_s = '1' then
103                        mode_s <= '1';
104                        start_s <= '1';
105                        input_s <= "0111100110011110";
106                    end if;
107                when T1 =>
108                    start_s <= '0';
109                    if done_s = '1' then
110                        if (class_o_s(0) = results_s(0))
111                            and (class_o_s(1) = results_s(1))
112                            and (class_o_s(2) = results_s(2)) then
113                            led1_o <= '1';
114                        end if;
115                        input_s <= "0100010001000010";
116                        start_s <= '1';
117                    end if;
118                when T2 =>
119                    start_s <= '0';
120                    if done_s = '1' then
121                        if (class_o_s(0) = results_s(3))
122                            and (class_o_s(1) = results_s(4))
123                            and (class_o_s(2) = results_s(5)) then
124                            led2_o <= '1';
125                        end if;
126                        input_s <= "1110001001001111";
127                        start_s <= '1';
128                    end if;
129                when T3 =>
130                    start_s <= '0';
131                    if done_s = '1' then
```

```
132                         if (class_o_s(0) = results_s(6))
133                             and (class_o_s(1) = results_s(7))
134                             and (class_o_s(2) = results_s(8)) then
135                             led3_o <= '1';
136                         end if;
137                     end if;
138             end case;
139         end if;
140     end process;
141
142     process(state_fsm, done_s)
143     begin
144         nstate_fsm <= state_fsm;
145
146         case state_fsm is
147             when IDLE =>
148                 nstate_fsm <= L1;
149             when L1 =>
150                 if done_s = '1' then
151                     nstate_fsm <= L2;
152                 end if;
153             when L2 =>
154                 if done_s = '1' then
155                     nstate_fsm <= L3;
156                 end if;
157             when L3 =>
158                 if done_s = '1' then
159                     nstate_fsm <= T1;
160                 end if;
161             when T1 =>
162                 if done_s = '1' then
163                     nstate_fsm <= T2;
164                 end if;
165             when T2 =>
166                 if done_s = '1' then
167                     nstate_fsm <= T3;
168                 end if;
169             when T3 =>
170                 nstate_fsm <= T3;
171             when others =>
172                 nstate_fsm <= IDLE;
173         end case;
174     end process;
175
176
177 end rtl;
```

APÊNDICE C – R^2-ajustado

O coeficiente de determinação R^2 é uma equação que descreve a proporção da variação em uma variável de resposta Y que pode ser linearmente predito pelas covariáveis do modelo. Contudo, esse coeficiente é viesado, aumentando com o número de variáveis preditoras. Esse viés é removido com a utilização de R^2-ajustado, dado pela seguinte equação (RUPPERT, 2011):

$$AdjR^2 = 1 - \left[\left(\frac{\sum_{i=1}^{N} (Y_i - \hat{Y}_i)^2}{\sum_{i=1}^{N} (Y_i - \bar{Y})^2} \right) \left(\frac{N-1}{N-P-1} \right) \right] \tag{1}$$

onde N é o número de amostras, Y_i o i-ésimo valor observado, $\hat{Y}_i$ o i-ésimo valor predito, $\bar{Y}$ a média aritmética dos valores observados e P o número de variáveis preditoras.