UNIVERSIDADE FEDERAL DA BAHIA
ESCOLA POLITÉCNICA DEPARTAMENTO DE ENGENHARIA ELÉTRICA
CURSO DE MESTRADO EM ENGENHARIA ELÉTRICA
CMEE-DEE-EP-UFBA

Projeto em Circuito Integrado de uma Arquitetura Flexível de Neuroprocessador de Alto Desempenho para Redes Perceptron Multicamadas

Igor Dantas dos Santos Miranda

Orientadora: Prof.ª Dr.ª Ana Isabela Araújo Cunha

Dissertação de Mestrado
apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da UFBA (área de concentração: Microeletrônica) como requisito parcial a obtenção do grau de Mestre em Engenharia Elétrica.

Número de ordem PPGEE:
Salvador, BA, dezembro de 2009

Divisão de Serviços Técnicos

Catalogação da publicação na fonte. UFBA / Biblioteca Central

Miranda, Igor Dantas dos Santos.

Projeto em Circuito Integrado de uma Arquitetura Flexível de Neuroprocessador de Alto Desempenho para Redes Perceptron Multicamadas / Igor Dantas dos Santos Miranda - Salvador, BA, 2009

84 p.

Orientadora: Prof.ª Dr.ª Ana Isabela Araújo Cunha

Dissertação (mestrado) - Universidade Federal da Bahia. Escola Politécnica. Programa de Pós-Graduação em Engenharia Elétrica.

BA/UF/ CDU

Projeto em Circuito Integrado de uma Arquitetura Flexível de Neuroprocessador de Alto Desempenho para Redes Perceptron Multicamadas

Igor Dantas dos Santos Miranda

Dissertação de Mestrado aprovada em 21 de dezembro de 2009 pela banca examinadora composta pelos seguintes membros:

Prof.ª Dr.ª Ana Isabela Araújo Cunha (orientadora) ..... DEE/UFBA

Prof.ª Dr.ª Luciana Martinez ..... DEE/UFBA

Prof. Dr. Robson Nunes de Lima ..... DEE/UFBA

Prof. Dr. Tiago de Oliveira ..... DCT/UNIFESP

Agradecimentos

- À minha mãe pelo apoio incondicional e incansável, ao meu pai pelos conselhos e exemplo de perseverança e à minha irmã, Iorrana, pelo companheirismo.

- À minha orientadora, professora Ana Isabela, pela orientação, compreensão e confiança. Suas contribuições e ações tiveram uma importância incontestável para a realização deste trabalho.

- Aos amigos pela compreensão da ausência e dos momentos de stress.

- Aos colegas do LSITEC-NE pelas críticas e sugestões.

- Ao LSITEC-NE por conceder acesso às ferramentas de projeto de circuitos integrados.

Resumo

As redes neurais artificiais continuam sendo uma importante ferramenta na solução de diversos problemas das ciências aplicadas, como biologia, medicina e engenharias. Dentre estes, existem sistemas com fortes requisitos de desempenho, como aqueles destinado ao reconhecimento de padrões de áudio e vídeo, que exigem arquiteturas de *hardware* dedicadas para suportar o processamento massivamente paralelo das redes neurais em alta velocidade.

Adicionalmente, com os novos desafios na integração dos sistemas, em que muitos blocos funcionais são colocados em um único chip, são solicitadas flexibilidade e reusabilidade dos *hardwares* para redes neurais, visando a implementação em ASIC. Neste trabalho é apresentado um *IP-Core* de uma arquitetura flexível de neuroprocessador de alto desempenho para o processamento de redes neurais do tipo *perceptron* multi-camadas para implementação em circuito integrado.

O número de unidades e de núcleos de processamento, a precisão e a quantidade de memória para os pesos são recursos de *hardware* ajustáveis durante a implementação. A topologia da rede, os valores dos pesos e a função de ativação são configuráveis durante a execução. O projeto lógico do neuroprocessador proposto é apresentado, além de um exemplo de layout para validação da viabilidade física do projeto, utilizando a tecnologia CMOS de 90nm.

Uma nova métrica de avaliação de desempenho é apresentada e utilizada na comparação do desempenho do neuroprocessador proposto com outros encontrados na literatura e disponíveis comercialmente.

Palavras-chave: Redes Neurais, Neuroprocessadores, Perceptron Multi-Camadas, ASIC.

Abstract

Artificial Neural Networks have been an important tool to solve several problems in science, biology, medicine and engineering. Some of these systems present severe performance requirements. This is the case of the audio and video pattern recognition systems, which demand dedicated hardware architecture in order to support the massively parallel processing of the neural networks at high speeds.

Moreover, together with the new challenges in system integration, where a single chip comprises many functional blocks, features as flexibility and reusability are required by neural network hardwares that target ASIC implementation. This work presents an IP-Core of a flexible and high speed neuroprocessor architecture for computing multi-layered neural networks targeting integrated circuit implementation.

The number of processing elements and cores, the precision and the weight memory size are adjustable hardware resources available in the circuit implementation. The network topologies, the weight values and the activation function may be configured during the running phase. The neuroprocessor design flow is detailed throughout the work. A layout example has been built to validate the physical feasibility of the design, using CMOS 90 nm technology.

A new performance evaluation metric has been presented and has been used to compare the proposed neuroprocessor with some neurohardwares found in the literature and commercially available.

Keywords: Neural networks, Neuroprocessors, Multi-layers Perceptron, ASIC.

Sumário

Sumário i

Lista de Figuras iii

Lista de Tabelas v

Lista de Símbolos e Abreviaturas vii

1 Introdução 1

2 Hardware para Redes Neurais 5
2.1 Redes Neurais Artificiais . . . 5
2.2 Considerações na Implementação de RNA . . . 8
2.3 Classificações de Hardware para RNA . . . 9
2.4 Trabalhos Relacionados . . . 10

3 Arquitetura do Neuroprocessador 15
3.1 Requisitos . . . 15
3.2 Processamento discreto das redes MLP . . . 16
3.3 Visão Geral da Arquitetura . . . 16
3.4 Protocolo de Comunicação . . . 19
3.5 Descrição do Kernel . . . 23
3.5.1 Caminho de Dados . . . 23
3.5.2 Unidade de Processamento . . . 25
3.5.3 Função de Ativação . . . 28
3.5.4 Arquitetura de Memória . . . 29
3.5.5 Unidade de Controle . . . 34
3.6 Virtualização . . . 34
3.7 Sumário . . . 35

4 Projeto em Circuito Integrado 39
4.1 Metodologia de Projeto . . . 39
4.2 Front-end . . . 43
4.2.1 Modelo de Referência . . . 43
4.2.2 Projeto Lógico . . . 46
4.2.3 Verificação Funcional . . . 52
4.2.4 Síntese Lógica . . . 54

4.2.5 Geração de Memórias 55
4.3 Back-end 56

5 Avaliação de Características Físicas e de Desempenho **59**
5.1 Análises do Erro 59
5.2 Análise de Desempenho 60
5.3 Características Físicas 64

6 Conclusões **67**

Referências bibliográficas **69**

A Código-fonte em Verilog da UP **73**

B Folha de Especificações de Bloco de Memória **77**

Lista de Figuras

2.1 Neurônio Biológico [Haykin 1999] 5
2.2 Componentes do Neurônio Artificial (Figura adaptada de [Omondi e Rajapakse 2006]) 6
2.3 Exemplo de rede MLP de 3 camadas, com 4 entradas e 3 neurônios em cada camada (Figura adaptada de [Demuth et al. 2008]) 7
2.4 Grupos de classificação de neurohardware segundo [Ienne 1997] 9
2.5 Arquitetura BBA (Figura adaptada de [Ienne 1993]) 11
2.6 Modelo do neurohardware (esquerda) e da UP (direita) apresentados por [Yun et al. 2002] 11
2.7 Esquema funcional do neurohardware apresentado por [Vitabile et al. 2005] 12
2.8 Esquema funcional do CogniMem [CogniMem 2008] 13

3.1 Kernel de núcleo simples 18
3.2 Kernel de núcleo duplo 19
3.3 Esquema para três camadas no modo desempenho 20
3.4 Conexão entre blocos comunicantes 20
3.5 (a) Transmissão simples (b) Dispositivo de recepção ocupado 22
3.6 Caminho de Dados de Núcleo Simples 24
3.7 Caminho de Dados de Núcleo Duplo 25
3.8 Diagrama da Unidade de Processamento 26
3.9 Esquema de truncamento na UP 27
3.10 Arquitetura da memória de pesos 31
3.11 Atribuição de dados ao banco virtual 32
3.12 Exemplo de configuração para diversas topologias de RNA 33
3.13 Arquitetura de memória incluindo LUT e configuração 34
3.14 Máquina de estados para um núcleo do caminho de dados 36
3.15 Esquema de virtualização 37

4.1 Versão simplificada das etapas da metodologia de células padrões 40
4.2 Verificações realizadas durante o desenvolvimento (Figura adaptada de [Xiu 2007]) 41
4.3 Visão geral do fluxo de projeto 41
4.4 Sub-etapas do front-end 42
4.5 Sub-etapas do back-end 43
4.6 Comparação entre modelo de referência e projeto lógico durante simulação 44
4.7 Diagrama do modelo de referência 45
4.8 Organização hierárquica do RTL 47

4.9 Esquemático da UP 49
4.10 Sinais do processamento na UP 50
4.11 Envio de dados ao neuroprocessador 51
4.12 Simulação do sistema com um testbench (Figura adaptada de [Xiu 2007]) 52
4.13 Ambiente de Simulação do LVM 54
4.14 Visão externa da memória gerada 56
4.15 Layout do neuroprocessador 58

5.1 Área e potência em função do número de UPs 65

Lista de Tabelas

3.1 Sinais no protocolo de comunicação 21
3.2 Registradores de configuração 35
3.3 Sumário das propriedades do neuroprocessador 37

4.1 Resultados da Síntese 55

5.1 Erro no Proben1 e no Matlab 60
5.2 Erro introduzido com o cálculo em ponto fixo de 16 bits no RTL 60
5.3 Análise de desempenho para simulações no modo desempenho 62
5.4 Análise de desempenho para simulações no modo área com 8 UPs 63
5.5 Comparações do neuroprocessador proposto com outros *neurohardwares* 63
5.6 Sínteses para diversos números de UPs 64
5.7 Resultados da síntese para maior frequência de operação suportada 64

Lista de Símbolos e Abreviaturas

E_C	Erro percentual de classificação dos padrões
E_Q	Erro quadrático percentual
n_Q	Eficiência da paralelização
ADC	*Analog to Digital Converter*
ASIC	*Application Specific Integrated Circuit*
BBA	*Broadcast Bus Architecture*
BFM	*Bus Functional Model*
CI	Circuito Integrado
CMOS	*Complementary Metal-Oxide-Semiconductor*
CPCPU	Conexões por Ciclo por Unidade de Processamento
CTS	*Clock Tree Synthesis*
DAC	*Digital to Analog Converter*
EDA	*Electronic Design Automation*
FIFO	*First in First Out*
FPGA	*Field Programmable Gate Array*
FPNA	*Field Programmable Neural Array*
FPNN	*Field Programmed Neural Network*
HDL	*Hardware Description Language* - Linguagem de Descrição de *Hardware*
IHM	Interface humano máquina
IP	*Intellectual Property*
LSB	*Least Significant Bit* - Dígito Menos Significativo
LSITEC-NE	Laboratório de Sistemas Integráveis Tecnológico Nordeste

LUT	*Look-up Table*
LVM	*LSITEC-NE Verification Methodology* - Metodologia de Verificação do LSITEC-NE
MAC	Unidade Multiplica-Acumula
MCPS	Milhões de Conexões por Segundo
MIPS	Milhões de Instruções Por Segundo
MLP	*Multi Layer Perceptron*
MSB	*Most Significant Bit* - Dígito Mais Significativo
RNA	Rede Neural Artificial
RTL	*Register Transfer Level* - Nível de Transferência de Registradores
SoC	*System-on-Chip*
SRAM	*Static Random Access Memory*
UP	Unidade de Processamento
WTA	*Winner Takes All*

Capítulo 1

Introdução

A natureza e a engenharia reservam uma infinidade de problemas que a modelagem matemática formal ainda não consegue resolver. Uma das alternativas para lidar com esses sistemas são as técnicas de inteligência artificial.

As redes neurais artificiais (RNA), que figuram entre as principais técnicas de inteligência artificial, tiveram suas primeiras publicações na década de 40, com o trabalho de McCulloch e Pitts [McCulloch e Pitts 1943]. No entanto, após a publicação do livro Perceptrons de Minsky e Papert em 1969 [Minsky e Papert 1969], no qual uma falsa ineficiência da técnica era citada, as pesquisas a respeito do assunto reduziram significativamente.

Somente cerca de dez anos depois, houve uma retomada dos trabalhos em RNA, com o advento do algoritmo de treinamento *backpropagation* [Rumelhart et al. 1986] e outros resultados teóricos importantes. Desde então, aumentaram vertiginosamente a quantidade e a variedade de aplicações, como relatado em [Kemsley et al. 1992] e [Widrow et al. 1994].Outro fator motivador do renascimento das RNAs foi o aumento da capacidade de processamento dos computadores, fato contemporâneo aos avanços teóricos. Com o poder computacional do *hardware* da época já foi possível executar a grande quantidade de operações aritméticas requeridas em aplicações da técnica a problemas reais.

Com o passar dos anos, a revitalização do interesse nesse campo de pesquisa resultou na aparição de pesquisadores, recursos, conferências e jornais dedicados às redes neurais, além da inclusão do tema na formação básica oferecida por diversas escolas de engenharia e ciência da computação. Então, essa técnica se consolidou como uma importante ferramenta computacional, se tornando uma solução atrativa para inúmeros problemas do mundo real num amplo espectro de aplicações.

Atualmente, as redes neurais são muito utilizadas, segundo [Madani 2006], em problemas relacionados a: otimização; modelagem; sistemas decisórios; classificação de padrões; aproximação de funções não-lineares; mineração de dados.

Dentre essas, classificadores de padrões aplicados a sistemas de reconhecimento de padrões têm sido a utilização mais popular, sendo implementada com sucesso em várias pesquisas acadêmicas e produtos disponíveis no mercado [Bishop 1995]. Alguns exemplos de sistemas de reconhecimento de padrões nos quais as RNAs têm sido utilizadas são:

- manutenção preditiva e preventiva;

- monitoramento de condições;
- reconhecimento de caracteres;
- diagnóstico médico;
- reconhecimento de padrões sonoros;
- reconhecimento de voz;
- síntese da fala;
- reconhecimento de padrões de vídeo.

Diversos produtos finais utilizam ou podem utilizar *hardwares* capazes de processar reconhecimento de padrões. Dentre estes produtos, podemos citar câmeras fotográficas, robôs dotados de visão, identificadores de frutas em supermercados, controladores de acesso por voz e face, brinquedos e detectores de disparos de armas de fogo. No setor automobilístico, esses *hardwares* podem ser utilizados em sistemas de controle adaptativo de velocidade de cruzeiro de automóveis, onde um reconhecedor de padrões analisa a distância entre o automóvel e os veículos ao redor assim como a velocidade média destes para determinar a velocidade de cruzeiro. Também no setor automobilístico, padrões faciais, extraídos do movimento da cabeça e dos olhos, a taxa de piscadela e a média de tempo em que os olhos permanecem fechados, podem ser analisados utilizando um reconhecedor de padrões em sistemas de detecção de sonolência do motorista.

Devido à grande quantidade de informação manipulada, a aplicação de RNA na classificação de padrões de áudio ou vídeo requer soluções de *hardware* de alto desempenho, especialmente para sistemas de tempo real.

Implementação de RNAs em processadores de propósito geral apresentam um desempenho baixo devido à sua natureza seqüencial de operação, em contraste com a característica paralela das RNAs. Se implementadas em *hardware* paralelo, a característica paralela inerente das RNAs pode ser aproveitada para acelerar o processamento em muitas ordens de magnitude. Assim, a aceleração necessária para aplicações de tempo real de alto desempenho pode ser proporcionada por um conjunto de unidades de processamento simples, trabalhando juntas em paralelo.

Diante dessa necessidade, surgiu uma categoria de *hardware* dedicado ao processamento de RNAs, chamada *neurohardware*. *Neurohardware* é definido como qualquer sistema de *hardware* dedicado ao processamento de RNAs, dando suporte parcial ou total ao paralelismo da rede. O *neurohardware* pode ser projetado como um sistema composto de vários subsistemas, passando a ser chamado neurocomputador. Pode também ser projetado como um chip independente que faz todo o processamento neural, os chamados neuroprocessadores.

Vale ressaltar que, com o avanço das tecnologias de integração de circuitos, todos os blocos funcionais dos dispositivos de aplicação estão sendo implementados em um único chip, como em modems Zig-bee e leitores de mídia Blue-Ray. Em um nível de integração ainda maior, há o paradigma do System-on-Chip (SoC), em que um sistema eletrônico inteiro é integrado em uma pastilha de circuito integrado, reunindo diversos dispositivos de aplicação como ADCs, DACs, microprocessadores, modems, gestão de IHM, memória etc. Uma vez que, em geral, os *neurohardwares* são apenas um dos blocos funcionais do sistema, eles devem ser projetados visando a sua integração no dispositivo de aplicação. Por exemplo, para sistemas de reconhecimento de voz em Circuito Integrado (CI), a RNA

que faz a classificação dos padrões é apenas um dos blocos funcionais que compõe o processamento da informação, pois existe o pré-processamento da informação. Este CI, por sua vez, pode ser integrado em um SoC para um celular, que possuirá muitas outras funcionalidades além do reconhecimento de voz. Então, os neuroprocessares precisam ser desenvolvidos como blocos de Propriedade Intelectual (IP), que são módulos ASIC projetados para ter fácil integração e alta reusabilidade.

Por conta disso, novos requisitos surgem no projeto de neuroprocessadores visando integrá-los em dispositivos de aplicação ou SoCs. Esses neuroprocessadores devem suportar uma parametrização personalizada da arquitetura visando deixá-los compatíveis com os requisitos do sistema. Em outras palavras, um integrador de sistema deve ser capaz de ajustar os recursos do neuroprocessador para atender requisitos de área, desempenho ou consumo do dispositivo de aplicação. Além disso, o projeto deve oferecer o máximo de independência possível da tecnologia de fabricação. Estas propriedades proporcionarão reusabilidade e escalabilidade ao neuroprocessador.

Uma grande variedade de *neurohardwares* tem sido desenvolvida, como sumarizado em [Dias et al. 2003]. As principais diferenças estão no compromisso entre desempenho e área, no grau de paralelismo, na abordagem da arquitetura do sistema e no tipo de RNA suportada.

Muitos dos *neurohardwares* encontrados na literatura foram desenvolvidos como dispositivos independentes para ASIC ou FPGA, chips que apresentam somente a função de neuroprocessador, como em [Omondi e Rajapakse 2006] e [Dias et al. 2003]. Esse tipo de projeto, em geral, é incompatível com a flexibilidade necessária para integração em dispositivos de aplicação ou SoC.

Outros trabalhos, como [Vitabile et al. 2005], [Yun et al. 2002] e [Girau 2000], apresentaram arquiteturas flexíveis quanto à quantidade de recursos instanciados, tais como o número de Unidades de Processamento (UP) e a quantidade de memória, representando um avanço significativo na direção de IPs para neuroprocessadores.

Entretanto, duas dificuldades são encontradas na utilização dessas arquiteturas para integração com outros blocos em ASIC. A primeira dificuldade está no fato de todas essas arquiteturas terem sido projetadas para FPGA, desconsiderando aspectos importantes relacionados à implementação em ASIC. A segunda dificuldade é que nenhuma das arquiteturas flexíveis encontradas apresenta um esquema de reutilização do neuroprocessador para mais de uma topologia de RNA em tempo de execução. Nessas situações, o desempenho do sistema cai bruscamente devido à necessidade de mudança de pesos sinápticos na memória interna do neuroprocessador.

A partir do que foi discutido anteriormente, os objetivos do trabalho desenvolvido nesta dissertação são os seguintes:

- apresentar uma arquitetura de neuroprocessador para ser implementado como um IP de ASIC;
- desenvolver o projeto lógico do neuroprocessador e estudar sua viabilidade física através de um layout de validação;
- avaliar o desempenho do *hardware* proposto e realizar comparações com outros neuroprocessadores.

O neuroprocessador desenvolvido deve:

- apresentar flexibilidade na escolha dos recursos;
- considerar os aspectos de implementação em CI;
- ter desempenho compatível com os neuroprocessadores encontrados na literatura ou comercialmente;
- processar redes perceptron multicamadas.

A escolha do tipo de RNA, as redes Multi Layer Perceptron (MLP), foi realizada por conta desta ser a categoria mais utilizada na solução de diversos problemas práticos, como mencionado por [Samarasinghe 2006]. Dessa forma o neuroprocessador terá um amplo espectro de aplicação.

As contribuições deste trabalho são:

- a arquitetura de um neuroprocessador flexível de alto desempenho;
- uma métrica para avaliar o desempenho de neuroprocessadores, independente da freqüência de operação ou do número de UPs da implementação.

O trabalho apresentado nesta dissertação está distribuído em seis capítulos.

O segundo capítulo apresenta conceitos importantes no projeto de *neurohardware*. Trabalhos anteriores no desenvolvimento de *neurohardware* também são mostrados.

No terceiro capítulo a arquitetura do neuroprocessador proposto é apresentada em detalhes. As características do projeto são descritas e justificadas.

O capítulo quatro descreve a metodologia e o processo de desenvolvimento do *hardware*, passando pelo projeto lógico e pelo projeto físico.

O capítulo cinco mostra as avaliações das características físicas e de desempenho do neuroprocessador. Uma comparação dos resultados com outros neuroprocessadores encontrados na literatura ou comercialmente também é apresentada.

No último capítulo o trabalho é sumarizado e concluído. As principais contribuições dessa dissertação são discutidas.

Capítulo 2

Hardware para Redes Neurais

2.1 Redes Neurais Artificiais

Classificadas como uma técnica de inteligência artificial, as RNAs, assim como outras técnicas desse segmento, são modelos matemáticos desenvolvidos visando imitar comportamentos inteligentes presentes na natureza.

Dentro do cérebro, os neurônios biológicos são interligados entre si por intermédio de axônios e dentritos. Numa abordagem superficial, nós podemos considerar que estes são espécies de elementos condutores de eletricidade e podem, dessa forma, conduzir uma mensagem de um neurônio para o outro. Os dentritos representam as entradas do neurônio e o axônio a saída, como mostrado na figura 2.1.

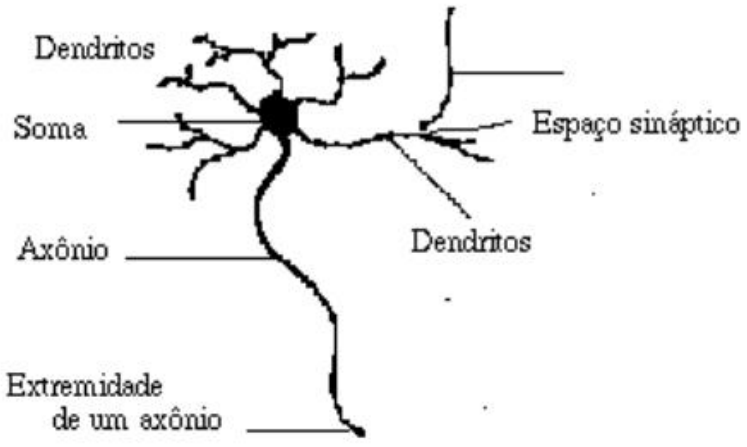


Figura 2.1: Neurônio Biológico [Haykin 1999]

Um neurônio emite um sinal em função dos sinais provenientes de outros neurônios. No neurônio, existe uma integração dos sinais recebidos no decorrer do tempo, podendo ser visto como um somatório dos sinais de entrada.

Na conexão entre um axônio e um dentrito, existe um espaço vazio em que não há condução. Nessa região o sinal é transmitido através de substâncias químicas e depois

convertido novamente para impulsos elétricos. Abstraindo o processo químico, podemos considerar que essas conexões entre os axônios e os dentritos fazem a comunicação entre os neurônios, mas também excitam ou inibem os sinais na entrada de cada neurônio, ponderando suas entradas. Essas conexões ponderadas recebem o nome de sinapses. Mudar as ponderações pode significar aprender.

Esta simples unidade biológica de processamento se interliga aos outros 10 bilhões de neurônios, o que produz cerca de 16 trilhões de interconexões, dotando o cérebro humano com uma capacidade absurda de processamento e com o que chamamos de inteligência.

Partindo dessa idéia, mas com pretensões numéricas bem mais modestas, surgem as RNAs, onde modelos matemáticos simples são agrupados e interconectados visando um sistema com um pouco da capacidade de aprendizagem e generalização do cérebro humano. Os primeiros trabalhos em RNA datam de 1943, onde foi apresentado em [McCulloch e Pitts 1943] um modelo bem simples de neurônio e a exploração de suas possibilidades. O modelo de neurônio que se consolidou foi o perceptron [Rosenblatt 1962], cuja expressão é dada por:

$$y = \varphi\left(\sum_{i=1}^{N} w_i \cdot x_i + \Theta\right), \tag{2.1}$$

Onde N é o número de entradas, w_i os pesos sinápticos, x_i as entradas e Θ é a polarização (ou limiar) do neurônio. φ é a função de ativação, que pode ser linear ou não-linear, dependendo do sistema que se deseja modelar. Diversas funções podem ser usadas como função de ativação, podendo ser encontradas em [Haykin 1999]. A figura 2.2 mostra os componentes básicos de um neurônio artificial.

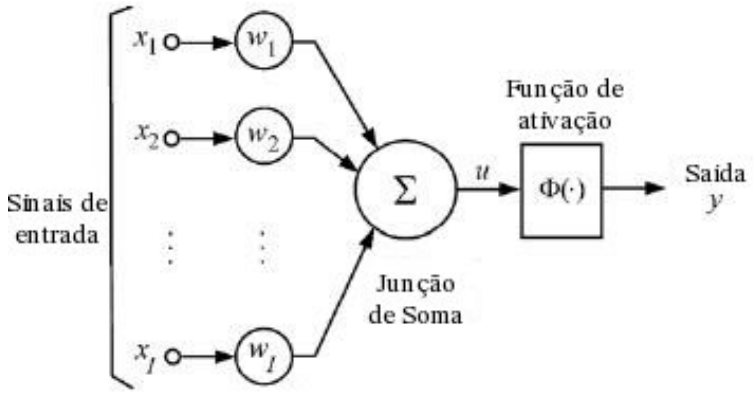


Figura 2.2: Componentes do Neurônio Artificial (Figura adaptada de [Omondi e Rajapakse 2006])

Estabelecido o modelo matemático dos neurônios, basta interconectá-los para termos um comportamento inteligente em pequena escala, com características similares às do cérebro humano. A um grupo destas unidades matemáticas interconectadas dá-se o

nome de RNA. As diversas possibilidades de conexão dão origem às várias categorias de RNA, dentre as quais se destacam as redes perceptron, as redes MLP, os mapas auto-organizáveis e as redes recorrentes. Os modelos dos neurônios também podem mudar com categoria de RNA. O avanço da pesquisa em RNA seguiu em direção à aplicação e não mais ao modelo de funcionamento do cérebro, trazendo categorias voltadas às aplicações específicas e já sem nenhum vínculo com os sistemas biológicos. A definição de RNA dada por [Haykin 1999] faz com que mesmo com modelos distantes das redes biológicas, ainda existam similaridades que as incluam no mesmo grupo.

Como mencionado anteriormente, segundo [Samarasinghe 2006], as redes MLP são as mais conhecidas e utilizadas e, por conta disso, foram adotadas nesse trabalho. Nas redes MLP os neurônios são organizados em camadas, em que existem conexões de um neurônio de uma camada com os da camada posterior como mostrado na figura 2.3. As redes perceptron, constituídas de um ou mais neurônios em uma única camada, podem ser vistas como um caso particular das redes MLP.

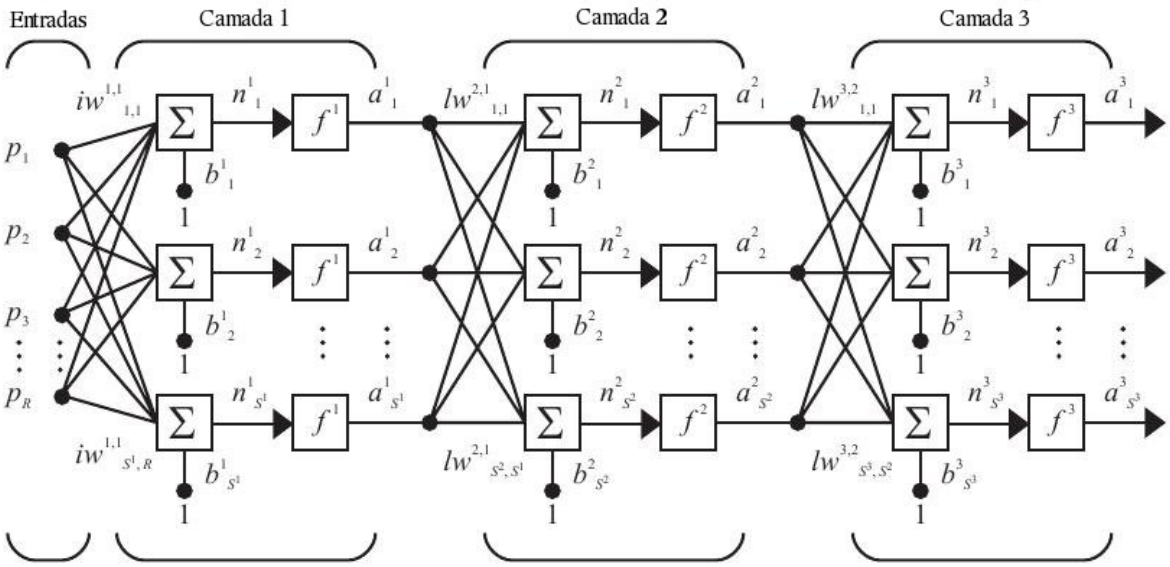


Figura 2.3: Exemplo de rede MLP de 3 camadas, com 4 entradas e 3 neurônios em cada camada (Figura adaptada de [Demuth et al. 2008])

Fazendo uma generalização da equação 2.1, a equação que representa o processamento de cada camada da rede MLP é dado pela equação 2.2.

$$y_k^c = \varphi\left(\sum_{i=1}^{M} w_i^c \cdot y_i^{c-1} + \Theta_k\right), k = 1...G \tag{2.2}$$

Onde M é o número de nós da camada anterior, G é o número de nós da camada atual, o índice c representa a camada, y_i^{c-1} são as saídas da camada anterior e y_k^c são as saídas da camada atual.

O algoritmo *backpropagation* foi o principal precursor das técnicas de treinamento

das RNAs, permitindo a aprendizagem através de exemplos. Desde então, diversos outros algoritmos de treinamento surgiram e são encontrados na literatura, mas esse tópico foge ao escopo deste trabalho.

2.2 Considerações na Implementação de RNA

O projeto de *neurohardware* necessita de abordagens especiais se comparado ao projeto de processadores baseados na arquitetura von Neumann. A principal razão disso é a característica de processamento paralelo, inerente ao conceito das redes neurais. As RNAs podem ser processadas seqüencialmente, porém, em aplicações restritivas com relação ao desempenho, é útil se utilizar do fato de que os dados têm pouca dependência de cálculos anteriores para acelerar a execução.

A primeira questão a ser considerada no projeto está relacionada à escolha entre implementação analógica ou digital. Implementações analógicas têm vantagens sobre as digitais com relação à área, pois os neurônios podem ser implementados com apenas alguns transistores e os pesos sinápticos associados às razões de aspectos dos transistores. No entanto, muitas desvantagens aparecem dentre as quais podem ser citadas a falta de programabilidade, ausência de escalabilidade, baixa imunidade a ruído e a perda de precisão dos pesos devido às incertezas do processo de fabricação [Ienne 1993]. A flexibilidade é um fator essencial no projeto de um neuroprocessador, levando imediatamente à escolha de implementações digitais.

Definida a implementação como digital, os principais aspectos que precisam ser considerados durante a elaboração da arquitetura são o grau de paralelismo, a escalabilidade do *hardware*, a portabilidade entre tecnologias, o armazenamento dos pesos, a virtualização, a precisão e o compromisso entre desempenho e área [Omondi e Rajapakse 2006].

Há vários graus de paralelismo que podem ser considerados: de bit; de neurônio; de camada; de padrões de entrada [Omondi e Rajapakse 2006]. O paralelismo de bit indica que vários bits podem ser processados de uma só vez, como em uma operação aritmética. Processar neurônios, camadas ou padrões diferentes simultaneamente confere paralelismo de neurônio, camada ou de padrões de entrada, respectivamente. Em geral, os paralelismos de bits e neurônios são sempre adotados.

A escalabilidade é a característica do projeto de aumentar os recursos sem prejudicar o funcionamento. Essa é uma característica essencial para arquiteturas que precisam fornecer flexibilidade para adaptar o *hardware* à aplicação [Ienne 1993]. A virtualização e a escalabilidade estão intimamente ligadas, pois quando o número de UPs do *neurohardware* é menor que o número de neurônios da rede, deve haver reuso de recursos. A este reuso de recursos se dá o nome de virtualização [Ienne 1993]. Em muitos casos a escalabilidade do sistema é feita através da virtualização.

A precisão e o compromisso entre área e desempenho são questões comuns encontradas na maioria dos projetos de ASIC digital. Em geral, aumentar o poder de processamento ou a precisão implica no aumento da área e vice-versa [Xiu 2007]. Nos neuroprocessadores este compromisso está relacionado basicamente à quantidade de UPs, enquanto que a questão da precisão relaciona-se com a resolução dos dados e dos pesos.

2.3 Classificações de Hardware para RNA

Como mencionado no capítulo 1, existe uma quantidade considerável de trabalhos publicados acerca da implementação de *hardware* para RNAs. Diversas publicações propuseram classificações para os *neurohardwares* como em [Heemskerk 1995], [Ienne 1997] e [Lindsey e Lindblad 1998], descrevendo métodos próprios de categorização. Devido aos diversos aspectos de implementação, como citado na seção 2.2, e à variedade de técnicas enquadradas como RNAs, criar categorias homogêneas e critérios bem definidos não é uma tarefa simples.

Em geral, a principal forma de classificação dos *neurohardwares* é baseada no tipo de circuito que os implementa. O projeto pode conter circuitos analógicos, digitais ou ambos, resultando nas categorias de *neurohardwares* analógicos, digitais ou híbridos, respectivamente. Como os *neurohardwares* digitais possuem muito mais flexibilidade em termos de configuração e arquitetura que os analógicos, a maioria das outras classificações enquadram somente os digitais.

A respeito dessa classificação, [Ienne 1997] apresenta a categorização para *neurohardwares* digitais mais abrangente em relação aos demais trabalhos pesquisados. As categorias sugeridas por [Ienne 1997] são: tipo de sistema; grau de paralelismo; representação numérica; tipo de comunicação entre as UPs; tipo de partição da rede. A figura 2.4 mostra uma visão geral deste sistema de classificação de *neurohardware*

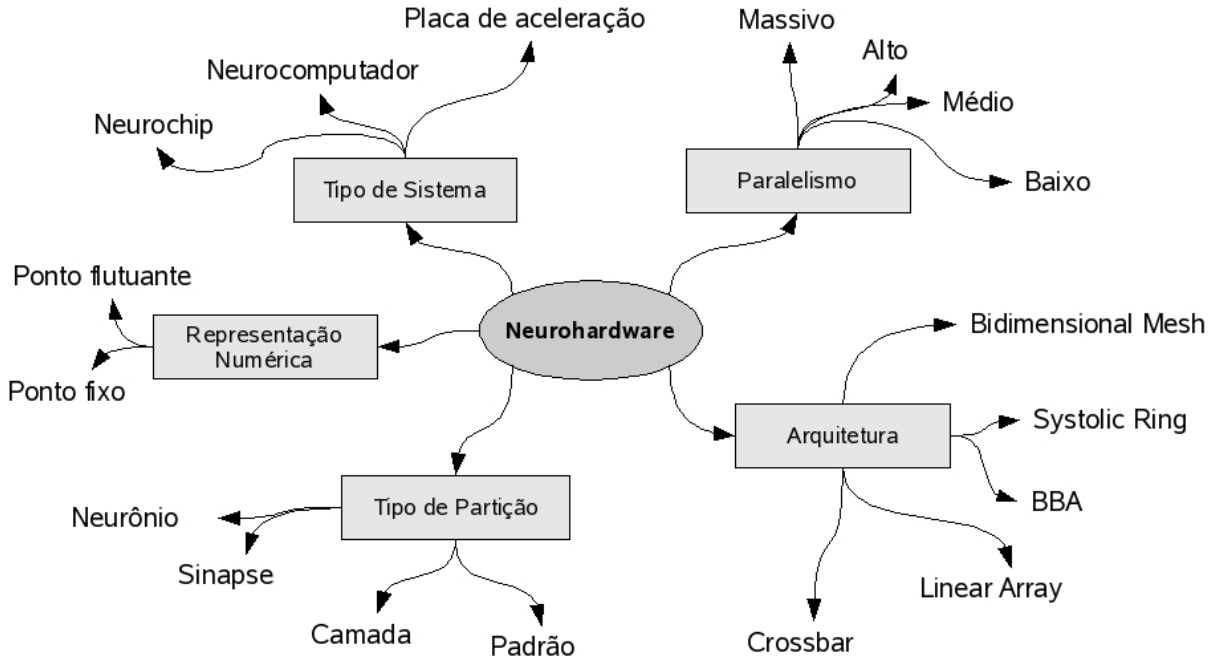


Figura 2.4: Grupos de classificação de neurohardware segundo [Ienne 1997]

Na classificação do tipo do sistema, o *neurohardware* pode ser enquadrado como neuroprocessador (ou neurochip), neurocomputador ou placa de aceleração. Os neurocomputadores são CIs que possuem toda a lógica necessária para o processamento da RNA,

podendo ser usados tanto como um dispositivo independente em uma placa de aplicação ou como um periférico de um microprocessador. Em neurocomputadores e placas de aceleração, neuroprocessadores ou processadores de propósito gerais são utilizados para compor o sistema que processa a RNA. O termo neuroprocessador, ao invés de neurochip, foi adotado durante este trabalho, para evitar confusão com neurochips de redes de comunicação de dados.

O grau de paralelismo é medido através do número de UPs do sistema. [Ienne 1997] propõem uma transição superposta entre as categorias, condizente com a subjetividade da divisão. As categorias para o grau de paralelismo são baixo (2 a 16 UPs), médio (8 a 128 UPs), alto (64 a 2048 UPs) e massivo (maior que 1024).

A classificação pela representação numérica está relacionada com o tipo de operação adotada, que pode ser em ponto fixo ou ponto flutuante.

O tipo de partição da rede define como as RNAs são divididas para executar o processamento. O processamento pode ocorrer neurônio a neurônio, sinapse a sinapse, camada a camada, padrão a padrão etc. Em *neurohardwares* de alto desempenho, mais de um padrão pode ser processado em paralelo.

Por fim, a classificação por tipo de comunicação entre as UPs é, em linhas gerais, a distinção dos *hardwares* com relação às suas arquiteturas. *Bidmensional Mesh*, *Systolic Ring*, *Broadcast Bus*, *Linear Array* e *Crossbar* são as arquiteturas sugeridas em [Ienne 1997], onde o detalhamento de cada uma delas pode ser encontrado.

[Ienne 1997] destaca a arquitetura *Broadcast Bus* (BBA) como a mais indicada quando modularidade e versatilidade são desejadas, como no caso do neuroprocesssador proposto. No BBA, todos os dados de entrada ou proveniente de camadas anteriores são enviados simultaneamente para todas as UPs disponíveis. As UPs efetuam a operação de multiplicar e acumular a cada novo dado, utilizando um peso sináptico armazenado em uma memória interna em cada UP. Uma função de ativação, também interna em cada UP, é então aplicada ao resultado da acumulação ao final da operação. A figura 2.5 mostra o diagrama da BBA.

À medida que novos *neurohardwares* surgiram, as categorias para classificação baseadas na arquitetura passaram a perder a generalização. Outra deficiência das classificações apresentadas por [Ienne 1997] é o fato de que nenhuma categoriza os *hardwares* pelo tipo de RNA processada. Em alguns casos, diferentes tipos de RNAs podem levar a *hardwares* sem nenhuma similaridade.

2.4 Trabalhos Relacionados

Neurohardwares com requisitos de projeto similares aos apresentados no capítulo 1 foram encontrados em [Vitabile et al. 2005], [Yun et al. 2002] e [Girau 2000]. Nestes três trabalhos, flexibilidade, configurabilidade e escalabilidade são as principais características dos *neurohardwares*, assim como esperado para o neuroprocessador proposto.

Outra semelhança entre esses três *neurohardwares* é que todos utilizam a arquitetura BBA para a comunicação dos dados internos, como no diagrama da figura 2.5. As figuras 2.7 e 2.6 mostram os diagramas dos *neurohardwares* apresentados em [Vitabile et al. 2005] e [Yun et al. 2002], respectivamente.

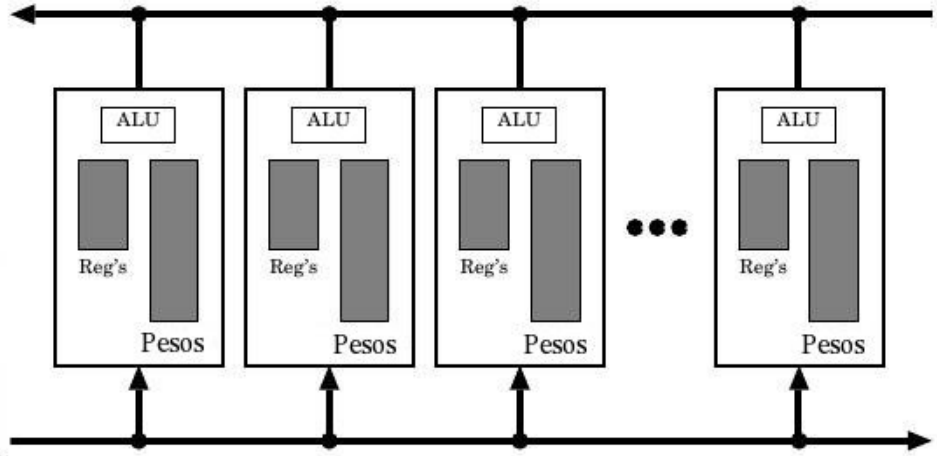


Figura 2.5: Arquitetura BBA (Figura adaptada de [Ienne 1993])

Entretanto, os três neuroprocessadores referidos foram projetados para implementação em FPGA, não atendendo por completo os requisitos estabelecidos no capítulo 1, principalmente no que se refere à implementação em ASIC. O principal problema causado pelo fato do projeto ser direcionar à tecnologia de FPGA está no armazenamento dos pesos dentro da UP, tal como na BBA. Este armazenamento disperso, necessário devido à estrutura interna da maioria dos FPGAs, pode reduzir a configurabilidade e dificultar o layout, como será discutido com mais detalhes adiante neste documento. Alguns FPGAs apresentam memória interna separada da lógica, porém não eliminam os problemas por conta das poucas portas de escrita e leitura.

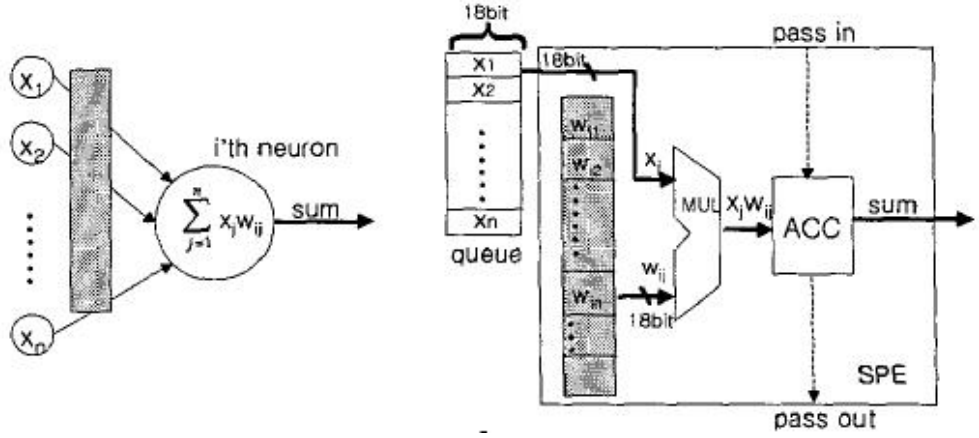


Figura 2.6: Modelo do neurohardware (esquerda) e da UP (direita) apresentados por [Yun et al. 2002]

Também foram encontrados neuroprocessadores no mercado, dentre os quais destaca-se o CogniMem [CogniMem 2008]. Este destina-se ao reconhecimento de padrões de

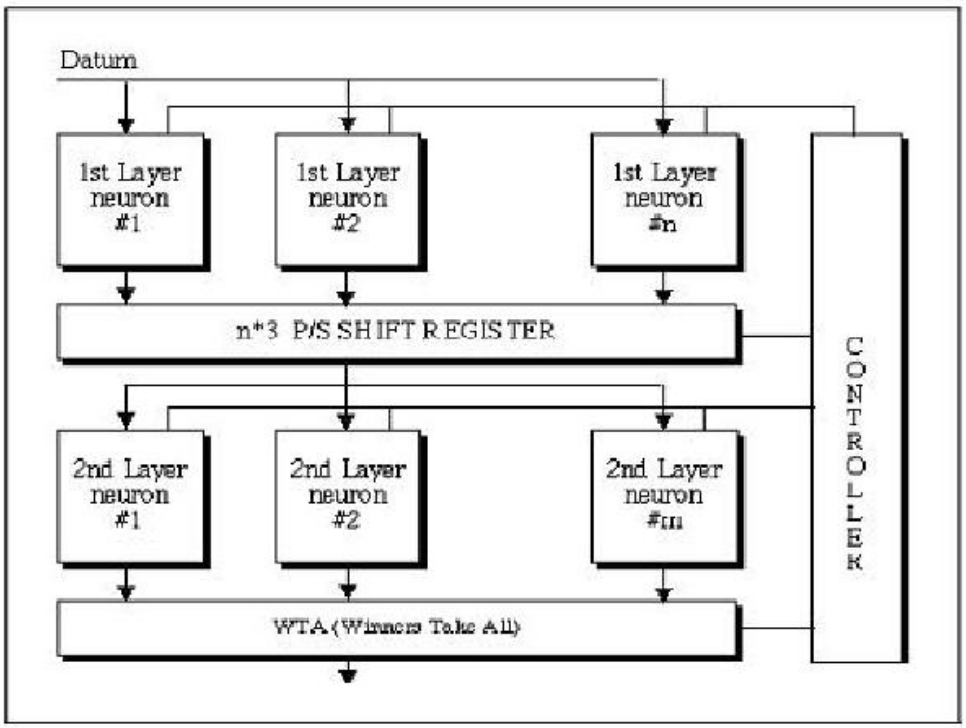


Figura 2.7: Esquema funcional do neurohardware apresentado por [Vitabile et al. 2005]

vídeo e foi implementado com uma arquitetura específica em ASIC. A figura 2.8 mostra o diagrama funcional do CogniMem.

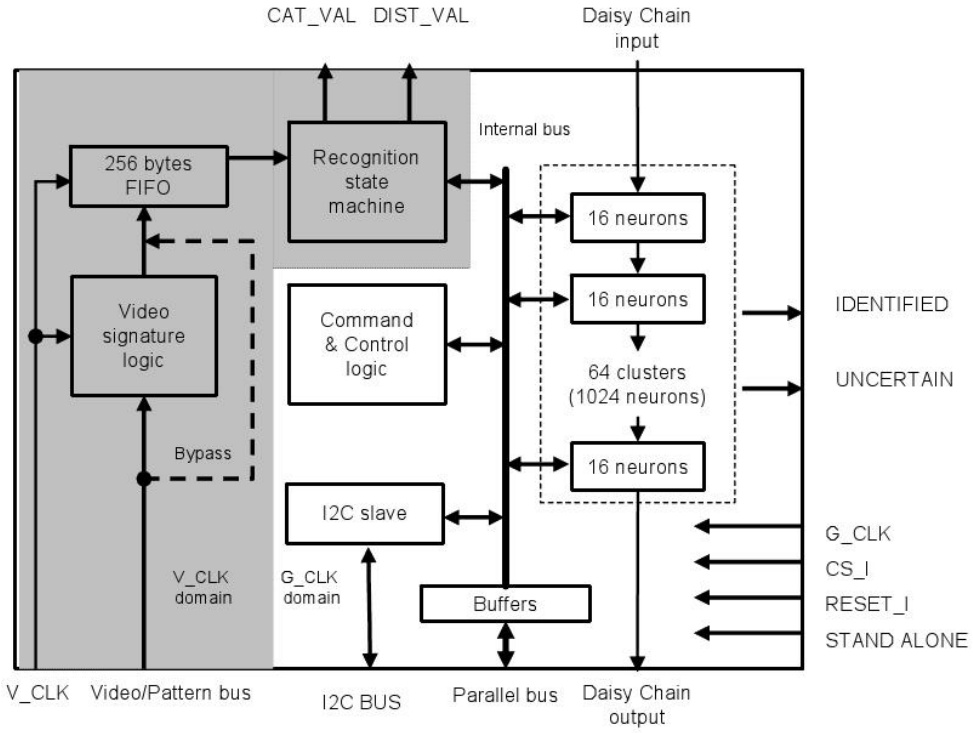


Figura 2.8: Esquema funcional do CogniMem [CogniMem 2008]

Capítulo 3

Arquitetura do Neuroprocessador

3.1 Requisitos

O neuroprocessador descrito no decorrer deste capítulo foi projetado como um módulo de propriedade intelectual em tecnologia ASIC, visando apresentar flexibilidade ao integrador de sistemas no que concerne ao compromisso entre desempenho e área. Este *hardware* foi desenvolvido para processar RNAs pré-treinadas, ou seja, sem suporte à aprendizagem.

Outro objetivo do projeto é suportar o reuso do sistema para processar mais de uma RNA com topologias distintas, modificando o número de camadas e neurônios em tempo de execução. Com esta característica, o neuroprocessador também pode ser implementado como um chip independente.

Para atender aos requisitos citados acima, têm sido adotadas as seguintes funcionalidades para o sistema:

- número ajustável de UPs;
- número ajustável de camadas processadas em paralelo;
- função de ativação implementada em Look-up Table (LUT);
- processamento em etapas (virtualização);
- tamanho ajustável da memória de pesos;
- tamanho ajustável da memória da LUT;
- precisões ajustáveis;
- topologia de RNA reconfigurável em tempo de execução;
- mudança dos pesos sinápticos e da LUT em tempo de execução para processar RNA diferentes.

Adicionalmente, foi incorporada como requisito a possibilidade de utilizar a comunicação com o ambiente externo palavra a palavra, de modo que o sistema seja independente da topologia de RNA e tenha um número factível de pinos de entrada. Isso também garante a compatibilidade com a maioria dos sistemas de processamento de sinais nos quais suas fontes de dados são conversores analógico-digital.

Com base nestes requisitos e com o exposto na seção 2.3, a arquitetura BBA foi utilizada como base para a concepção da arquitetura do neuroprocessador proposto. Ainda segundo a seção 2.3, o neurohardware aqui apresentado pode ser classificado como um

sistema do tipo neurochip (*IP core*), com tipo de partição por neurônio e camada, com paralelismo ajustável, representação numérica em ponto fixo e arquitetura BBA.

3.2 Processamento discreto das redes MLP

O cálculo para RNAs do tipo MLP foi apresentado no capítulo 2 através da equação 2.2. Para uma RNA de uma camada, podemos separar o argumento da função de ativação, como em 3.1 e 3.2.

$$u_k = \sum_{i=1}^{M} w_i^c \cdot x_j^{c-1} \tag{3.1}$$

$$y_k^c = \varphi(u_k + \Theta_k) \tag{3.2}$$

Dessa forma, o cálculo de 2.2 pode ser realizado em duas etapas, o produto vetorial e a função de ativação. Considerando uma implementação discreta, teremos primeiro multiplicações e um somatório com valores em ponto fixo. A precisão da representação dos dados e dos pesos vai inserir o primeiro estágio de erro. A operação por si só pode não inserir erro, caso haja bits suficientes na variável que armazena o resultado do somatório.

Entretanto, se tivermos a função de ativação implementada com uma LUT, teremos dois momentos em que ocorre perda de precisão. O primeiro é na entrada da função de ativação onde se tem um número limitado de dados armazenados, para não haver uma memória de armazenamento muito grande. Isto implica em truncar a saída do somatório, perdendo precisão. Como temos também uma precisão limitada nos valores armazenados na LUT, a saída da função de ativação vai inserir mais um erro, o de quantização.

A representação em ponto fixo utilizada para representar os dados, os pesos sinápticos, as entradas e as saídas foi o complemento de dois. Esta representação é normalmente adotada pelas ferramentas de EDA durante a interpretação do código HDL, pois resulta em menor hardware para operações aritméticas.

Para RNAs de duas ou mais camadas, as saídas das camadas intermediárias vão para as próximas camadas até chegar à saída. Isso significa que o ruído de quantização aumenta exponencialmente a cada camada, pois a cada vez que uma função de ativação é utilizada, duas novas fontes de erros aparecem no processamento.

Porém, nem sempre o erro final da RNA afeta o resultado, dependendo do tipo de classificação final adotado, como o *Winner Takes All* (WTA) [Prechelt 1994].

No projeto do neuroprocessador, cuidados foram tomados com o truncamento e com a discretização da função de ativação de modo a reduzir o erro final, conforme descrito no capítulo 4.

3.3 Visão Geral da Arquitetura

A flexibilidade do neuroprocessador proposto está relacionada com as suas características ajustáveis. Tais características podem ser divididas nos grupos das características parametrizáveis e das características configuráveis. As características parametrizáveis

estão relacionadas com a definição da quantidade de recurso utilizada em uma implementação do neuroprocessador. As características configuráveis definem como os recursos são utilizados.

A quantidade e a disposição das UPs, as precisões dos dados e pesos sinápticos, o tamanho da memória de pesos e o tamanho da LUT devem ser escolhidos durante a etapa de integração lógica do sistema, antes do mapeamento tecnológico. Estas são características parametrizáveis.

Por outro lado, no conjunto das características configuráveis estão as configurações relativas à topologia da RNA processada, como os valores dos pesos sinápticos e memória com a função de ativação, que devem ser ajustados após a implementação em uma tecnologia de circuito integrado, durante a utilização da rede. Estas e outras configurações serão detalhadas no decorrer do capítulo.

Durante a fase de parametrização do neuroprocessador, ainda no projeto lógico, dois modos de implementação podem ser escolhidos, dependendo dos requisitos da aplicação. Estes modos são:

- modo desempenho;
- modo área.

No modo Desempenho, como o nome sugere, o desempenho é priorizado, utilizando-se um número de UPs suficiente para processar todos os nós da rede da aplicação simultaneamente. No modo Área, visando reduzir a área, o número de UPs é inferior ao número de nós da rede, o que requer processamento seqüencial.

Em ambos os casos, todas as UPs são encerradas em um bloco principal, aqui chamado de *kernel* que, dependendo do modo, pode processar partes de uma rede ou toda a rede simultaneamente. O *kernel* é composto pelos sub-blocos Unidade de Controle, Módulo de Memória e Caminho de Dados. Uma vez que o processamento de RNAs requer um alto grau de paralelismo, o Caminho de Dados e o Módulo de Memórias têm suas capacidades distribuídas em pequenas unidades que funcionam paralelamente, provendo para cada nó da rede (um neurônio) uma unidade aritmética e o armazenamento dos pesos sinápticos referentes. O *kernel* do neuroprocessador é ilustrado na figura 3.1.

Utilizando a terminologia de descrição de *neurohardware* chamada Field Programmable Neural Array (FPNA), proposta por [Girau 2000], uma subunidade de memória e uma subunidade do caminho de dados podem ser vistos como um nó (neurônio) que realiza um produto vetorial completo e cujos recursos são disponibilizados para os outros nós da rede. Os nós, unidades de função de ativação e conexões virtuais (providos pela unidade de controle), todos juntos, consistem no conjunto completo de recursos necessário para a implementação de uma FPNA. Quando apropriadamente configurado o *hardware* torna-se uma Field Programmed Neural Network (FPNN) [Girau 2000], processando uma aplicação e topologias de RNAs específicas.

Porém, esta arquitetura não é completamente compatível com a definição de FPNA uma vez que a conectividade entre os nós é limitada às camadas consecutivas e os recursos do nó só são disponibilizados de maneira temporizada. Conseqüentemente, nem conexões recorrentes nem conexões paralelas são suportadas entre os neurônios.

Para implementação orientada ao desempenho, existem três possíveis estruturas que podem ser utilizadas, dependendo se o *hardware* deverá suportar no máximo uma, duas

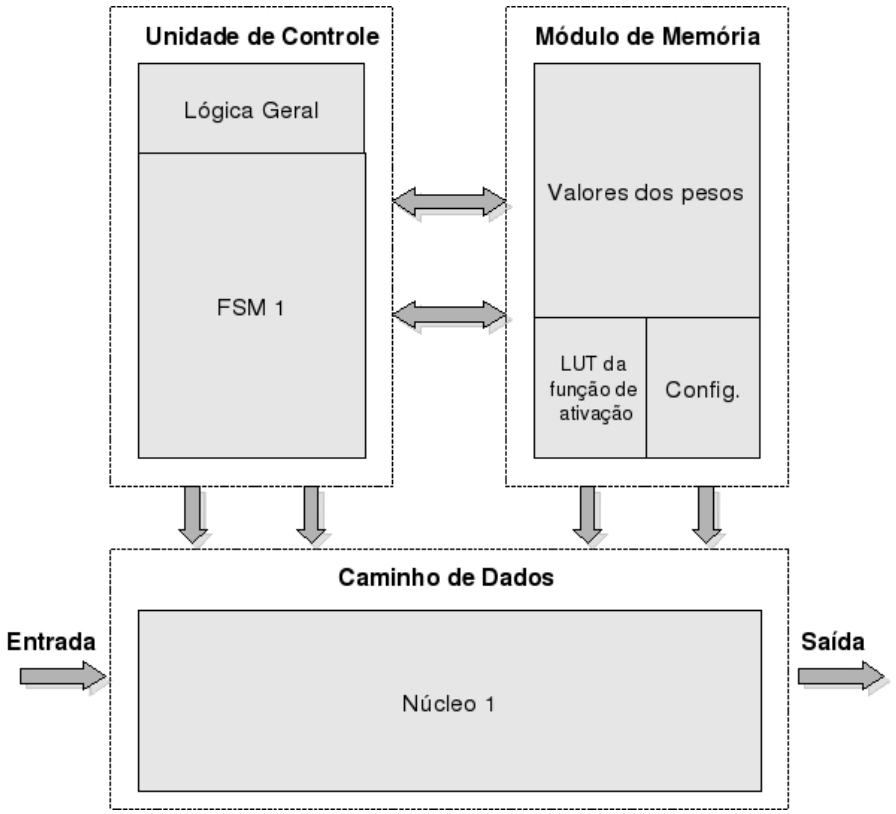


Figura 3.1: Kernel de núcleo simples

ou três camadas. Para RNAs com mais de três camadas, a arquitetura apresenta um baixo desempenho e, portanto, não foi implementada para tal. Topologias de mais de 3 camadas também não são muito encontradas na literatura. Para redes com uma camada, um *kernel* com apenas um núcleo é utilizado, como mostrado na figura 3.1. Neste são processados todos os nós da camada de saída de forma paralela.

Para redes de duas camadas, o *hardware* pode ser instanciado com uma estrutura de núcleo duplo. Dessa forma, cada camada é atribuída a um núcleo diferente, mantendo o processamento paralelo com relação às camadas e aos padrões de entrada. A camada oculta é processada pelo primeiro núcleo e a camada de saída pelo segundo. O *kernel* com núcleo duplo é ilustrado na figura 3.2. Após a implementação, as UPs não podem mais ser realocadas entre os núcleos.

Para redes de três camadas são utilizados dois *kernels*, de tal forma que um de núcleo duplo é concatenado a um de núcleo simples. As duas camadas ocultas são processadas no *kernel* de núcleo duplo e a camada de saída no *kernel* de núcleo simples. A figura 3.3 mostra o esquema para três camadas no modo desempenho.

Ainda com relação ao modo Desempenho, o número de UPs por núcleo tem que ser o suficiente para processar o máximo número de nós em cada camada nas topologias suportadas.

A unidade de controle do *kernel* é responsável por controlar o fluxo de dados internos.

Em casos de núcleo duplo, a unidade de controle também possui duas FSM, para controlar os núcleos, e uma lógica geral, que controla o fluxo de dados entres núcleos. Neuroprocessadores instanciados para três camadas suportam também duas e uma camada, assim como neuroprocessadores instanciados para duas processam topologias de uma camada.

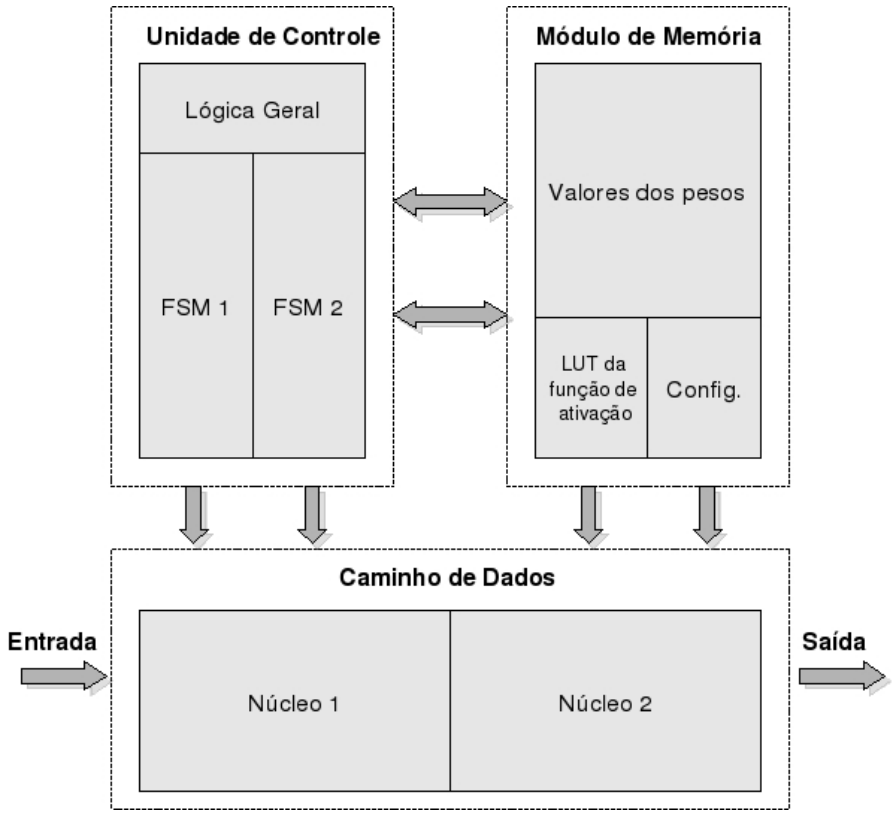


Figura 3.2: Kernel de núcleo duplo

No caso de projetos que visam otimizar a área, o neuroprocessador pode ser implementado com apenas um *kernel* de núcleo simples e um número pequeno de UPs, para qualquer número de camadas e de neurônios. Nesta implementação, o número máximo de neurônios em uma camada é geralmente maior que o número de UPs disponíveis, requerendo virtualização para processar seqüencialmente as operações relacionadas a cada camada. Além do *kernel*, algumas memórias e uma máquina de estados também são utilizadas para fazer o processamento seqüencial. A arquitetura e o funcionamento do uso da virtualização para o processamento seqüencial são elucidados na secção 3.6.

3.4 Protocolo de Comunicação

Um protocolo de comunicação de dados único está sendo utilizado em diversos níveis hierárquicos do sistema. As UPs, os núcleos dos caminhos de dados, os *kernels* e mesmo o neuroprocessador utilizam este protocolo para receber e enviar dados.

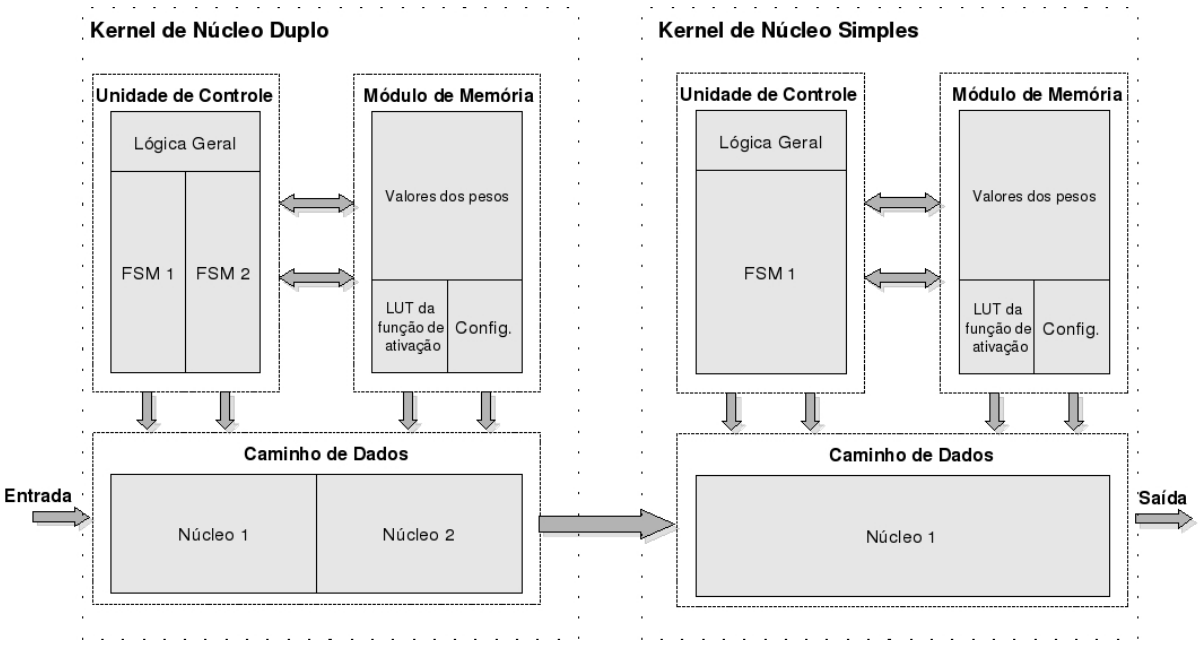


Figura 3.3: Esquema para três camadas no modo desempenho

Este protocolo é uma versão simplificada do protocolo utilizado pela Altera Corp. em diversos dos seus blocos de propriedade intelectual, o Avalon Streaming Interface [Altera 2009].

Todos os módulos envolvidos na comunicação devem possuir duas portas, uma de entrada e outra de saída. A comunicação sempre é feita ponto a ponto, com a porta de saída de um módulo conectada à porta de entrada do módulo seguinte. Os dados são transmitidos quadro a quadro, em que cada quadro pode ter tamanho variável. A figura 3.4 ilustra a conexão entre dois blocos na comunicação.

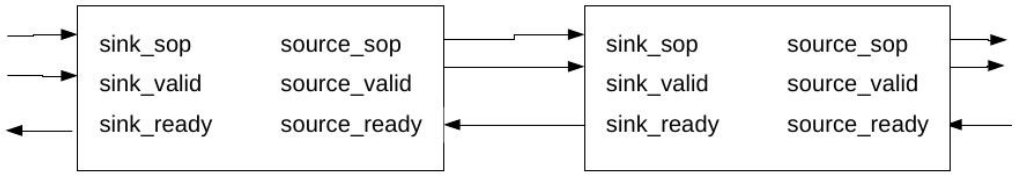


Figura 3.4: Conexão entre blocos comunicantes

Três situações são possíveis durante uma comunicação:

- início do quadro de dados;
- trasmissão do dado;
- dispositivo receptor pronto para receber.

O início de um novo quadro de dados é sempre marcado com um pulso do sinal 'sop'

(*start of operation*). Quando um dado está sendo transmitido, o sinal '*valid*' permanece alto. O final de um quadro de dados é marcado pela descida do sinal '*valid*' ou pelo começo de um novo quadro. O sinal '*ready*' alto indica que o receptor está pronto para receber. O prefixo '*sink*' indica que é uma porta de entrada, enquanto que o prefixo '*source*' indica uma porta de saída. A tabela 3.1 descreve todos os sinais e a figura 3.5 ilustra duas situações de transição.

Tabela 3.1: Sinais no protocolo de comunicação

Nome	Direção	Largura	Descrição
sink_sop	entrada	1	Indica o início de um quadro de dados chegando
sink_valid	entrada	1	Quando alto, indica que tem um dado sendo recebido
sink_ready	saída	1	Indica para o bloco anterior que está pronto para receber
source_sop	saída	1	Indica o início de um quadro de dados saindo
source_valid	saída	1	Quando alto, indica que tem um dado sendo enviado
source_ready	entrada	1	O próximo bloco está indicando que está pronto para receber

O neuroprocessador comunica-se com o ambiente externo através desse protocolo, assim como o(s) *kernel*(s). A unidade de controle manipula os dados que entram e saem dos núcleos com essa sinalização. Os sinais de comunicação provenientes da unidade de controle, que entram nos núcleos, seguem até as UPs, que também utilizam o protocolo. O fato de usar um protocolo único facilita a integração de novos blocos e a conexão entre eles.

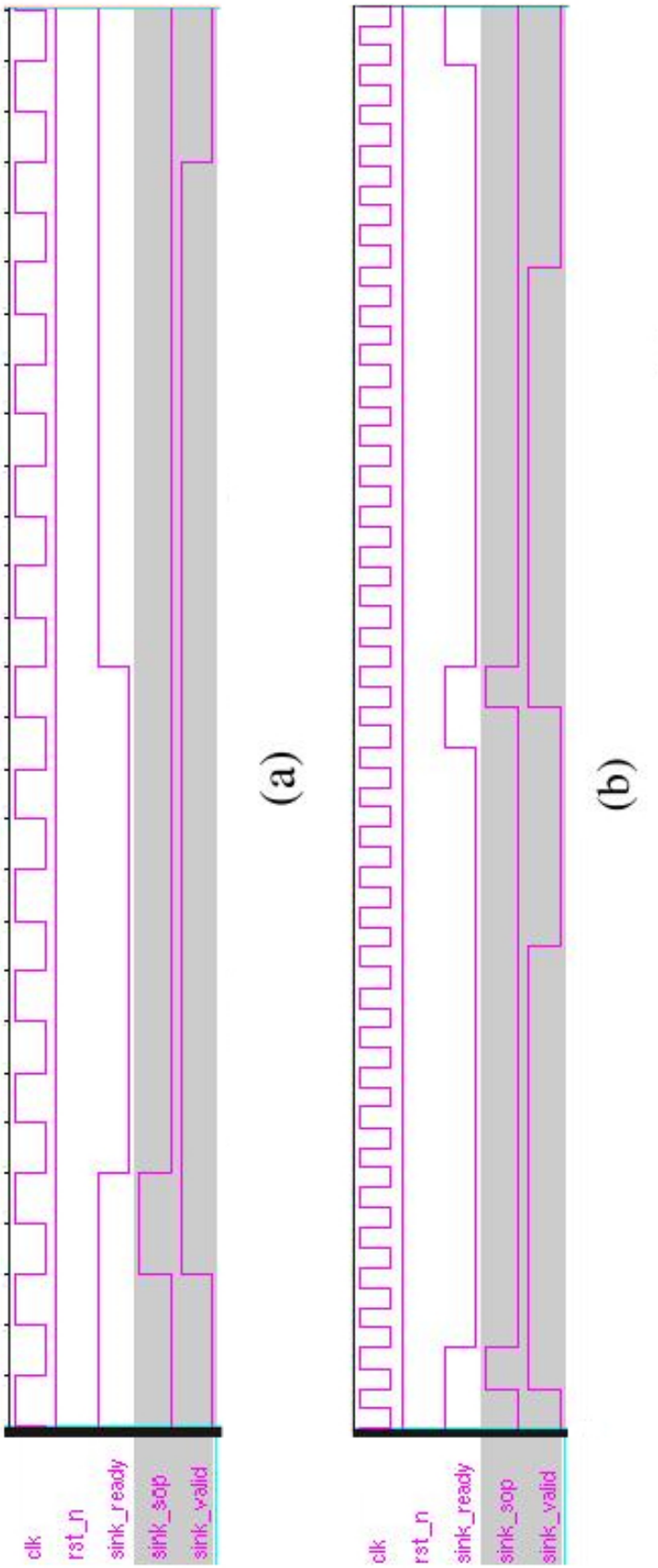


Figura 3.5: (a) Transmissão simples (b) Dispositivo de recepção ocupado

3.5 Descrição do Kernel

3.5.1 Caminho de Dados

O processamento matemático dos dados de uma RNA acontece no Caminho de Dados. Este concentra todas as UPs e as funções de ativação, que fazem acesso às memórias contidas no bloco de memórias para acessar os pesos sinápticos e a LUT, respectivamente. Como mencionado anteriormente, o caminho de dados pode ter um ou dois núcleos, dependendo do modo de implementação utilizado.

Para a implementação com um núcleo, o sistema pode processar apenas uma camada por vez. O número de UPs no caminho de dados do núcleo determina o número máximo de neurônios por camada suportado. O núcleo pode ser utilizado para processar camadas com um número menor de neurônios que o de UPs. Porém, nesses casos, não há reaproveitamento das unidades em excesso.

A arquitetura do caminho de dados de núcleo simples, ilustrado na figura 3.6, pode ser classificado como BBA, baseando-se na forma que as UPs recebem e entregam os dados. Cada entrada de um novo padrão vai, de um em um, para todas as UPs do núcleo. Estas UPs computam operações de multiplicar-acumular em paralelo, a cada dado que chega, da mesma forma que na arquitetura BBA. No entanto, diferentemente da BBA apresentado em [Ienne 1993], somente uma LUT tem sido usada para todas as UPs do núcleo. Esta contenção de recursos é uma limitação para o processamento, resultando em um desempenho pior em relação à BBA, mas com o benefício de diminuir significantemente a área.

Uma vez que os dados do padrão de entrada chegam um a um, a limitação só acontece quando o número de entradas é menor que o número de nós na camada de saída. Isso indica que, em algumas situações, a restrição de recursos pode ter somente a redução da área, sem prejudicar o desempenho.

Junto com cada dado de entrada também chega o peso correspondente, sendo estes multiplicados e acumulados em um registrador dentro da UP. Os pesos para as UPs chegam através de um barramento único que liga todas as UPs ao módulo de memórias. Existe um seletor de memória relacionado a cada UP que escolhe os fios corretos no barramento. Este barramento pode dar acesso às diversas posições de memória como será discutido em mais detalhes na seção 3.5.4.

Depois que todos os dados de entrada foram processados, os resultados da operação de multiplicar e acumular de cada UP são enviados um a um para a função de ativação e, em seguida, vão para a saída. Enquanto os dados estão aguardando para serem processados pela função de ativação, outro padrão de entrada pode começar a ser processado visto que os resultados calculados pelos UPs são armazenados em registradores temporários. Todo o processo é controlado por uma FSM presente na Unidade de Controle.

Se o neuroprocessador tiver sido implementado com *kernel* de núcleo duplo, o caminho de dados tem sua estrutura duplicada para manter o desempenho similar ao de núcleo simples, no caso de estar configurado para redes de duas camadas. Os dois núcleos do caminho de dados são conectados por meio de uma memória do tipo First in First Out (FIFO). Os dados processados pelo primeiro núcleo passam pela FIFO e vão para o segundo núcleo. A FIFO é utilizada com o intuito de reduzir a limitação na comunicação

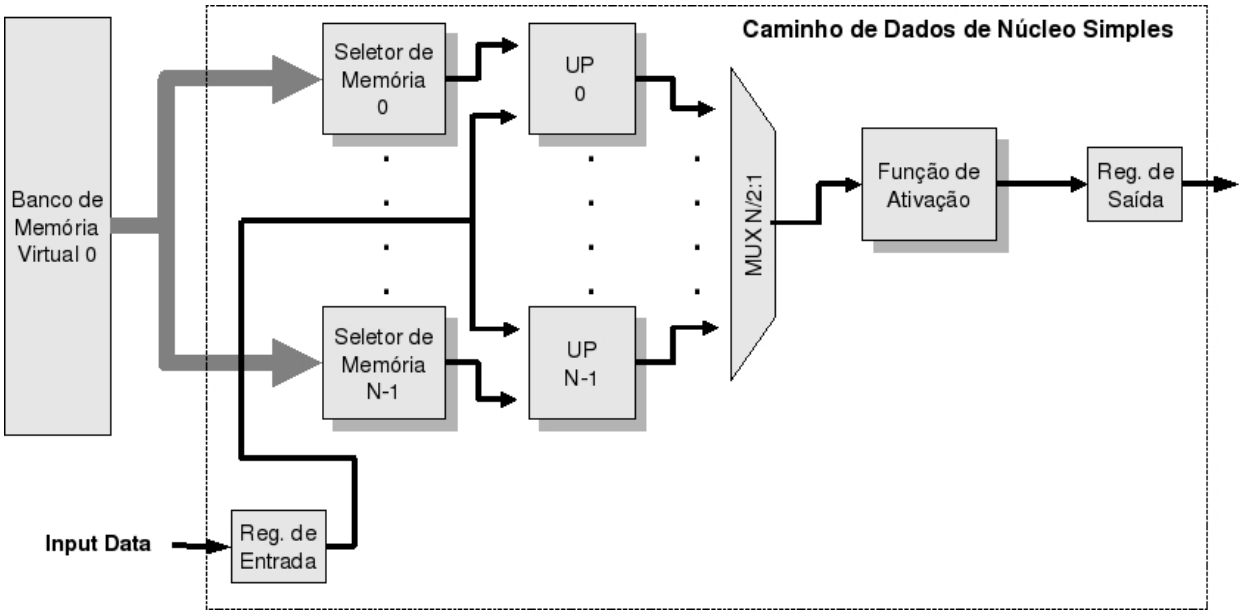


Figura 3.6: Caminho de Dados de Núcleo Simples

entre os núcleos. A figura 3.7 mostra como os núcleos são conectados.

O funcionamento individual de cada núcleo é exatamente o mesmo que o descrito acima para o caminho de dados de núcleo simples. Porém, cada camada é processada em um dos núcleos onde os dados da camada oculta são processados e enviados para o outro núcleo, que processa a camada de saída e envia os resultados para a saída. Se o neuroprocessador for configurado para uma rede de uma camada, apenas o primeiro núcleo é utilizado e sua saída é multiplexada para a saída do sistema.

As duas funções de ativação, apesar de serem usadas independentemente, acessam a mesma LUT, também visando economizar área. Como a LUT está numa memória RAM de porta dupla, os núcleos podem usar a função paralelamente sem problemas de conflito de recursos. Essa escolha apenas restringe o sistema a um tipo de função de ativação via LUT.

A função de ativação também pode ser configurada para executar duas funções que não utilizam a LUT, a função degrau (*threshold*) e a função linear. Se o caminho de dados for de núcleo duplo, duas funções diferentes podem ser usadas em cada núcleo. Para núcleo simples, qualquer um dos três tipos de função pode ser utilizado.

No caso de implementações para processar três camadas, dois *kernels* em cascata são utilizados (ver Fig. 3.3). A conexão entre estes dois *kernels* é feita diretamente, sem o uso de FIFOs. Isto não diminui o desempenho contanto que o número de nós na camada de saída seja menor que o da segunda camada oculta. A comunicação entre os dois *kernels* é feita usando o protocolo descrito em 3.4.

Neste caso de dois *kernels*, como estes são independentes, pode haver mais de uma LUT, sendo que uma usada no primeiro *kernel* e a outra no segundo. Obviamente, a área aumenta de forma significativa.

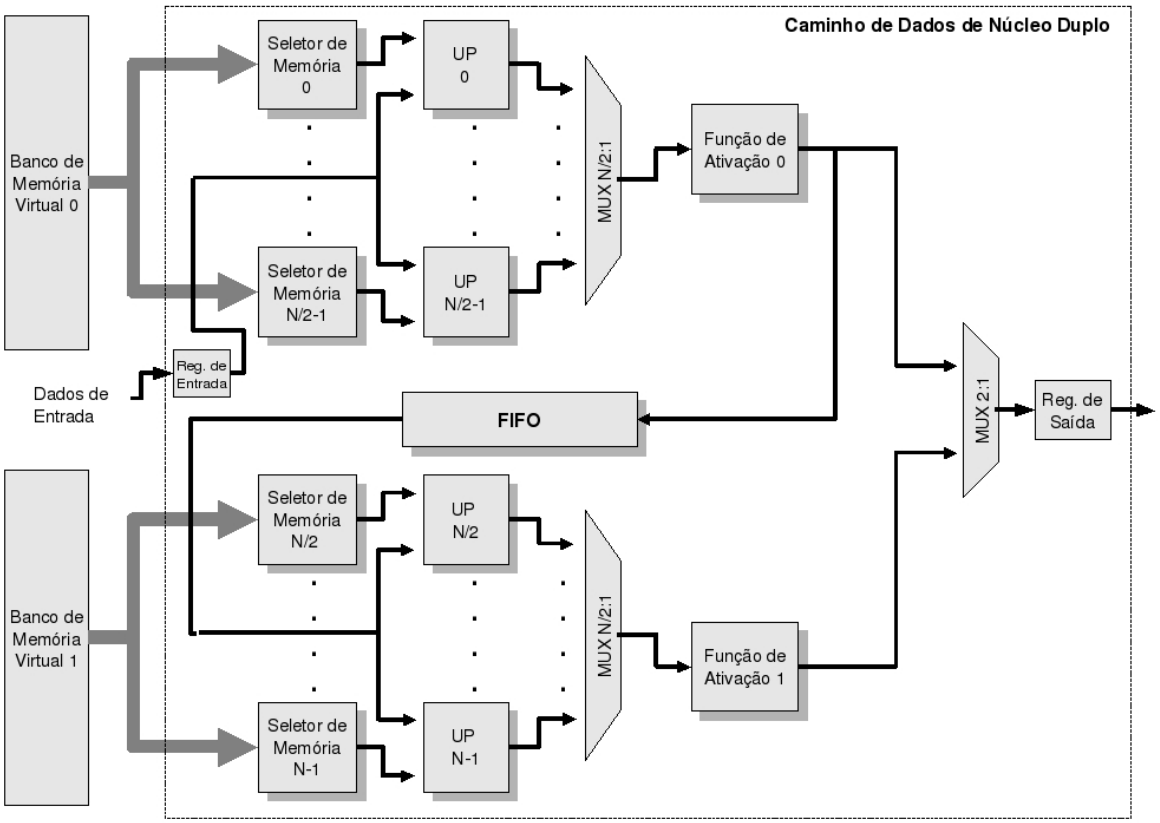


Figura 3.7: Caminho de Dados de Núcleo Duplo

3.5.2 Unidade de Processamento

O somatório de produtos descrito na equação 3.1 é a operação mais crítica que afeta o desempenho do sistema. De fato, as principais métricas de desempenhos podem ser medidas indiretamente pela quantidade e velocidade de processamento paralelo dos neurônios. Esta operação é realizada por unidades do tipo multiplica-acumula (MAC) a cada novo dado que chega.

A figura 3.8 mostra as etapas envolvidas no processamento de um nó da rede. Cada dado novo que chega é multiplicado por um peso que chega ao mesmo tempo. Em seguida, o resultado da multiplicação é somado ao registrador acumulador. O primeiro dado a chegar é a polarização, que é multiplicada por um ao mesmo tempo em que ocorre uma desativação da soma do acumulador. Assim, o valor da polarização é o primeiro a ser carregado a cada novo nó processado. Esta operação determina a máxima freqüência de trabalho do sistema.

Os multiplicadores e somadores utilizados para compor a UP tiveram suas arquiteturas específicas selecionadas pelo sintetizador. Este escolhe as arquiteturas de cada operação

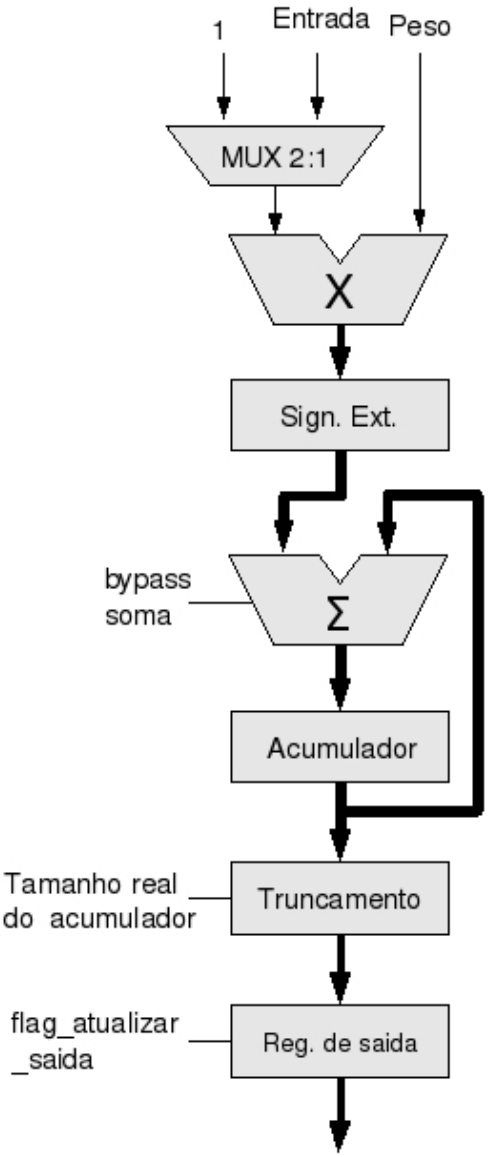


Figura 3.8: Diagrama da Unidade de Processamento

aritmética baseando-se em diversas variáveis de otimização do sistema como um todo, fazendo uma escolha melhor que a manual para a maioria dos casos.

Ao fim da recepção dos dados da camada anterior, o resultado da operação no acumulador é truncado e armazenado no registrador de saída. A informação permanece nesse registrador enquanto não poder ser enviada para a função de ativação, permitindo que a unidade já possa começar a receber dados referentes a outro padrão de entrada.

Em geral, o bloco da função de ativação tem uma entrada com largura menor que o tamanho do acumulador, devido à limitação de precisão (LUT pequena). O truncamento faz-se então necessário antes do dado ser enviado adiante.

A precisão durante o cálculo é assegurada por meio de bits de guarda que expandem a precisão do acumulador para suportar um número de adições igual ao número máximo de entradas em uma UP. Segundo [Omondi e Rajapakse 2006], o tamanho do acumulador é determinado a partir da equação:

$$acc\_width = log_2(n\_predecessors) + data\_width + weight\_width - 1 \quad (3.3)$$

Onde data_width e weight_width são as resoluções do dado e do peso, respectivamente. n_predecessors é o número de nós predecessores, que é igual à quantidade de dados de entrada na UP.

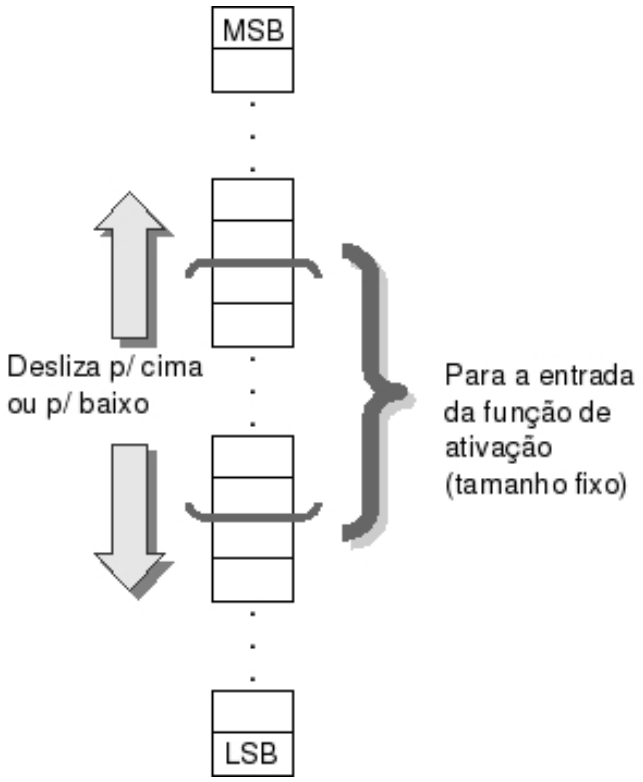


Figura 3.9: Esquema de truncamento na UP

Se o neuroprocessador for destinado a processar apenas uma topologia de RNA, o número de predecessores será fixo para cada um dos núcleos. Entretanto, caso haja reuso do *kernel* através de virtualização ou com topologias diferentes, o número de predecessores em cada núcleo dependerá da topologia que está sendo processada. A escolha mais coerente é considerar o maior número de operações que podem ser realizadas conforme as topologias que vão ser configuradas no neuroprocessador. Assim, sempre haverá bits suficientes para o processamento das UPs em todas as situações planejadas para o *hardware*.

Porém, o truncamento mencionado anteriormente não pode consistir somente de des-

cartar alguns bits menos significativos (LSB), porque para camadas diferentes e topologias diferentes temos um número de nós predecessores diferente. Isto significa que o bit mais significativo (MSB) para uma camada não é necessariamente o MSB da largura total do acumulador. Sendo assim, o truncamento deve descartar alguns LSB e alguns MSB mantendo uma saída fixa, igual à largura da entrada da função de ativação. A saída final do MAC desliza dentro da faixa de largura do acumulador, como ilustrado na figura 3.9. A posição é escolhida usando a largura real do acumulador para o número de predecessores da camada. Esta largura é computada no caminho de dados, realizando o mesmo cálculo da equação 3.3.

Para uma mesma configuração de topologia a faixa de bits deve se manter fixa para não inserir ganhos nos resultados de algumas camadas com relação às outras. Então, dentro de uma mesma topologia, o número de predecessores utilizado para o cálculo da largura efetiva do acumulador é o da camada com o maior número de entradas cujo valor é definido *off-line* e configurado no *hardware*.

3.5.3 Função de Ativação

Neste neuroprocessador, a função de ativação pode ser configurada para três tipos de operações diferentes:

- Função Linear;
- Função Degrau;
- Tabela de Look-up (LUT).

Para topologias de RNA lineares, nas quais as camadas de saídas são funções lineares, ou seja, simplesmente a soma ponderada dos valores de entrada, a função linear deve ser utilizada. A única operação realizada para esta função é um novo truncamento dos dados para deixá-los do mesmo tamanho da palavra do sistema.

A função degrau, que é muito utilizada em diversas aplicações, funciona colocando na saída o valor máximo ou mínimo se a entrada for positiva ou negativa, respectivamente. O valor mínimo pode ser configurado para zero ou o valor mais negativo representado na palavra do sistema.

Devido à flexibilidade, o uso da função de ativação através de uma tabela de look-up é a solução mais atraente e necessária para a maioria das aplicações.

A LUT tem que ser carregada na memória com uma aproximação suficientemente precisa da função não-linear escolhida. No conjunto das funções comumente usadas para redes MLP estão sigmoid, tangente hiperbólica, arco-tangente e rampa.

Para discretizar e armazenar em *hardware* uma função de ativação não-linear surgem algumas questões com relação à precisão.

A metodologia de discretização de funções descrita em [Omondi e Rajapakse 2006] têm sido adotada. De acordo com esta metodologia, somente um número conveniente n_s de MSB dentre o total de N_S bits da soma ponderada discreta deve ser usado como entrada da função de ativação discreta. Em [Omondi e Rajapakse 2006], o número ótimo n_{sopt} de bits de entrada é estimado em termos da derivada da função no ponto zero f_0', o máximo de conexões de entrada em um neurônio, o valor máximo absoluto M_w de pesos contínuos, o número n_z de bits de entrada para o perceptron discreto e N_S.

A expressão para calcular n_{sopt} [Omondi e Rajapakse 2006] é repetida aqui por conveniência:

$$n_{sopt} = \left\lceil log_2 \left[f_0' \#w M_w \frac{2^{N_S}(2^{n_z}-2)}{2^{N_S}-2} \right] \right\rceil \tag{3.4}$$

Uma vez que o número de conexões de entrada para um neurônio pode variar consideravelmente entre diferentes configurações de topologias de RNA, n_s não pode ser o valor ótimo para todas as situações. Um n_{sopt} pode ser encontrado para o pior caso em que o número de predecessores é o maior. Por exemplo, se adotarmos uma função sigmoide padrão, $\#w = 511$, $M_w = 1$, $N_S = 40$, $n_z = 16$ e $f_0' = 1/4$, a partir de 3.4, n_{sopt} será 23.

Além disso, ainda de acordo com [Omondi e Rajapakse 2006], uma vez que os valores da função de ativação discreta, em sua maioria¸ são iguais aos seus valores mínimo ou máximo, a LUT precisa armazenar apenas os valores correspondentes ao intervalo central de entrada, onde a variação é significativa. [Omondi e Rajapakse 2006] Também apresenta expressões para calcular os limites do intervalo central i_{min} e i_{max}, em termos dos parâmetros da função de ativação adotada. Portanto, o número de bits necessários para especificar a LUT é dado por [Omondi e Rajapakse 2006]:

$$n_{LUT} = \lceil log_2 (i_{max} - i_{min} + 1) \rceil \tag{3.5}$$

De fato, a LUT é aplicada somente aos valores da soma discreta ponderada dentro do intervalo i_{min} e i_{max}. Estes parâmetros têm que ser configurados para cada tipo de função adotada.

Como n_{LUT} é diretamente proporcional a ns através das expressões de i_{min} e i_{max}, o tamanho da memória da LUT pode ser diminuído reduzindo-se o valor de n_s para um valor abaixo de n_{sopt}, o que obviamente reduzirá a precisão da função.

Neste projeto, a LUT é implementada utilizando memória SRAM de porta dupla para que possa ser usada por dois núcleos simultaneamente. Memória de tecnologia *flash* também poderia ser usada para eliminar a necessidade de reescrita a cada vez que o sistema é reiniciado.

3.5.4 Arquitetura de Memória

Durante a operação do sistema, cada UP precisa acessar um valor de peso a cada pulso de clock, independente das outras UPs. Isto significa que o armazenamento dos pesos precisa ser feito em pequenas unidades, de forma que as UPs possam acessar os seus pesos em paralelo. A arquitetura de memória também deve permitir que os dois núcleos acessem todos os valores de pesos armazenados a qualquer hora. Além disso, quando o neuroprocessador é utilizado para mais de uma topologia de RNA, a arquitetura deve permitir que o conjunto de pesos seja modificado a cada transição de configurações de topologia.

Essas pequenas unidades de memória, aqui chamadas de blocos, estão sendo implementadas com memórias SRAM síncrona de duas portas, atendendo dois dos requisitos citados acima. Memórias de duas portas permitem leituras de endereços diferentes em paralelo durante um único ciclo.

Entretanto, se tivermos um bloco associado a cada UP e diversas topologias utilizadas no mesmo *hardware*, o tamanho dos blocos pode ser consideravelmente grande. Para projetar os circuitos integrados de memória na tecnologia CMOS, geradores automáticos de memória são amplamente utilizados, pois apresentam melhores resultados com relação ao alto nível de densidade, à velocidade e ao consumo. As informações com relação às memórias geradas vêm em forma de um *hard block*, para ser usado no *layout*, e um modelo em uma Linguagem de Descrição de *Hardware* (HDL) para simulação. Este *hard block* tem uma forma definida pelo gerador, o que pode ocasionar problemas em casos de um número elevado de blocos grandes, reduzindo drasticamente a flexibilidade de definição do formato do *layout*. A ocorrência de grandes e numerosas memórias também pode ser problemática para implementação em outras tecnologias, como na prototipação em FPGA.

Visando reduzir o tamanho dos blocos, estes foram agrupados em páginas. Cada página contém um número de blocos menor ou igual ao número de UPs. Um núcleo pode apontar para uma página por vez. No entanto, os dados dos neurônios podem estar tanto concentrados em um único bloco como espalhados em alguns blocos em páginas diferentes. A organização de blocos e páginas está ilustrada na figura 3.10, contendo uma arquitetura com H blocos de M palavras e K páginas de N blocos cada uma.

Essa arquitetura com divisão por páginas provê escalabilidade para o *hardware* com relação às memórias. Também possibilita um bom aproveitamento das memórias, associando páginas a núcleos. Assim, as páginas podem ter a quantidade de blocos igual à quantidade de UPs do núcleo.

A unidade de controle faz o gerenciamento de qual posição é lida pelas UPs a cada instante. A cada dado que chega no *kernel* (ou no núcleo), todas as UPs precisam acessar o peso associado àquela entrada simultaneamente. Para uma página selecionada, um único endereço é usado para acessar todos os pesos referentes às UPs de um núcleo. Por exemplo, se a página 1 e o endereço 64 estiverem selecionados, todas as UPs do núcleo acessarão a posição 64 de memória do bloco correspondente, dentro da página 1. Apesar desta organização, a unidade de controle trata a memória como contínua, simplesmente usando um endereço global composto pelos endereços da página (MSB) e interno do bloco (LSB).

A transferência dos dados entre o módulo de memória e os núcleos acontece por meio de barramentos onde são concatenados todos os dados que vão para um mesmo núcleo. Este barramento, chamado de banco virtual, conecta um núcleo com uma página, dando acesso a todos os seus blocos. A implementação em barramento, ao invés de matricial, foi escolhida por conta de limitações da linguagem Verilog em utilizar vetores bidimensionais como entrada ou saída dos módulos. Figura 3.11 ilustra a atribuição dos dados no banco virtual.

Quando o sistema é utilizado para processar mais de uma RNA, quando há virtualização ou mesmo para estrutura de núcleo duplo, a memória deve ser dividida para alocar os pesos relativos a cada camada ou parte de camada para cada RNA. Nestes casos, os dois núcleos podem precisar endereçar os mesmos blocos em regiões diferentes, o que exige o uso de memórias de duas portas. Um exemplo de divisão é mostrado na figura 3.12. Uma configuração é guardada para informar à unidade de controle o começo e o comprimento

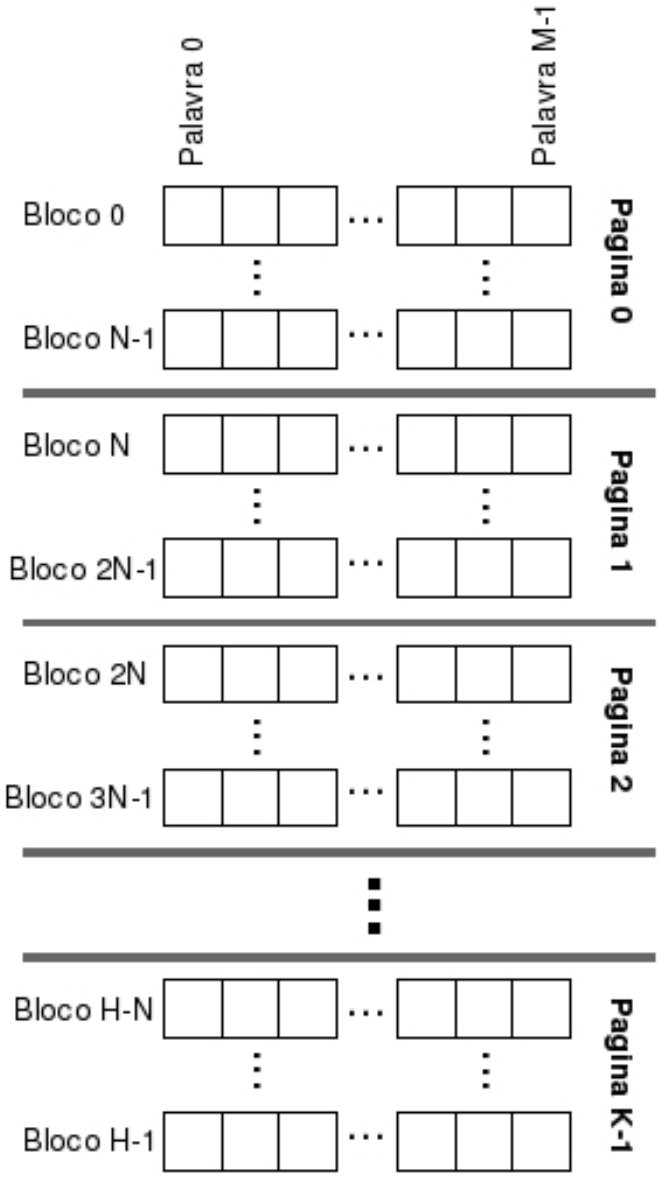


Figura 3.10: Arquitetura da memória de pesos

de cada trecho.

Todas as memórias do sistema têm sido implementadas utilizando a tecnologia SRAM, onde cada bloco da figura 3.10 é um módulo SRAM de duas portas. Sendo assim, elas precisam ser escritas antes do sistema começar a ser usado. O *kernel* disponibiliza duas portas para endereçar os dados, uma para acessar o bloco e outra para acessar uma posição dentro do bloco. Do ponto de vista da escrita externa, a LUT e uma memória de configurações são tratadas como blocos e podem ser escritas utilizando as mesmas portas,

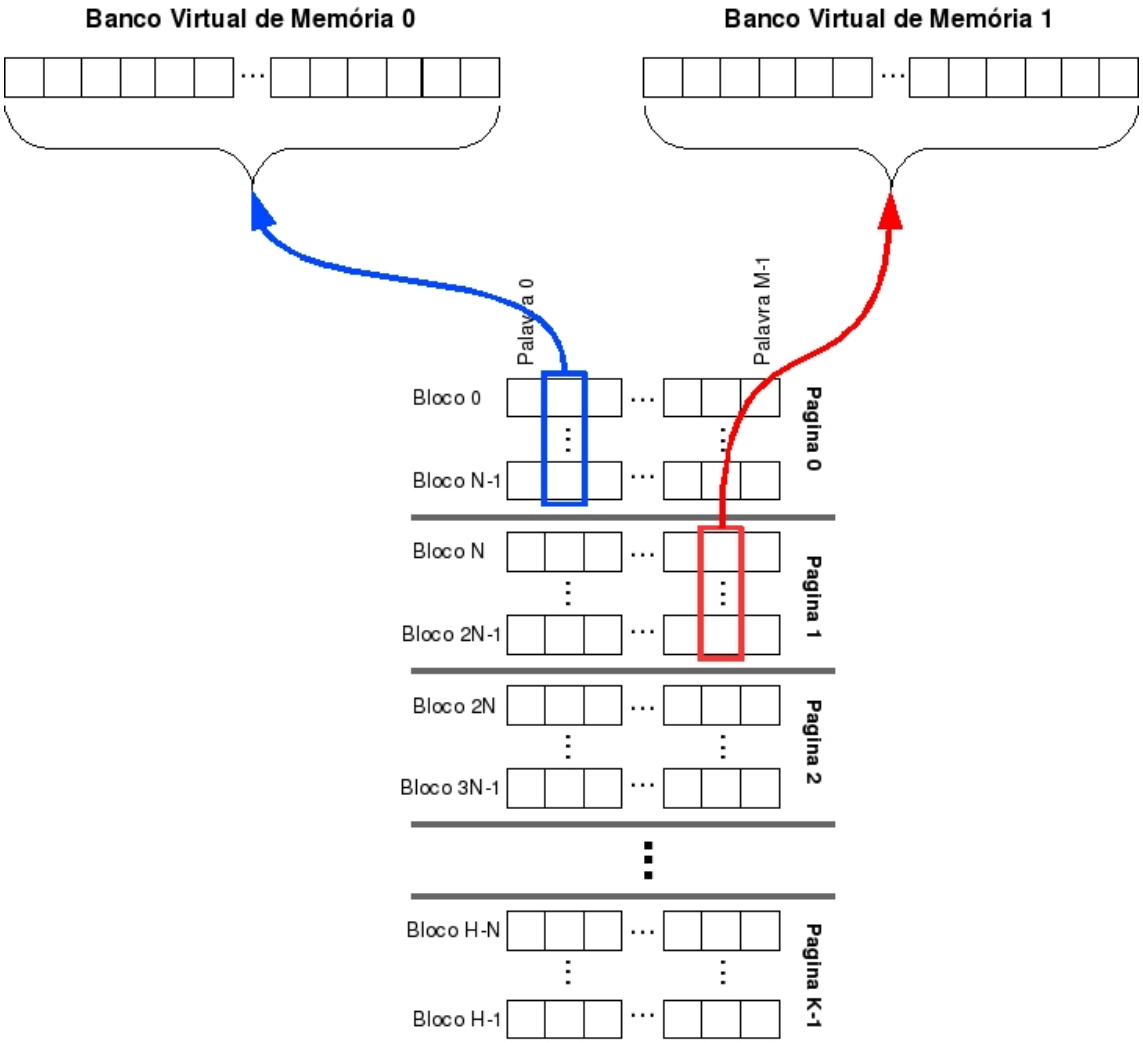


Figura 3.11: Atribuição de dados ao banco virtual

como mostrado na figura 3.13.

A memória de configurações armazena valores usados pela Unidade de Controle e pelo caminho de dados durante o processamento. A tabela 3.2 mostra a lista dos registradores de configuração. Note que existem vários pares dos registradores $cfg\_nodes\_virtual < xx >$ e $cfg\_start\_virtual < xx >$. Essas são várias opções de configuração visando dar suporte à virtualização, onde cada par pode estar relacionado com uma camada, pedaço de camada ou uma topologia de RNA. Todos os registradores de configuração possuem 16 bits. Para mudar a configuração para outra camada, pedaço de camada ou RNA, o registrador cfg_layers deve ser modificado, conforme indicado na tabela 3.2.

Dessa forma, a reconfiguração do *kernel* para processar camadas ou pedaços de camadas diferentes acontece em apenas um ciclo de relógio, bastando mudar o valor das configurações $cfg\_virtual\_conf < x >$. Por exemplo, o par $cfg_nodes_virtual$00 e

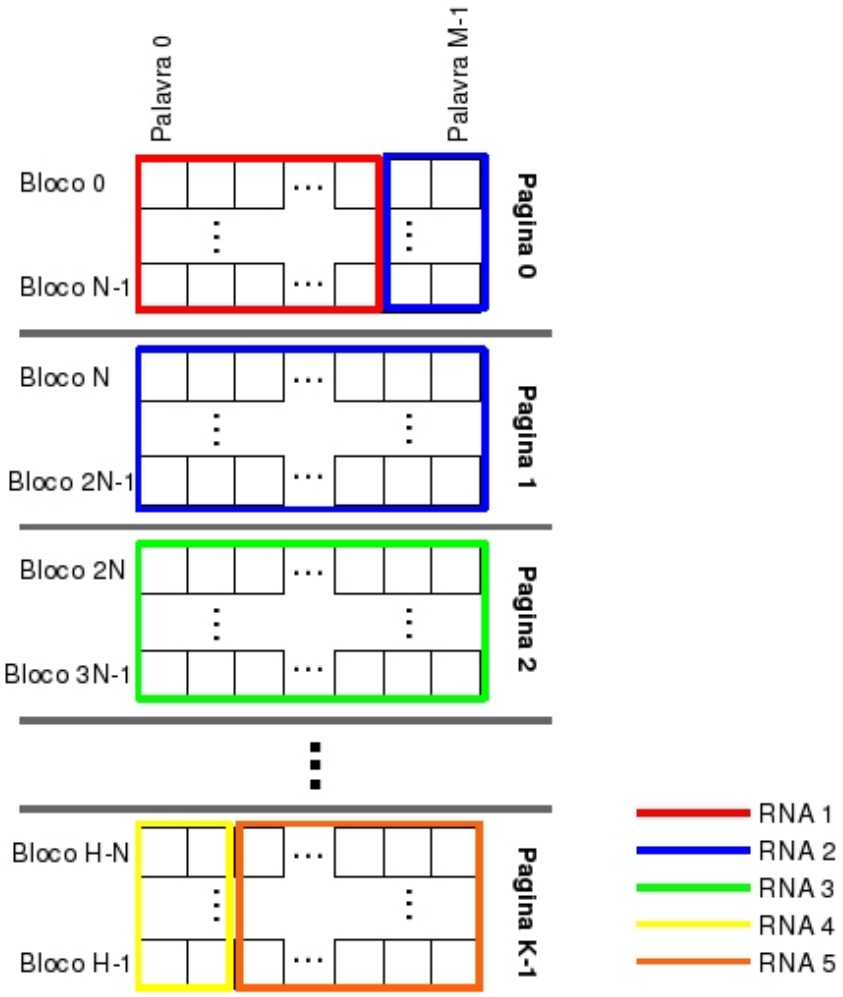


Figura 3.12: Exemplo de configuração para diversas topologias de RNA

$cfg\_start\_virtual00$ podem ser as configurações da primeira metade de uma camada e o par $cfg\_nodes\_virtual01$ e $cfg\_start\_virtual01$ a configuração da segunda metade.

Os registradores de configuração $cfg\_nodes\_virtual < xx >$ possuem os seguintes campos:

- bit 0-9 : Numero de nós
- bit 10: faixa da função de ativação. 0 - [0;1] / 1 [-1;1]
- bit 11: Reservado
- bit 12-13: Tipo de função de ativação. 00 - LUT; 01 - *bypass* (linear); 10 - *Threshold*; 11 – Reservado
- bit 14-15: Indica a camada sendo processada. 00 - entrada; 01 - oculta1; 10 - oculta2; 11 - saída

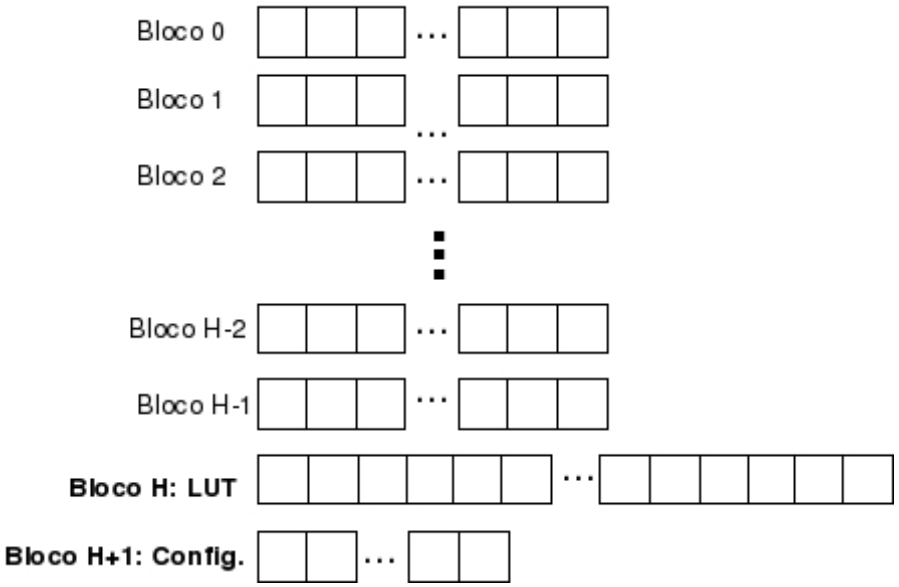


Figura 3.13: Arquitetura de memória incluindo LUT e configuração

3.5.5 Unidade de Controle

O modo como os dados fluem na cadeia de processamento do sistema faz com que não haja muita necessidade de *hardware* de controle. As atribuições da unidade de controle são:

- geração dos sinais do protocolo que vão para as UPs;
- gerenciamento das limitações entre UPs e a função de ativação;
- atribuição dos endereços da página e da posição no bloco de memória;
- multiplexação dos dados das UPs para funções de ativação.

Essas tarefas, já descritas anteriormente, são realizadas utilizando-se uma ou duas FSM, sendo que cada FSM está associada a um núcleo do caminho de dados. Cada FSM funciona de forma independente, não importando se o *kernel* está configurado para núcleo simples, núcleo duplo ou virtualização. Figura 3.14 ilustra o diagrama de estados da FSM.

A sinalização é feita com base nos sinais que chegam ao *kernel* e a disponibilidade do bloco seguinte. Funcionando de acordo com o protocolo utilizado no sistema. Em casos de núcleo duplo, a FIFO entre os núcleos facilita o trabalho do controle.

3.6 Virtualização

A operação de virtualização consiste em realizar o cálculo computacional da RNA inteira em algumas etapas, reusando o *kernel* para calcular camada por camada ou mesmo quebrando camadas em pequenos pedaços. O número de neurônios desses pedaços de camadas deve ser menor ou igual ao número de UPs disponíveis.

Tabela 3.2: Registradores de configuração

End.	Nome	Descrição
0	cfg_layers	bit 0-2 - Número de camadas bit 3 – reservado bit 4-9 - cfg_virtual_conf0 – Configuração virtual para o núcleo 0 bit 10-15 - cfg_virtual_conf1– Configuração virtual para o núcleo 1
1	cfg_actv_i_min	Valor mínimo de entrada na função de ativação onde a LUT precisa ser usada
2	cfg_actv_i_max	Valor máximo de entrada na função de ativação onde a LUT precisa ser usada
3	cfg_n_pred_lut	Número de predecessores para ser usado no truncamento das UPs
4	reservado	
5	cfg_nodes_virtual00	Config. 00 para os nós de um núcleo
6	cfg_start_virtual00	Posição de início dos pesos para conf. 00
7	cfg_nodes_virtual01	Config. 01 para os nós de um núcleo
8	cfg_start_virtual01	Posição de início dos pesos para conf. 01
9	cfg_nodes_virtual02	Config. 02 para os nós de um núcleo
10	cfg_start_virtual02	Posição de início dos pesos para conf. 02
11	cfg_nodes_virtual03	Config. 03 para os nós de um núcleo
12	cfg_start_virtual03	Posição de início dos pesos para conf. 03
13	cfg_nodes_virtual04	Config. 04 para os nós de um núcleo
14	cfg_start_virtual04	Posição de início dos pesos para conf. 04

Uma lógica de controle extra, uma FIFO, um *buffer* e um *kernel* são usado para fazer os cálculos passo a passo. Mudar o conjunto de neurônios a ser processado requer endereçar valores de pesos diferentes na memória. Como visto na seção 3.5.4, a mudança dos pesos previamente armazenados dura apenas um ciclo de relógio, bastando modificar o valor do registrador de configuração cfg_layers no campo $cfg\_virtual\_conf < x >$. O esquema de virtualização é mostrado na figura 3.15.

O número de neurônios na configuração tem sido limitado apenas pela capacidade de memória embutida da aplicação.

3.7 Sumário

Neste capítulo foi apresentada a arquitetura detalhada do neuroprocessador proposto, visando atender os requisitos citados na seção 3.1. Aspectos relativos à micro arquitetura do sistema, à tecnologia e à utilização foram abordados. Um sumário das propriedades do sistema proposto nos vários modos de implementação é exibido na tabela 3.3.

Tendo determinado os requisitos do sistema a ser implementado, um integrador de

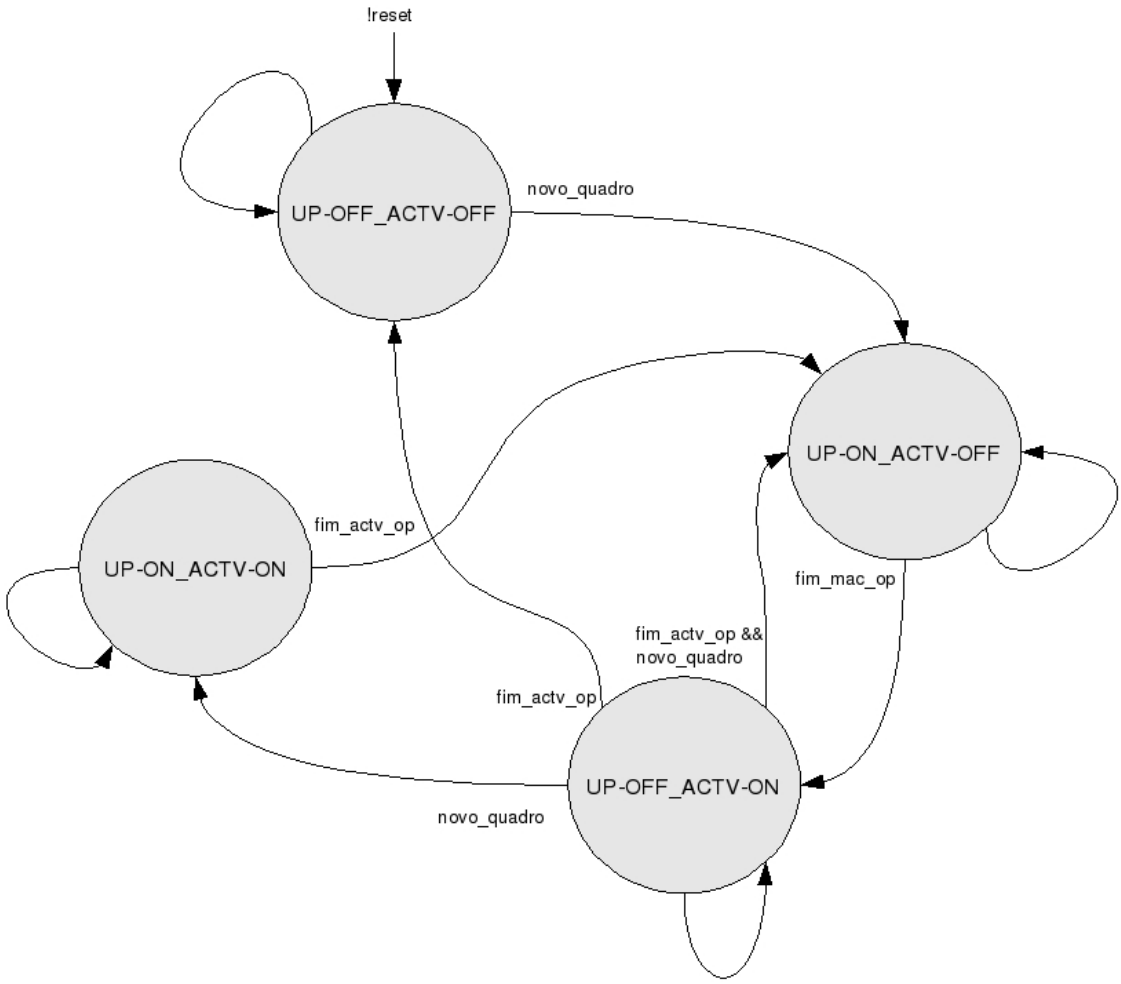


Figura 3.14: Máquina de estados para um núcleo do caminho de dados

ASIC poderá, ajustando os parâmetros descritos na tabela, adaptar o neuroprocessador às suas necessidades.

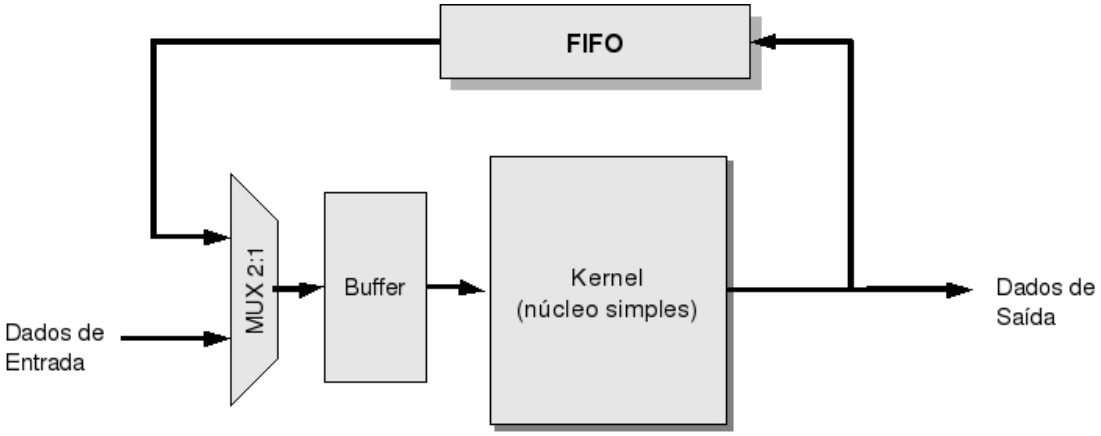


Figura 3.15: Esquema de virtualização

Tabela 3.3: Sumário das propriedades do neuroprocessador

Modo	*kernels*	núcleos	Max n^o de camadas	LUT	N^o de UPs	N^o de topologias suportadas	N^o de neurônios por camada
Desempenho - Núcleo Simples	1	1	1	1	depende da área	depende da memória	menor ou igual ao de UPs por núcleo
Desempenho - Núcleo Duplo	1	2	1 a 2	1	depende da área	depende da memória	menor ou igual ao de UPs por núcleo
Desempenho - Nucleo triplo	2	2+1	1 a 3	1	depende da área	depende da memória	menor ou igual ao de UPs por núcleo
Área - Virtualização	1	1	1 a 3	1	depende da área	depende da memória	depende da memória

Capítulo 4

Projeto em Circuito Integrado

4.1 Metodologia de Projeto

Desenvolver um módulo de propriedade intelectual com características ajustáveis, tal como o neuroprocessador proposto, requer o uso de uma metodologia apropriada para garantir reusabilidade do projeto e portabilidade entre tecnologias. A metodologia também deve prover abstração suficiente de modo a permitir a implementação do sistema em tempo factível, dada a complexidade dos circuitos envolvendo milhões de transistores dos chips atuais [Keating e Bricaud 2002]. As ferramentas de automatização eletrônica de projeto (EDA) são elementos indispensáveis para levar o projeto da abstração à fabricação considerando essa complexidade.

Segundo [Keating e Bricaud 2002], dentre as metodologias de projeto de circuito integrado atuais, a mais difundida na indústria e que consegue atender os requisitos mencionados acima é a metodologia de células padrões (standard cells). Esta metodologia, que é direcionada exclusivamente para sistemas digitais, utiliza portas lógicas padronizadas para compor o funcionamento desejado a partir de uma descrição do sistema em um nível mais alto de abstração. Estas portas lógicas (ou células) possuem o *layout* padronizado visando facilitar a disposição e a interconexão automatizadas das mesmas.

Em geral, as empresas de fabricação de circuito integrado (*foundry*) fornecem, compilados em uma biblioteca, todas as características físicas e lógicas das células para uma determinada tecnologia. As principais características são: temporização, lógica booleana, *layout* e modelo elétrico. Estas informações são utilizadas em diversas etapas do projeto.

De fato, as ferramentas de EDA desempenham três tarefas essenciais que caracterizam a metodologia:

- Mapeamento tecnológico;
- *Placement*;
- Roteamento.

No mapeamento tecnológico (ou síntese lógica), a descrição funcional do sistema em uma HDL de alto nível é traduzida para uma representação em mais baixo nível, usando apenas as portas lógicas contidas na biblioteca de células. O *placement* é a etapa automatizada de disposição das portas lógicas geradas no mapeamento tecnológico na área do chip. A interconexão entre as portas lógicas é realizada no roteamento, também

de forma automatizada. A figura 4.1 ilustra de forma simplificada o processo de projetar o chip desde a sua arquitetura até o *layout* final, utilizando a metodologia de células padrões. Essas etapas mencionadas acima se desdobram em dezenas de sub-etapas das quais nem todas são usadas em todos os projetos. Mais adiantes neste capítulo, algumas delas serão descritas.

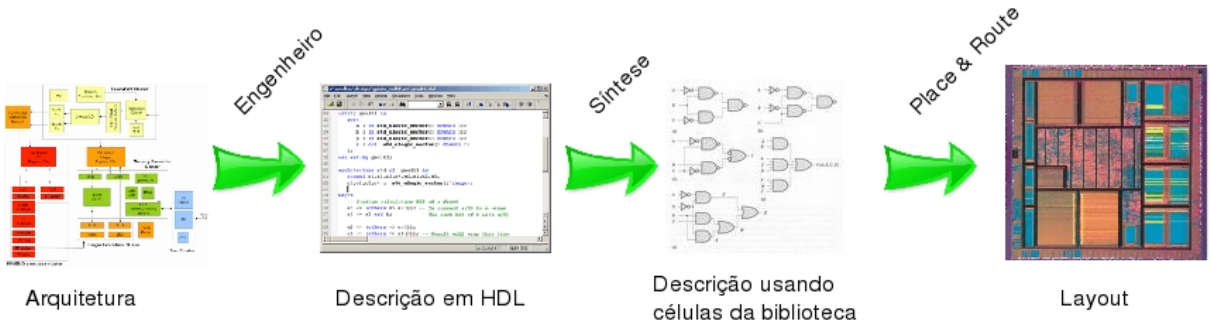


Figura 4.1: Versão simplificada das etapas da metodologia de células padrões

A portabilidade entre tecnologias é basicamente conseguida por conta da síntese lógica, onde um projeto em HDL pode ser facilmente migrado de uma tecnologia para outra utilizando uma ferramenta de EDA que faz o mapeamento tecnológico em poucos minutos (ou horas). O *layout* também deve ser refeito, normalmente durando algumas semanas.

Como a parte funcional do projeto é desenvolvida em HDL de alto nível, a reusabilidade pode ser obtida usando técnicas parecidas com as encontradas em desenvolvimento de *software*.

Mesmo para os sistemas de baixa e média complexidade, realizar síntese lógica, *placement* e roteamento sem o auxílio de ferramentas de EDA não é uma tarefa humanamente possível em tempo hábil. [Keating e Bricaud 2002] estima que, para um projeto de 12 milhões de portas lógicas, com um engenheiro projetando 100 portas lógicas/dia demandaria 500 anos para completar esta tarefa, com o custo associado a todos esses anos de homem-hora trabalhados. A abstração somada às ferramentas podem aumentar consideravelmente a produtividade, encurtando esse longo tempo de desenvolvimento para alguns meses, mesmo para sistemas de alta complexidade.

Outras duas questões estão sempre presentes durante o fluxo de projeto de circuitos integrados, apontadas por [Keating e Bricaud 2002]. A primeira é o compromisso entre área, consumo e desempenho. Estas três características estão sempre interligadas, de modo que melhorar uma muitas vezes implica em piorar as outras. Assim, o balanceamento da qualidade destas características é uma tarefa que deve começar desde o projeto da arquitetura. A segunda questão importante no projeto é a verificação da exatidão dos resultados produzidos a cada etapa do projeto. A verificação aborda aspectos funcionais, temporais, físicos e de integridade, como ilustrado na figura 4.2.

O fluxo de projeto em circuito integrado digital usando a metodologia de células padrões é dividido em duas grandes etapas, o *front-end* e o *back-end*. O *front-end* está relacionado com o projeto funcional do sistema. Já o *back-end* refere-se ao projeto físico

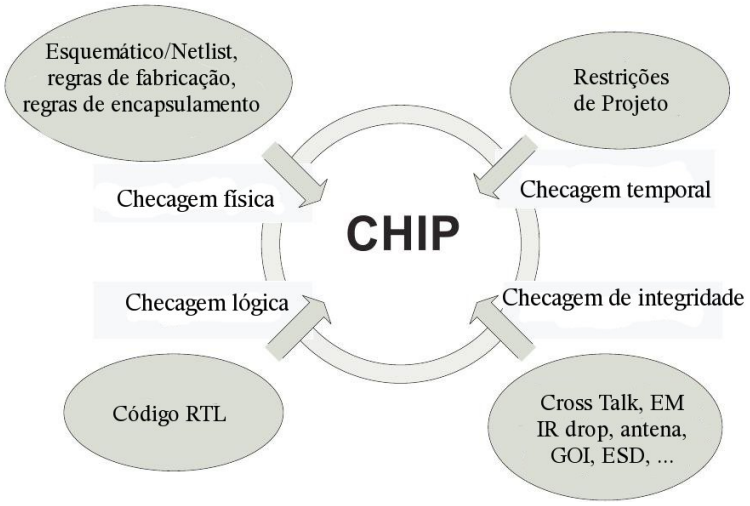


Figura 4.2: Verificações realizadas durante o desenvolvimento (Figura adaptada de [Xiu 2007])

do chip, ou seja, ao *layout*. A figura 4.3 ilustra a divisão do fluxo nestas duas grandes etapas, dando uma visão geral do fluxo.

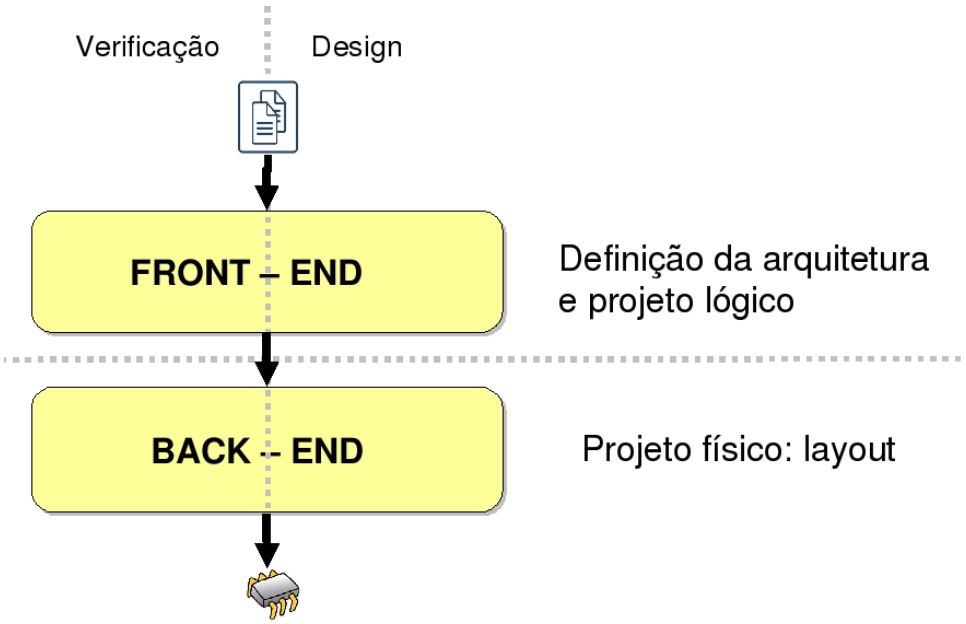


Figura 4.3: Visão geral do fluxo de projeto

O mapeamento tecnológico é a última etapa do *front-end*. Nesta etapa as primeiras informações com relação à implementação física começam a aparecer e a abstração começa a diminuir. A partir do mapeamento tecnológico, a utilização de ferramentas especializadas e com alto desempenho é impreterível. Na síntese e no *back-end*, as ferramentas

precisam lidar com milhões de elementos (portas lógicas) simultaneamente, sem perder de vista questões de consumo, área, desempenho e integridade, além de manter uma equivalência funcional com o que foi projetado no *front-end*.

As etapas do *front-end*, que não a síntese, também necessitam de ferramentas especializadas no seu desenvolvimento, porém com menos intensidade que no *back-end*. As ferramentas de simulação e verificação são as mais utilizadas nesse estágio. Uma versão simplificada das sub-etapas do *front-end* é mostrada na figura 4.4.

O neuroprocessador proposto foi desenvolvido usando esse fluxo, com exceção da etapa de Design for Test (DFT), que foge do escopo deste trabalho. A realização das etapas de especificação e arquitetura já foi apresentada no capítulo 3. As etapas de implementação do modelo de referência, desenvolvimento do projeto lógico, verificação funcional e síntese estão apresentadas na seção 4.2. Foram utilizadas as ferramentas da Cadence Design Systems durante as etapas do *front-end*.

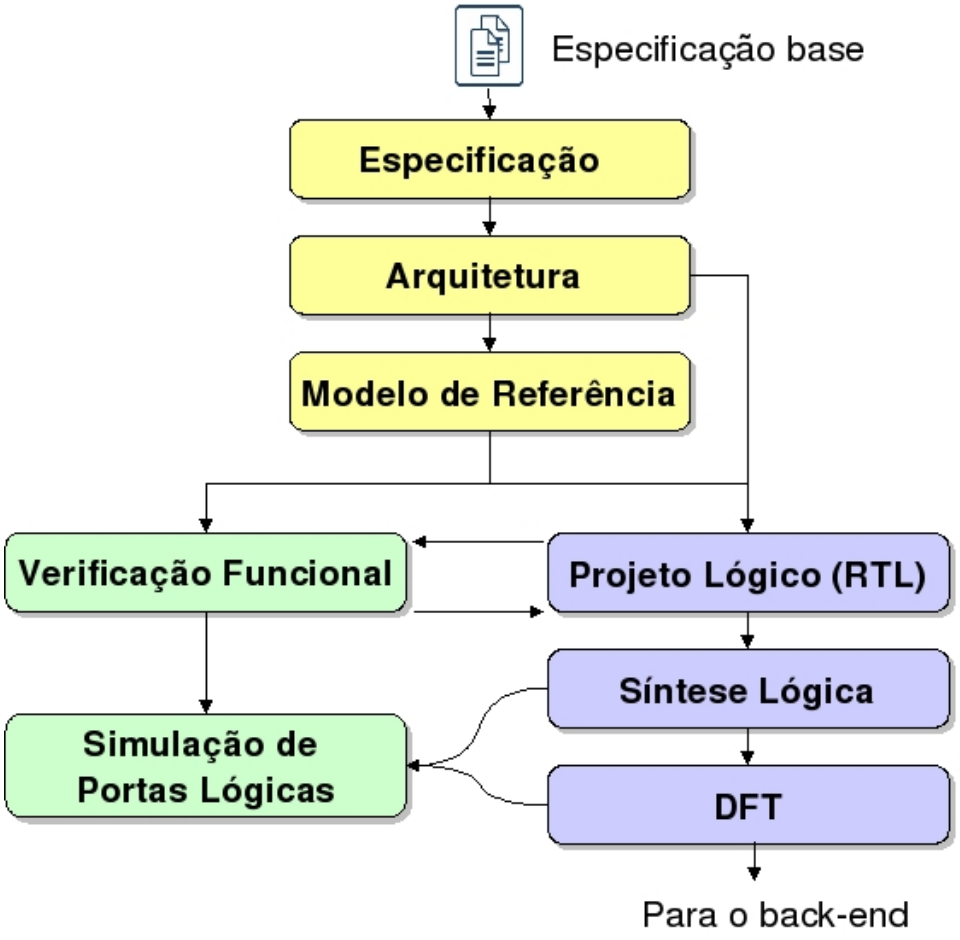


Figura 4.4: Sub-etapas do front-end

Para validar a viabilidade de implementação da arquitetura proposta em termos de área e consumo, também foi desenvolvido, como prova de conceito, o projeto *back-end* do neuroprocessador. O fluxo mostrado na figura 4.5 foi utilizado. As etapas de *floorplanning*,

placement, *Clock Tree Synthesis* (CTS) e roteamento foram realizadas. Também foram utilizadas as ferramentas da Cadence nessa etapa.

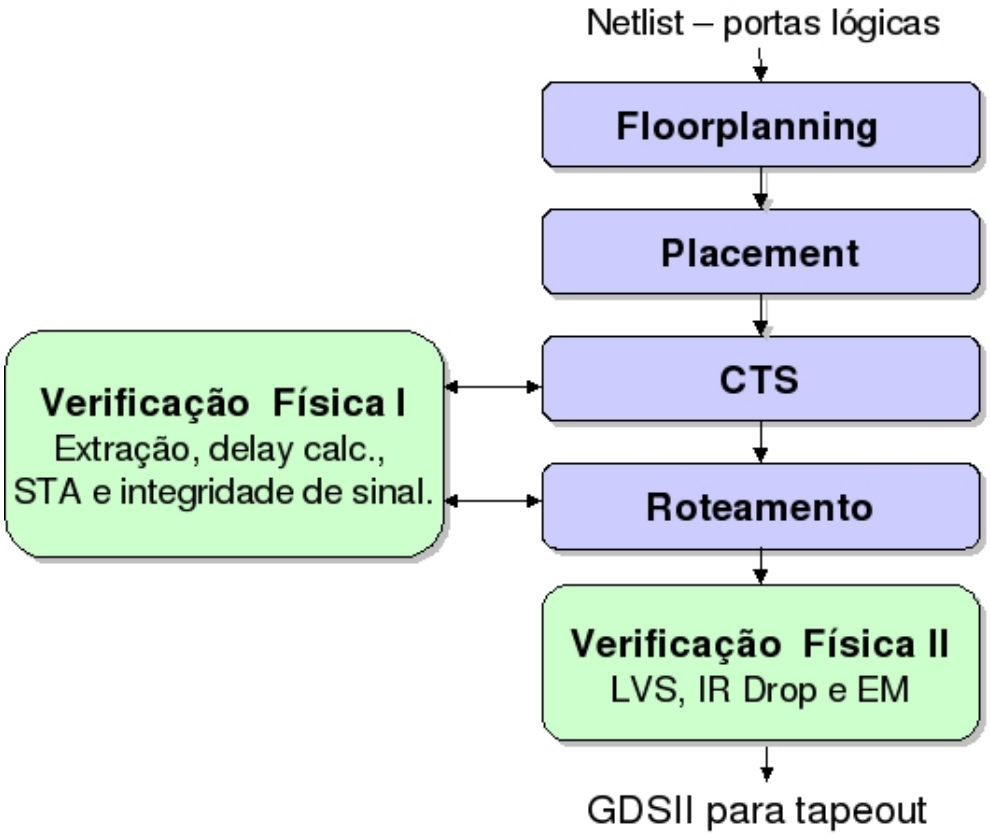


Figura 4.5: Sub-etapas do back-end

Um exemplo de implementação tem sido usado no decorrer deste capítulo tanto para o *front-end* quanto para o *back-end*. Os parâmetros adotados para esse exemplo foram 128 UPs, precisão de 16 bits para dados e pesos sinápticos, memória de LUT com 6144 posições, dois núcleos e 16384 palavras de memória de pesos (128 blocos de 128 palavras).

4.2 Front-end

4.2.1 Modelo de Referência

Idealizada a arquitetura do *hardware*, se faz necessário garantir que o projeto funcionará antes de iniciar as etapas de desenvolvimento do *hardware*. No caso de um sistema para processamento digital de sinais é fundamental avaliar se as técnicas de implementação discreta são adequadas e produzirão resultados com erros dentro de um faixa tolerável. Para tanto, um modelo do sistema deve ser desenvolvido utilizando uma linguagem de programação de alto nível (C/C++, SystemVerilog, Matlab). Neste modelo em alto ní-

vel, apenas o algoritmo de processamento da informação é considerado, enquanto que os aspectos relativos à implementação em *hardware* são deixados de lado, tornando-o menos propenso a erros.

Utilizando o modelo em alto nível também é possível estimar a quantidade de recursos de *hardware* necessária para implementar o sistema. Isto pode ser feito considerando o número de operações aritméticas e a quantidade de memória necessária para executar o algoritmo. Com relação à quantidade de memória, a estimativa pode não ser muito boa, pois esta é muito dependente da implementação em *hardware*.

Outra aplicação do modelo em alto nível (e talvez a mais importante) é como referência para a implementação do sistema em *hardware*. Durante o processo de desenvolvimento, é necessário se certificar de que o projeto lógico está efetuando as operações de acordo com o definido nas especificações e na arquitetura do sistema. Então, o algoritmo do sistema implementado em alto nível, que tem menos propensão ao erro, pode ser utilizado como modelo de referência, validando as saídas fornecidas pelo projeto de *hardware* durante a fase de simulação. A figura 4.6 ilustra como a comparação dos dados de saída é realizada durante a simulação.

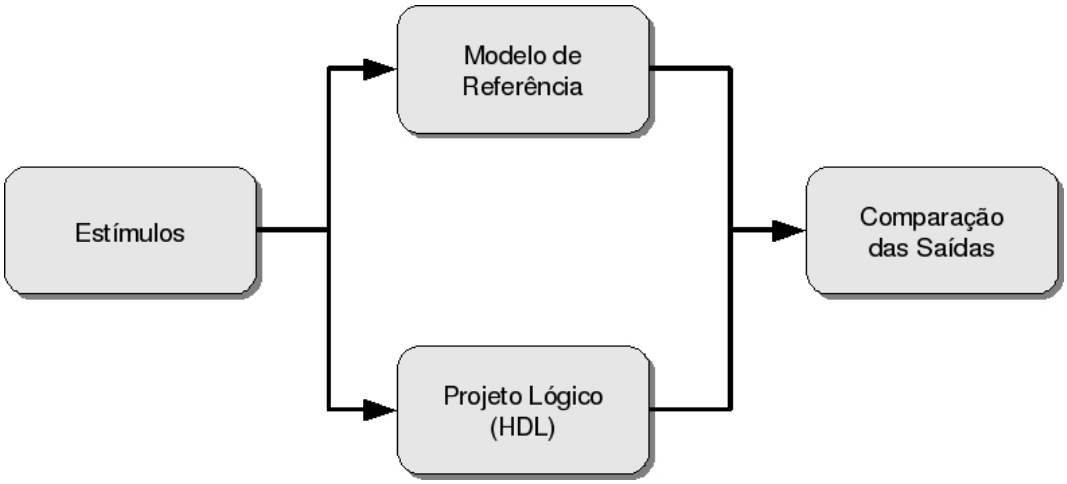


Figura 4.6: Comparação entre modelo de referência e projeto lógico durante simulação

Portanto, dentre as principais aplicações do modelo de referência, utilizadas no projeto proposto, temos:

- Validação do algoritmo discreto;
- Avaliação dos recursos disponíveis;
- Utilização como modelo de referência para o projeto.

Em alguns casos, o modelo de referência também é utilizado para guiar a implementação, mas essa abordagem não foi adotada neste projeto.

O Matlab foi escolhido como plataforma de desenvolvimento para o modelo de referência do *hardware* pela facilidade de uso e por ser dotado de diversas ferramentas para redes neurais. Foram desenvolvidos *scripts* do Matlab contendo o algoritmo de processamento de redes MLP fazendo somente operações em ponto fixo. As UPs fazem a operação

como descrito em 3.5.2, truncando os dados de forma variável, de acordo com um número pré-definido para o máximo de predecessores da configuração. A função de transferência, assim como no projeto do *hardware*, utiliza uma LUT com o mesmo método de discretização citado na seção 3.5.3. A figura 4.7 mostra um diagrama do modelo de referência do neuroprocessador.

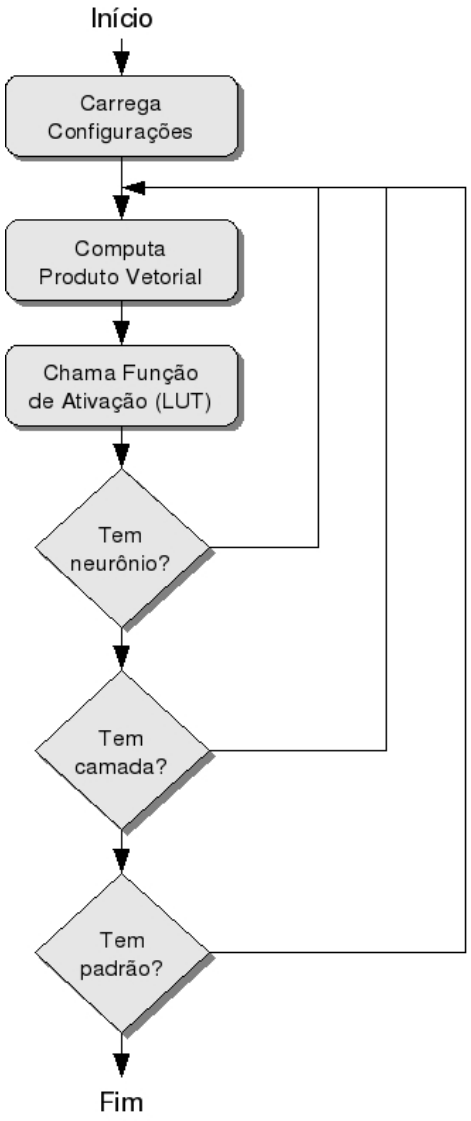


Figura 4.7: Diagrama do modelo de referência

Essa implementação não considera os aspectos de *hardware*, como a arquitetura de memória, a temporização, a unidade de controle ou a virtualização, reduzindo a probabilidade de erro na escrita do código-fonte. Então suas saídas são tidas como corretas para verificar o bom funcionamento do projeto do *hardware*.

4.2.2 Projeto Lógico

Esta é a etapa de projeto em que a especificação e a arquitetura são descritas considerando aspectos de implementação em *hardware*. Registradores, memórias, barramentos e máquinas de estados começam a figurar como elementos importantes para consecução dos resultados esperados. O projeto lógico descreve as funcionalidades do sistema através do comportamento de blocos construtores e como estes blocos estão interconectados [Xiu 2007]. Isso permite a simulação das funcionalidades e o mapeamento tecnológico em um nível de abstração do sistema ainda alto, possibilitando aos projetistas se concentrarem em decisões estratégicas do projeto e aumentando a sua produtividade.

Para fazer esse elo de ligação entre especificação e o *hardware* são utilizadas linguagens HDL. Estas linguagens possuem as características mencionadas acima de descrever o *hardware*, mantendo um nível de abstração. Além disso, o uso de uma HDL pode beneficiar o processo de desenvolvimento de CI na documentação do projeto, na simulação comportamental (mais rápida) e na síntese da descrição em *hardware* real, com o auxílio de ferramentas de EDA. As duas linguagens principais são VHDL e Verilog HDL, sendo que a segunda foi utilizada neste projeto.

As HDLs possuem ainda três níveis de abstração: nível de portas lógicas, nível de transferência de registradores (RTL) e nível comportamental. O nível de portas lógicas descreve o *hardware* em termos de portas lógicas primitivas, priorizando a representação da implementação em *hardware* em detrimento da representação da funcionalidade. O nível comportamental é o oposto do nível de portas lógicas, representando apenas a funcionalidade. O RTL está entre a representação comportamental e a de portas lógicas, sendo então o nível de abstração adequado para a transição da especificação/arquitetura para o *hardware*.

Portanto, no RTL o funcionamento do sistema é definido descrevendo como os dados são transferidos e manipulados. Os elementos de armazenamento são referenciados, bem como a manipulação de dados entre eles, mas ainda sem fazer menção à tecnologia utilizada. Decisões de implementação que ponderam área, consumo e desempenho também podem ser realizadas durante o desenvolvimento do RTL.

Por exemplo, um circuito acumulador pode ser implementado em RTL como no código Verilog a seguir:

```
1  module acumulador
2  #(parameter IN_WIDTH = 8, OUT_WIDTH = 32)
3  (input clk,
4  input rst_n,
5  input signed [IN_WIDTH-1:0] data_in,
6  output reg signed [OUT_WIDTH-1:0] data_out);
7
8  always @ (posedge clk or negedge rst_n) begin
9      if(!rst_n)
10         data_out <= 0;
11     else
12         data_out <= data_out + data_in;
```

```
13 end
14 endmodule
```

Observe que, no código, são mencionados a operação, a atribuição de valor ao registrador, a largura das portas de entradas e saída e o sincronismo da operação. Porém, não existe referência às portas lógicas com as quais o circuito será implementado.

O desenvolvimento do RTL do neuroprocessador proposto seguiu a mesma divisão de módulos apresentada no capítulo 3, onde aspectos de implementação já foram considerados na definição da arquitetura do sistema. Alguns blocos auxiliares foram necessários para resolver problemas específicos de implementação. Um diagrama de blocos da estrutura completa do RTL do *kernel* é exibido na figura 4.8.

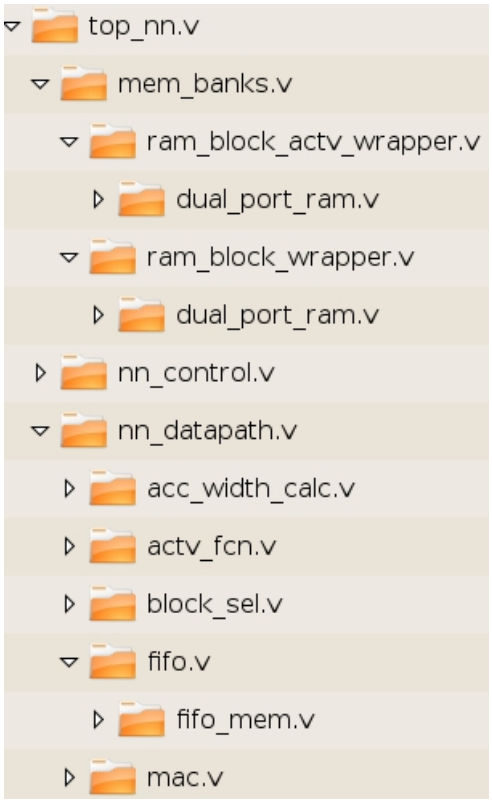


Figura 4.8: Organização hierárquica do RTL

A seguir a descrição de cada um dos módulos exibidos na figura 4.8:

- top_nn.v: módulo principal do *kernel* que faz as interconexões entre os blocos principais do *kernel* e conecta estes às portas de entrada e saída. Neste módulo não há lógica;
- mem_banks.v: Módulo de memória que encerra todas as memórias do sistema. Os pesos sinápticos, a LUT e as configurações estão contidas nesse módulo. Utilizando um construtor *generate*, instancia a quantidade de blocos de memória configurados;
- ram_block_wrapper.v: Encapsula a memória referente ao bloco de memória. Durante a simulação, um modelo de memória é utilizado para emular a SRAM de porta

dupla. Durante a síntese, este modelo é ocultado para dar lugar aos *hard blocks* de memória gerados pelo gerador de memórias;
- ram_block_actv_wrapper.v: Encapsula a memória referente à LUT. Durante a simulação, um modelo de memória é utilizado para emular a SRAM de porta dupla. Durante a síntese, este modelo é ocultado para dar lugar aos *hard blocks* de memória criados pelo gerador de memórias;
- dual_port_ram.v: Modelo de memória para emular a SRAM de porta dupla;
- nn_control.v: Contém a lógica de controle do *kernel*. A lógica possui duas máquinas de estado para cada núcleo do caminho de dados e uma lógica geral de configuração das duas máquinas de estado;
- nn_datapath.v: Caminho de dados onde todo o processamento aritmético do sistema está contido. Instancia o número de UPs e de núcleos de acordo com a configuração. No caso de dois núcleos também instancia uma FIFO;
- acc_width_calc.v: Calcula o tamanho real do acumulador segundo a equação 3.3, para ser usado durante o truncamento na UP;
- actv_fcn.v: Implementa a função de ativação conforme descrito na seção 3.5.3;
- block_sel.v: Seleciona os fios corretos do banco virtual de memória para cada UP. A seleção ocorre durante a compilação do projeto, na fase de elaboração;
- fifo.v: Lógica para a memória do tipo FIFO usada na comunicação entre os núcleos;
- fifo_mem.v: Memória usada na FIFO. Como a quantidade de memória utilizada é pequena, não houve a necessidade de utilização do gerador;
- mac.v: Implementa a UP conforme descrito na seção 3.5.2.

Além dos arquivos listados acima, há os códigos RTL relacionados à virtualização:

- top_virtualization.v: módulo principal quando o sistema utiliza virtualização. Faz as interconexões entre os blocos principais e conecta estes às portas de entrada e saída. Neste módulo não há logica;
- virtual_mux.v: Multiplexador para selecionar se os dados da entrada vêm da porta de entrada ou se vêm da FIFO;
- virtual_control.v: Lógica de controle da virtualização. Contém pequena memória que guarda a configuração;
- dual_port_ram.v: O mesmo modelo de memória utilizada no *kernel*. Aqui usada como *buffer* intermediário da virtualização;
- fifo.v: O mesmo módulo de FIFO utilizado no *kernel*. Aqui usado no processo de virtualização;
- fifo_mem.v: Memória usada na FIFO;
- top_nn.v: Instância do *kernel* utilizada durante a virtualização. Instancia todos os sub-módulos nele contidos, conforme a figura 3.1.

Na maioria dos módulos descritos anteriormente, a preocupação principal durante o desenvolvimento do RTL foi com relação à área, devido ao número grande de memórias. A única exceção é a UP no qual estão contidas as aritméticas necessárias para o processamento de cada neurônio. A escolha da arquitetura da UP é crítica, pois afeta a área, devido às várias UPs instanciadas, e o desempenho, pois contém o caminho combinacional crítico do sistema. O esquemático da UP é mostrado na figura 4.9.

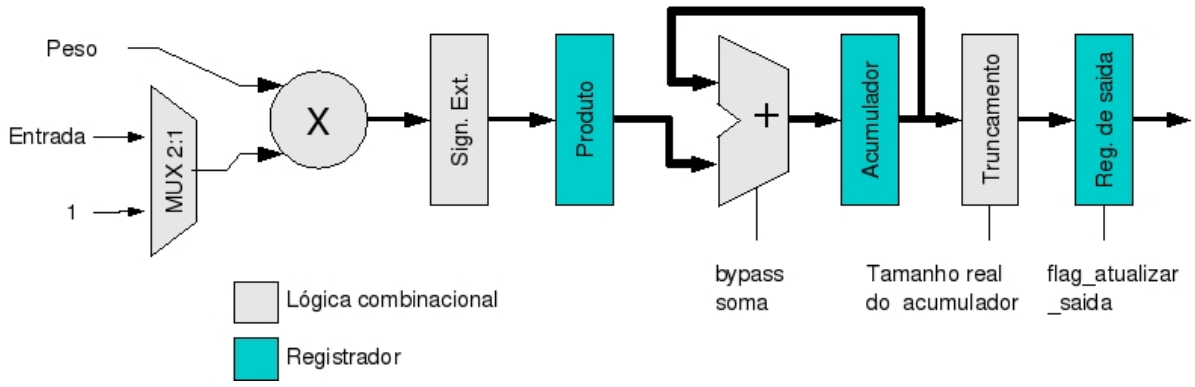


Figura 4.9: Esquemático da UP

A operação de multiplicação entre o peso e o dado de entrada e a operação de somar e acumular os dados foram separadas em etapas diferentes, como pode ser visto na figura 4.9. Mesmo que as tecnologias de integração atuais permitam colocar uma quantidade significativa de lógica combinacional entre dois registradores, graças à velocidade das portas lógicas, e trabalhando a uma freqüência razoável, esta estratégia não foi adotada. Apesar da latência total do sistema aumentar pelo fato de ter sido adicionado mais um estágio com registrador, o *throughput* também pode melhorar, caso seja utilizada uma freqüência de trabalho mais alta. Com caminhos combinacionais mais curtos é possível aumentar a freqüência de trabalho.

No anexo A encontra-se o código-fonte RTL em Verilog da UP (mac.v). Na figura 4.10 as formas de onda de uma simulação da UP são exibidas. Como pode ser notado na figura 3.6 e comprovado na figura 4.10, a restrição mencionado na seção 3.5.1 pode aparecer se, no momento de carregar o dado do acumulador no registrador de saída, o módulo da função de ativação ainda não tiver utilizado o dado da operação anterior. Nesse caso, a unidade de controle não mandará o pulso para a porta flag_atualizar_saida, fazendo com que a UP fique impossibilitada de receber outro vetor de entrada. Estes intervalos entre os envios são observados na figura 4.10 nos instantes em que o sinal *sink_valid* está baixo.

Este intervalo de espera entre vetores de entrada nas UPs refletirá em intervalos de espera entre padrões de entrada no neuroprocessador. Estes intervalos de espera podem ser vistos nas formas de onda mostradas na figura 4.11, onde é mostrado o envio de padrões de entrada para o neuroprocessador. Novamente, os intervalos entre os envios são observados nos instantes em que o sinal *sink_valid* está baixo.

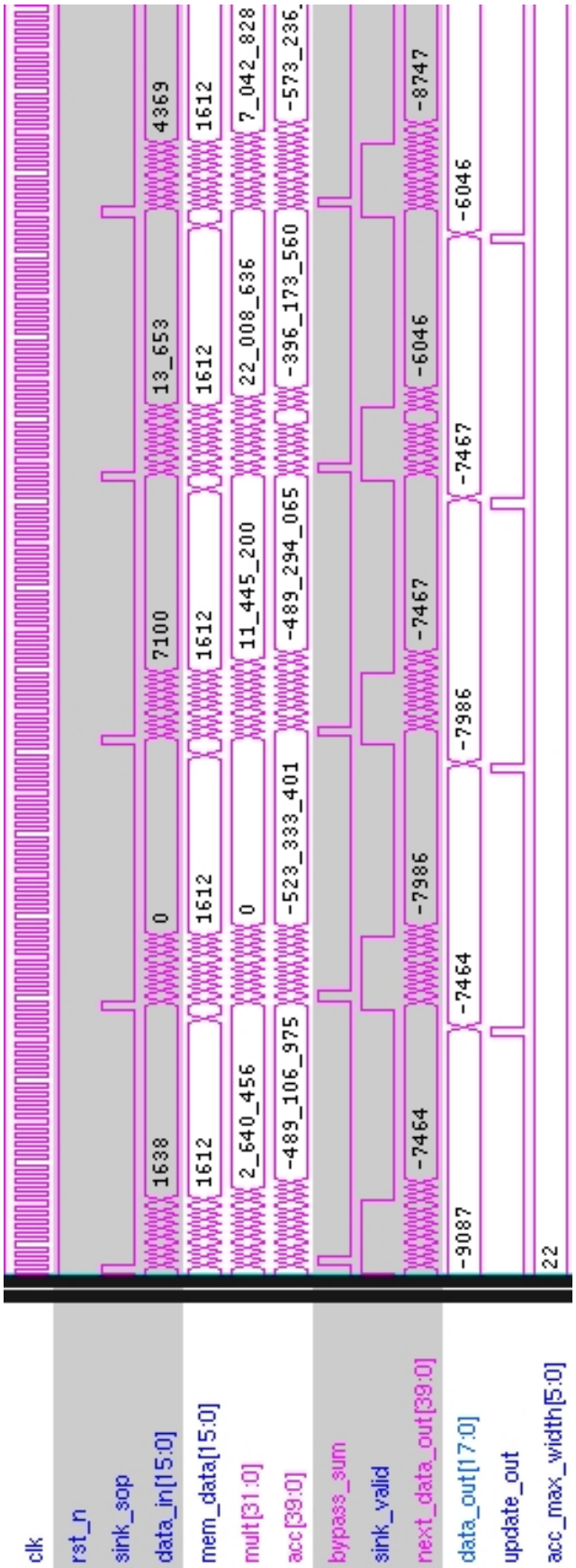


Figura 4.10: Sinais do processamento na UP

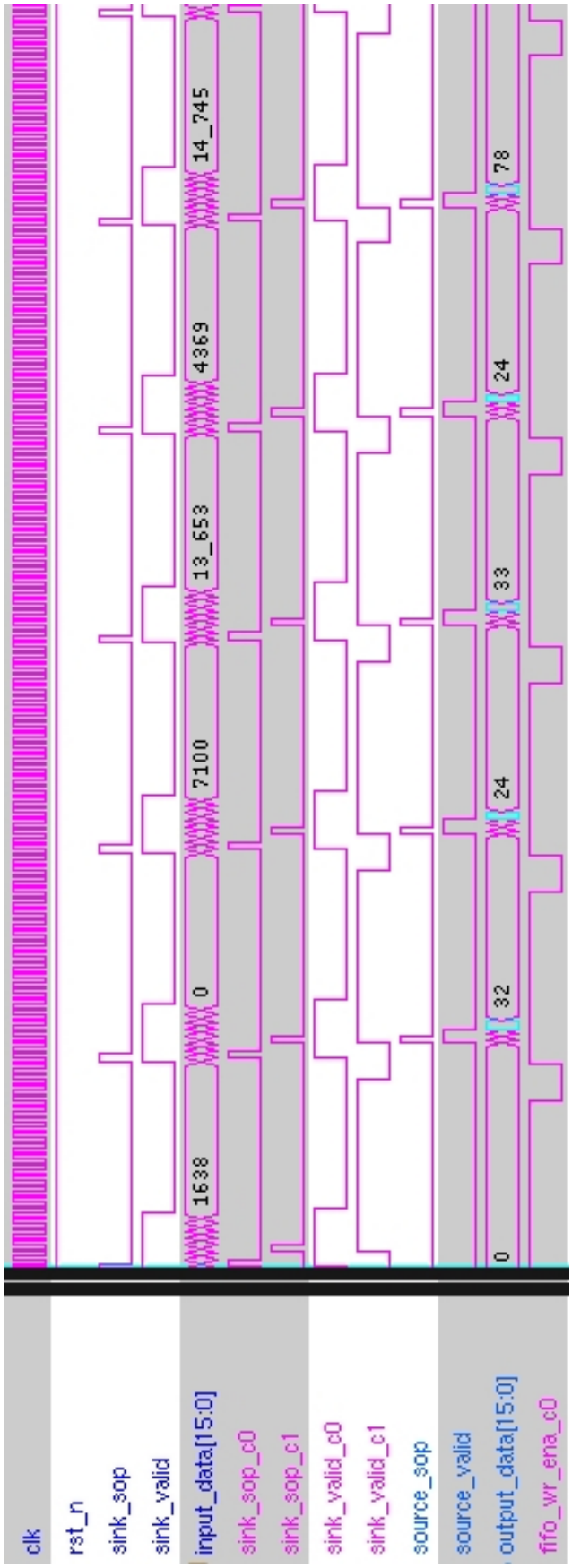


Figura 4.11: Envio de dados ao neuroprocessador

4.2.3 Verificação Funcional

A verificação funcional é o processo de certificar que a descrição RTL do sistema possui todas as funcionalidades desejadas implementadas corretamente [Bergeron 2000]. Em outras palavras, este é o processo de encontrar erros inseridos durante o projeto lógico e de garantir que tudo o que foi definido durante a fase de especificação está contemplado no RTL.

No projeto de um ASIC, um erro funcional pode resultar em um grande prejuízo financeiro, dependendo do estágio de desenvolvimento. Se o projeto estiver avançado, o prejuízo para refazer as etapas está associado com os custos dos projetistas e com as ferramentas de EDA. Se o erro for detectado após a fabricação do chip, além dos custos de engenharia, há também os custos de fabricação que, na presente data, variam de centenas de milhares a milhões de dólares americanos.

Por conta disso, muito esforço é empregado em verificação funcional de projetos de ASIC, onde diversas técnicas são utilizadas para o caso de projetos complexos. Segundo [Bergeron 2000], a grande quantidade de estados em sistemas digitais complexos faz com que haja um número altíssimo de possibilidades de excitação das entradas, inviabilizando definitivamente simulações exaustivas (tentar todas as combinações de entradas). É desejável o uso de técnicas que garantam a confirmação da implementação das funcionalidades especificadas em um tempo factível.

Dentre todas as técnicas de verificação, um fator comum é a utilização do *testbench* como elemento provocador da interação com a descrição RTL. O *testbench* é uma entidade implementada em HDL para estimular o projeto lógico em RTL e avaliar o seu comportamento em ambiente de simulação, como ilustrado na figura 4.12. As principais linguagens utilizadas são VHDL, Verilog, SystemVerilog, SystemC e ‘e language’.

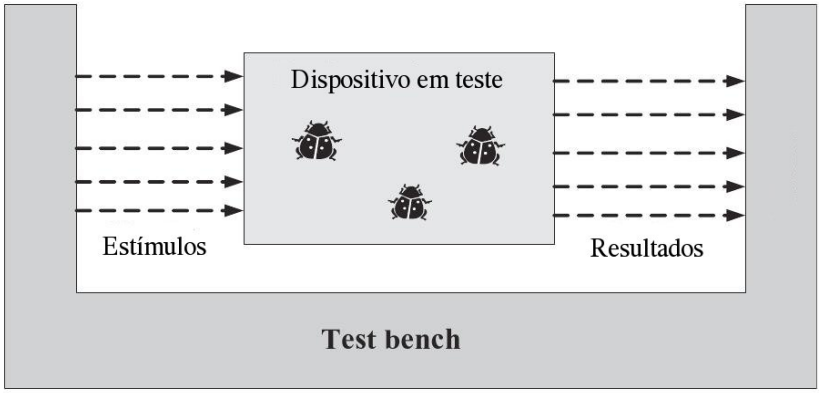


Figura 4.12: Simulação do sistema com um testbench (Figura adaptada de [Xiu 2007])

Na verificação do projeto do neuroprocessador, as técnicas utilizadas foram a inspeção de formas de onda, os testes diretos e a verificação orientada à cobertura (*coverage-driven*). A inspeção de formas de ondas é a verificação mais básica e intuitiva, mas ainda com grande serventia para depuração das funcionalidades básicas. Esta consiste em observar as formas de onda para avaliar se a resposta do sistema aos estímulos está correta.

Na verificação por testes diretos, dados são inseridos no sistema e a saída gerada é comparada com uma saída já conhecida associada à entrada. Então, pares de entrada e saída são utilizados para exercitar funcionalidades conhecidas, atestando seu funcionamento. Neste projeto, conjuntos de dados de reconhecimento de padrões foram utilizados como testes diretos do sistema. Trata-se de dados de diversas aplicações com várias topologias de RNA diferentes, todas provenientes do conjunto Proben1 [Prechelt 1994].

Entretanto, em sistemas de grande complexidade é difícil e trabalhoso olhar para formas de ondas ou ter vetores suficientes e variados, de modo a cobrir um número razoável de funcionalidades. Surge então a necessidade de métodos de verificação em que os vetores possam ser gerados automaticamente quando há necessidade de cobrir mais funcionalidades.

Os métodos orientados à cobertura são usados justamente para atender essas necessidades e outras que aparecem em conseqüência da geração automática. Tais métodos podem ser vistos como um conjunto de técnicas visando automatizar a geração de dados e avaliar o progresso da verificação. As principais características dos métodos orientados à cobertura são:

- Geração aleatória de estímulos de entrada;
- Abstração da comunicação entre o *testbench* e o sistema;
- Auto-checagem da resposta dos circuitos usando modelo de referência;
- Avaliação do progresso usando métricas de cobertura funcional e de código.

O processo de verificação começa na geração de dados aleatórios em um nível alto de abstração. Estes dados, apesar de aleatórios, são gerados com restrições para manter conformidade com o problema. Em seguida, os dados em alto nível são enviados através de um Bus Functional Model (BFM), cuja função é traduzir os dados gerados em alto nível para os sinais de baixo nível na porta de entrada do sistema.

O BFM também recebe a resposta do sistema conforme o protocolo descrito em 3.4, convertendo os dados de resposta para uma representação em alto nível. Os dados recebidos são então confrontados com respostas geradas pelos modelos de referência para checar a validade do resultado. Para avaliar o andamento da verificação, são utilizadas métricas de cobertura funcional, com as quais é possível determinar se as funcionalidades descritas na especificação foram atendidas durante a simulação. Também são usadas métricas de cobertura de código para determinar se todos os códigos e todas as possibilidades de execução foram exercitados.

Linguagens específicas são utilizadas para descrever as funcionalidades em alto nível, tais como SystemVerilog, SystemC, PSL e *e language*. Utilizando essa ideologia de geração aleatória e cobertura funcional, diversas metodologias têm surgido implementando modernas infra-estruturas de verificação, dentre as quais estão OVM, URM, SVM e *e*RM.

Para o *hardware* proposto, foi utilizada a Metodologia de Verificação do LSITEC-NE (LVM) que, assim como as metodologias mencionadas acima, possui todas as características dos métodos orientados à cobertura. O LVM foi desenvolvido pelo autor desta dissertação no Laboratório de Sistemas Integráveis Tecnológico Nordeste (LSITEC-NE) e cedido pela instituição para ser utilizado neste trabalho. A figura 4.13 mostra um diagrama do ambiente de simulação do LVM. Esta metodologia consite em um conjunto

de códigos em SystemVerilog que simplificam a criação de novos *testbenchs*, utilizando técnicas modernas de verificação como geração aleatória, cobertura funcional, abstração dos dados etc.

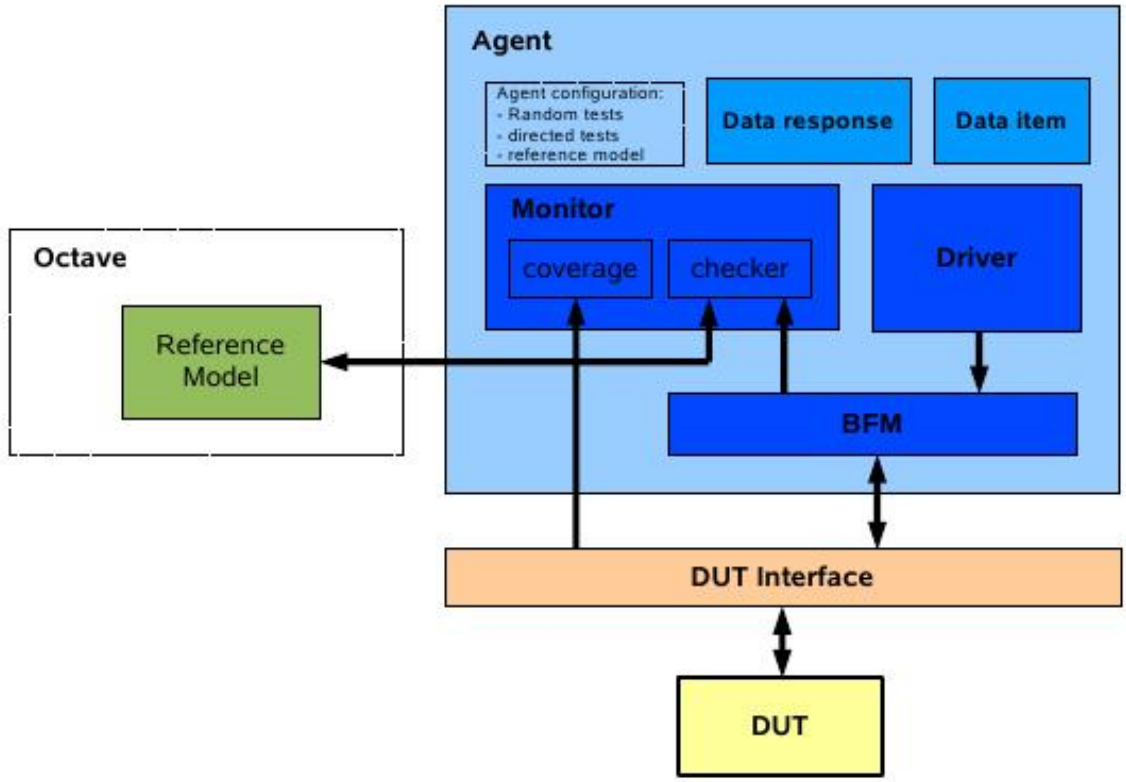


Figura 4.13: Ambiente de Simulação do LVM

Durante a simulação, testes diretos e aleatórios foram utilizados para atingir 100% de cobertura funcional. Diversas topologias e diversos padrões de entrada para cada topologia foram gerados para atestar o funcionamento do circuito. A geração dos dados foi realizada utilizando estruturas apropriadas da linguagem SystemVerilog para dados aleatórios, tais como *Class Randomization* e *Constraints*. O modelo de referência escrito em linguagem do MATLAB foi utilizado para validar os resultados gerados pelo RTL. Como o modelo foi implementado em ponto fixo e com as mesmas etapas que o *hardware*, equivalência bit a bit foi conseguida nas comparações.

Como mostrado na figura 4.13, o modelo de referência é executado no programa GNU Octave, uma vez que este suporta a linguagem do MATLAB e apresenta facilidade de comunicação com o ambiente de simulação.

4.2.4 Síntese Lógica

Como mencionado anteriormente, a síntese lógica é uma das etapas mais importantes do fluxo de projeto de circuito integrado na metodologia de células padrões, por fazer a

transição do nível alto para um nível baixo de abstração, conferindo independência da tecnologia.

De acordo com [Scheffer et al. 2006], a síntese lógica é o processo de mapear um *hardware* descrito em RTL para portas lógicas, preservando a funcionalidade e utilizando apenas as portas lógicas presentes na biblioteca de células. Esta é uma tarefa extremamente complexa mesmo para sistemas de pequeno porte, uma vez que a ferramenta de EDA deve:

- Encontrar solução de compromisso entre área, desempenho e consumo a partir de restrições definidas pelo usuário;
- Garantir o mesmo comportamento do circuito gerado que do RTL;
- Processar projetos grandes, fazendo otimizações de forma globalizada;
- Realizar todo o processo em um tempo mais curto possível.

Todas essas considerações devem ser ponderadas pela ferramenta de EDA com base nas informações fornecidas pelo usuário (RTL e restrições) e pela *foundry*. As *foundries* fornecem bibliotecas contendo informações das portas lógicas disponíveis, descrevendo os tempos de resposta, a lógica, o *layout* e o consumo para cada porta. O resultado da síntese é um arquivo HDL, contendo uma descrição em baixo nível (*netlist*), usando apenas instâncias de portas lógicas e suas interconexões.

Neste projeto tem sido utilizada uma biblioteca na tecnologia CMOS de 90nm da SMIC, em conjunto com a ferramenta de síntese da Cadence, o Encounter RTL Compiler. A tabela 4.1 mostra os resultados da síntese para o exemplo de implementação do neuroprocessador. Estes resultados são referentes apenas ao caminho de dados e às logicas de controle, visto que os blocos de memória não são gerados durante a síntese.

Tabela 4.1: Resultados da Síntese

Característica	Resultado
Portas Lógicas	155318
Slack	375.2 ns
Potencia Estática	4.01 mW
Potência Dinâmica	37.00 mW
Potência Total	41.00 mW
Área	85347 μm^2

4.2.5 Geração de Memórias

Em geral, *hardwares* para RNA têm grande parte de suas áreas constituídas de memórias dentro do chip para armazenar os pesos sinápticos. Para essa quantidade de memória, utilizar registradores definidos no RTL não é apropriado, devido à baixa densidade de integração com relação a outras técnicas de síntese de memória [Keating e Bricaud 2002].

Inversamente, *softwares* dedicados para a geração de memórias são utilizados. Estes *softwares* são fornecidos pelas *foundries* e são dependentes da tecnologia. Como este é

um processo que depende da tecnologia e até da *foundry*, o fluxo de projeto lógico tem uma quebra na filosofia de se manter independente de tecnologia.

Os geradores conseguem sintetizar memórias em uma área reduzida, mas mantendo bons o consumo e o tempo de resposta. Avançadas técnicas de arquitetura de memórias e o descumprimento de regras de checagem do projeto (DRC) são as estratégias utilizadas para conseguir tais resultados.

O gerador de SRAM na tecnologia CMOS de 90nm da SMIC(R) foi utilizado para gerar a LUT e o bloco de memória dos pesos sinápticos. No projeto exemplo, têm sido adotadas 128 palavras para cada bloco de memória e 6144 palavras para a LUT, ambos com a palavra de 16 bits. O gerador tem como saídas o *hard block* da memória, um modelo em verilog, uma folha de especificações e bibliotecas com informações de atraso.

A figura 4.14 mostra a visão externa da memória gerada. Foi gerada uma memória para a LUT com dimensões 709.3 μm x 464.2 μm e para o bloco de memória 254.2 μm x 152.2 μm. Mais informações sobre o funcionamento da memória pode ser encontrado no anexo B, na folha de especificações criada pelo gerador de memória para o bloco de memória de 128 palavras.

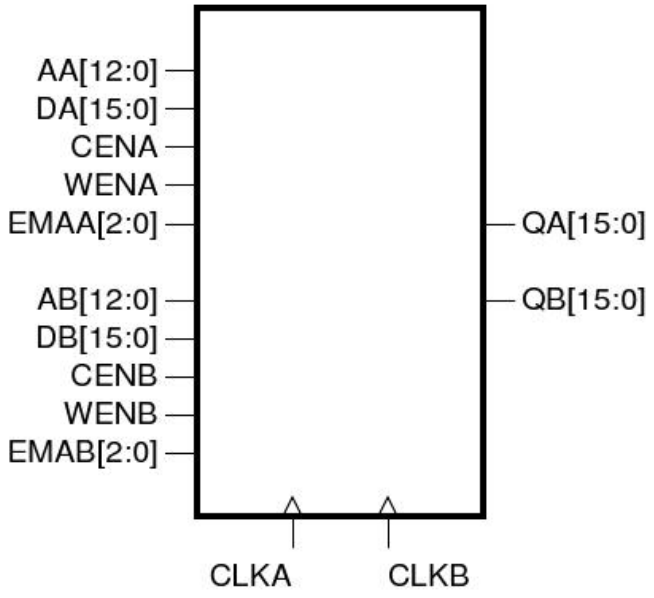


Figura 4.14: Visão externa da memória gerada

4.3 Back-end

Durante o desenvolvimento de um IP, o projeto físico tem o papel somente de provar a viabilidade de integração do mesmo. Isso devido ao fato das ferramentas de EDA atuais conseguirem melhores resultados fazendo otimizações globais. Mesmo nos casos em que o *layout* é feito bloco a bloco, os engenheiros que fazem a integração do sistema

normalmente refazem o *layout* para ajustar os parâmetros do IP e para garantir que este funcionará de acordo com o esperado. Os hard-IP (IP com *layout* pronto) vêm se tornando uma alternativa cada vez menos adotada.

Portanto, o *layout* do neuroprocessador foi realizado exclusivamente para provar a sua viabilidade em termos de área e roteabilidade. A seguir, algumas etapas são brevemente explicadas e o *layout* final é apresentado.

O *floorplanning* é o ponto de partida do projeto físico, onde são determinados o tamanho da área útil do chip, posição dos pinos de entrada e saída, disposição dos hard-blocks e o roteamento da alimentação [Scheffer et al. 2006]. Também pode ser indicado como as células lógicas serão posicionadas na etapa posterior. Esta é uma das principais etapas do projeto do neuroprocessador, uma vez que este é constituído em grande parte por hard-blocks de memória. A disposição das memórias foi feita manualmente. A área útil do CI escolhida foi de 4mm x 4mm.

Ainda segundo [Scheffer et al. 2006], o *placement* é o processo de alocação das células lógicas na área útil do CI. A ferramenta de EDA deve procurar o melhor lugar para cada célula, respeitando os limites impostos no *floorplanning*. A área, o roteamento e o desempenho do sistema são severamente afetados pelo resultado da disposição das células. Um *placement* ruim pode levar a problemas de congestionamento durante roteamento ou fios muito longos, causando atrasos que reduzem o desempenho do sistema. No *layout* do neuroprocessador, esta etapa incluiu apenas os as portas lógicas provenientes da unidade de controle, do caminho de dados e alguma lógica do módulo de memórias.

O processo de conectar as células lógicas é feito durante a etapa de roteamento. Durante o *placement*, a posição das portas lógicas já foi definida. A ferramenta de EDA, a partir das conexões descritas no modelo sintetizado, tenta encontrar a melhor rota para ligar as portas das células. O tamanho dos fios, o desempenho do sistema e a integridade dos sinais são as principais preocupações durante o roteamento.

A figura 4.15 mostra o *layout* do *hardware* implementado. A grande quantidade de UPs e de memórias adotadas para este exemplo de implementação fez com que o *layout* tivesse uma área significativa (4 mm x 4 mm). Os retângulos azuis na figura 4.15 são os blocos de memória (*hard blocks*) criados pelo gerador de memória. Note que estes blocos ocupam uma parte significativa da área do neuroprocessador.

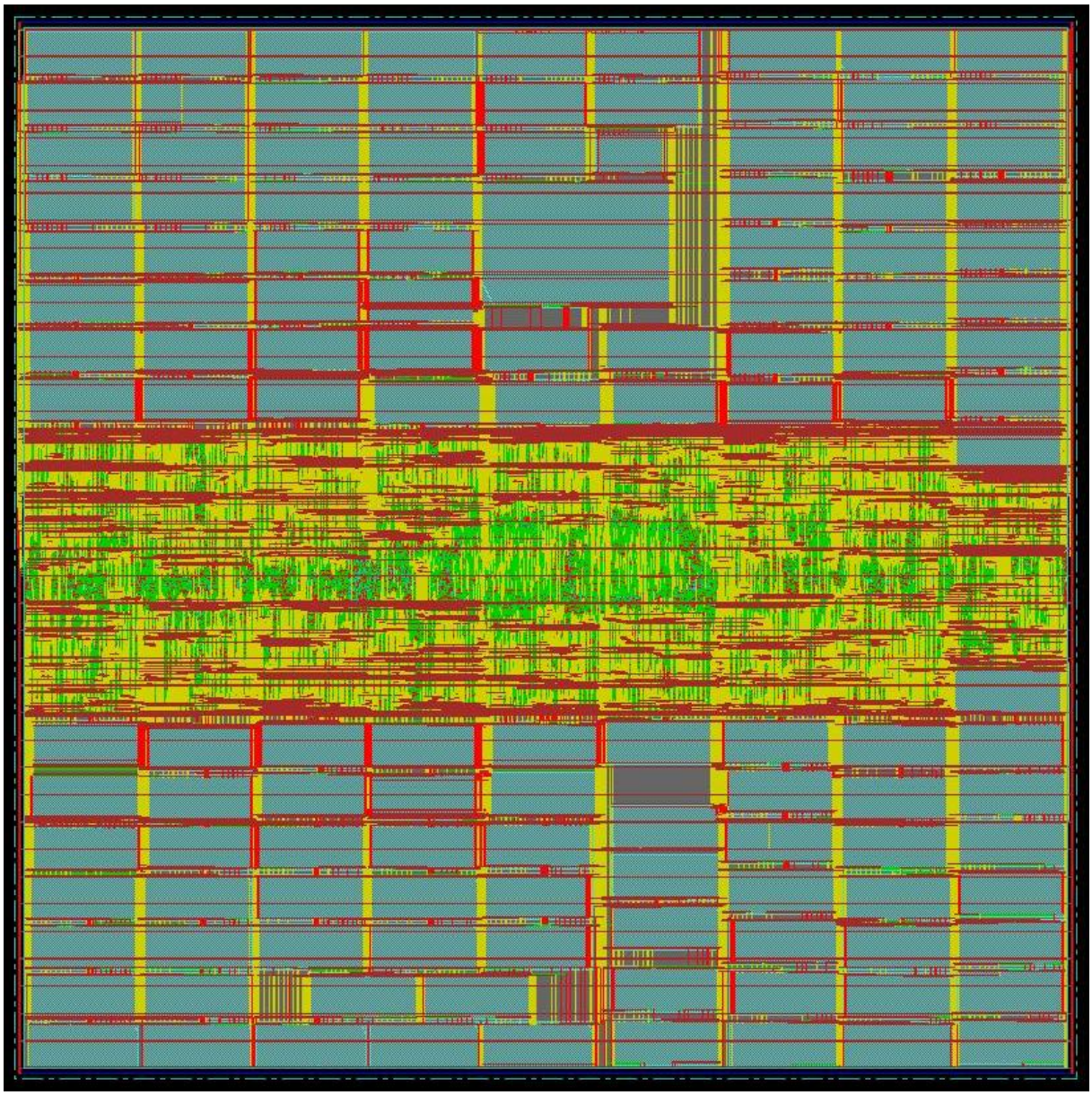


Figura 4.15: Layout do neuroprocessador

Capítulo 5

Avaliação de Características Físicas e de Desempenho

5.1 Análises do Erro

Como mencionado anteriormente, nos testes diretos realizados para validar o sistema foram utilizados conjuntos de dados disponibilizados em [Prechelt 1994], que compõem uma coleção de problemas de classificação de padrões e aproximação de funções provenientes de diversos domínios de aplicações diferentes, visando a utilização como um *benchmark* para redes neurais. Estes dados foram utilizados para avaliar tanto o erro introduzido durante o processamento em ponto fixo quanto o desempenho do neuroprocessador desenvolvido.

Em [Prechelt 1994], também são apresentados os resultados de treinamentos de diversas topologias de RNA para processar os conjuntos de dados do *benchmark*. Os resultados contêm os erros percentuais de cálculo e de classificação dos padrões para cada topologia, servindo como uma base de comparação imediata para outros trabalhos que utilizem estes padrões. O algoritmo RPROP, uma variante do *backpropagation*, foi utilizado em [Prechelt 1994] para o treinamento das RNAs. Todo o processamento foi feito em ponto flutuante.

A medida de erro de cálculo utilizada em [Prechelt 1994] foi o erro quadrático percentual, determinado pela expressão 5.1.

$$E_Q = 100 \cdot \frac{o_{max} - o_{min}}{N \cdot P} \sum_{p=1}^{P} \sum_{i=1}^{N} (o_{pi} - t_{pi})^2 \tag{5.1}$$

o_{max} e o_{min} são o máximo e mínimo para cada neurônio de saída. o_{pi} e t_{pi} são a saída e a referência, respectivamente, para o p-ésimo exemplo no i-ésimo neurônio de saída. P é o número de vetores de exemplo e N o número de neurônios de saída.

O erro percentual de classificação dos padrões, E_C, é calculado fazendo a proporção entre o número de erros e o número de vetores do conjunto de dados.

O Neural Network Toolbox™do Matlab [Demuth et al. 2008] foi utilizado para treinar RNAs para os problemas do *benchmark*, com algumas topologias iguais e outras diferentes das sugeridas em [Prechelt 1994]. Mesmo quando usando topologias diferentes, os resultados do treinamento foram similares aos de [Prechelt 1994], como pode ser visto

na tabela 5.1. O algoritmo Levenberg-Marquardt *backpropagation* [Demuth et al. 2008], também baseado no *backpropagation*, foi utilizado no treinamento em Matlab.

Para todas as topologias simuladas, a função tansig foi utilizada como função de ativação das camadas intermediárias e a função de ativação linear (saída igual à entrada) na saída. O método de classificação utilizado foi o WTA, em que o padrão reconhecido é definido escolhendo a saída com maior valor.

Tabela 5.1: Erro no Proben1 e no Matlab

Conjunto	Proben1			Matlab		
	Topologia	E_Q	E_C	Topologia	E_Q	E_C
Mushroom1	125x2	0.014	0.00	125x2	0.00	0.00
Diabetes2	8x16x8x2	17.46	23.44	8x24x2	19.24	26.04
Gene3	120x4x2x3	9.41	13.62	120x4x2x3	1.08	0.88
Horse1	58x4x3	13.38	26.37	58x4x3	6.35	4.95

Depois de treinadas as RNAs, os pesos sinápticos e os dados são convertidos para ponto fixo para serem utilizados no neuroprocessador. A simulação do RTL foi realizada utilizando esses dados, onde puderam ser medidos os erros introduzidos pelo cálculo em ponto fixo. Os resultados das simulações em ponto flutuante de 64 bits no Matlab e em ponto fixo de 16 bits no RTL são exibidos na tabela 5.2. Note que o aumento do erro de classificação é muito baixo (menor que 0.3%) se comparado com o erro no cálculo em ponto flutuante.

Tabela 5.2: Erro introduzido com o cálculo em ponto fixo de 16 bits no RTL

Conjunto	Topologia	Matlab	RTL	Aumento do erro	
		E_C	E_C	Absoluto	Percentual
Mushroom1	125x2	0.00	0.00	0	0.00
Diabetes2	8x24x2	26.04	26.30	2	0.26
Gene3	120x4x2x3	0.88	0.88	0	0.00
Horse1	58x4x3	4.95	4.95	0	0.00

5.2 Análise de Desempenho

Definir métricas para avaliar o desempenho de arquiteturas de *hardware* é uma tarefa difícil, mesmo para processadores de propósito gerais. Até as medidas de desempenho mais consolidadas, como o MIPS, têm falhas de generalização de uso e eficácias questionáveis, como abordado em [Patterson e Hennessy 2005]. Outra dificuldade dessas métricas está em conseguir avaliar a arquitetura independentemente da implementação, ou seja, sem que o número de unidades de processamento ou de bits de precisão usados mascarem a ineficiência do projeto.

A avaliação de *neurohardware* sofre das mesmas dificuldades encontradas para os processadores de propósito gerais. Da mesma forma, o Milhões de Conexões por Segundo

(MCPS), que é uma métrica amplamente utilizada em diversos trabalhos de *hardware* para redes neurais, apresenta os mesmo problemas de generalização encontrados com o MIPS. O MCPS é a medida de quantas operações de multiplicar e acumular são computadas por unidade de tempo. [Ienne 1993] aborda a ineficácia de utilizar somente o MCPS como figura de mérito e apresenta outras métricas relevantes na avaliação da arquitetura.

Dentre as métricas apresentadas em [Ienne 1993], as mais importantes são:

- Latência: Número de ciclos de relógio entre a apresentação do primeiro dado de entrada e a aparição da primeira saída;
- MCPS: Conforme descrito anteriormente;
- Throughput: Número de padrões de entrada processados por ciclo de relógio. Com exceção da virtualização, apresenta resultados melhores do que o inverso da latência, por conta dos estágios em pipeline do sistema;
- Eficiência da paralelização (n_Q): Mostra quão proveitoso é o paralelismo do *hardware*. O cálculo da eficiência da paralelização n_Q é definido na equação 5.2.

$$n_Q = \frac{throughput\_do\_sistema}{numero\_de\_up \cdot throughput\_up} \tag{5.2}$$

Onde $throughput\_do\_sistema$ é o throughput do sistema completo, $numero\_de\_up$ é o número de UPs da implementação e $throughput\_up$ é o throughput individual da UP.

Em termos qualitativos, a latência visa avaliar o tempo de processamento de um vetor de entrada individualmente. O MCPS e o throughput avaliam a quantidade de informação processada por unidade de tempo, diferenciados apenas pela quantidade de dados considerada. Pode haver ou não uma relação linear entre MCPS e throughput, a depender da arquitetura. Por fim, a eficiência da paralelização é uma tentativa de avaliar a qualidade da arquitetura, sem a influência do número de UPs utilizadas.

Entretanto, a medição da eficiência de paralelização apresentada na equação 5.2 também não contempla todas as arquiteturas, uma vez que se torna difícil definir o throughput de cada UP quando existe dependência do número de UPs no desempenho de cada UP, como no caso do neuroprocessador proposto em que a função de ativação é compartilhada. Além disso, para comparar de forma justa o trabalho desenvolvido com outros *neurohardwares* nos quais a única informação de desempenho disponível é o MCPS, faz-se necessário o uso desta informação para obter uma métrica do paralelismo do sistema. Esta métrica deve fornecer uma medida da capacidade do sistema, independente do número de UPs e da freqüência de trabalho utilizados na implementação. Alguns trabalhos apresentam métricas com alguns destas características, como em [van Keulen et al. 1994], mas nenhum atende completamente os requisitos citados acima.

Portanto, uma nova métrica é apresentada neste trabalho, chamada Conexões por Ciclo por Unidade de Processamento (CPCPU), proveniente da normalização do CPS com relação ao número de UPs e colocando-o em termos de ciclos de relógio, no lugar de uma unidade de tempo. A expressão da CPCPU é definida na equação 5.3.

$$CPCPU = \frac{CPS}{frequencia \cdot numero\_de\_up} \tag{5.3}$$

Onde $frequencia$ é a freqüência de operação do sistema.

Sendo uma medição indireta de desempenho, proveniente do CPS, o CPCPU apresentará os mesmos problemas de generalização de uso que o CPS, como descrito em [Ienne 1993]. Não obstante, o CPCPU é uma métrica boa para avaliar e comparar a eficiência da arquitetura em si, sem influência do número de UPs instanciadas ou da frequência de trabalho.

Sistemas desenvolvidos utilizando técnicas de projeto que garantem a escalabilidade do hardware e a portabilidade com relação à tecnologia de fabricação necessitam de uma avaliação que indique a qualidade da arquitetura, sem pesar as escolhas de uma implementação específica. Além disso, faz-se necessário avaliar se esta qualidade se mantém para diversos casos de implementação, quando o número de UPs, a frequência e qualquer outra característica variam. O CPCPU também pode ser utilizado para esta avaliação da uniformidade da arquitetura nas diversas situações.

É importante observar que esta métrica só é aplicável para sistemas cuja partição do processamento é feita por neurônio, não sendo útil em sistemas particionados com relação às sinapses. Isso devido ao fato do CPCPU ter como base o CPS, que baseia-se no número de unidades processadoras de neurônios. Utilizando a mesma idéia do CPCPU, outra métrica pode ser desenvolvida para sistemas orientados à sinopse.

O CPCPU de uma arquitetura pode variar de zero a infinito. Com o CPCPU igual a um, uma conexão é processada a cada ciclo de relógio por UP. Valores menores que um indicam que, a cada ciclo de relógio, menos de uma conexão é processada por UP, enquanto que CPCPU maiores que um podem indicar que mais de um padrão está sendo processado por vez.

Utilizando a latência, o throughput, o MCPS e o CPCPU, o neuroprocessador foi avaliado em algumas configurações de topologias e modos de funcionamento.

A tabela 5.3 exibe os parâmetros de desempenho para o neuroprocessador operando no modo desempenho. A escolha entre núcleo simples em um *kernel*, núcleo duplo em um *kernel* ou núcleo triplo em dois *kernels*, bem como o número de UPs, foi baseada na topologia de RNA utilizada.

Tabela 5.3: Análise de desempenho para simulações no modo desempenho

Conjunto	Topologia	N^o de padrões	Latência	Throughput	MCPS	CPCPU
Mushroom1	125x2	8124	132	0.0077	192.31	0.9615
Diabetes2	8x24x2	768	61	0.0345	826.99	0.3181
Gene3	120x4x2x3	3175	146	0.0080	395.18	0.4391
Horse1	58x4x3	364	75	0.0159	387.10	0.5530

As mesmas topologias de RNA apresentadas na tabela 5.3 para o modo desempenho também foram simuladas para o modo área, ou seja, usando virtualização. Os resultados desta simulação são apresentados na tabela 5.4. Note que, apesar de sempre haver redução do número de unidades de processamento, essa redução pode não ser significativa caso a topologia possua muitos pesos, onde a área ocupada por estes predominará na área do chip. Um exemplo disso seria uma topologia com muitos nós de entrada.

Tabela 5.4: Análise de desempenho para simulações no modo área com 8 UPs

Conjunto	Topologia	N^o de padrões	Latência	Throughput	MCPS	CPCPU
Mushroom1	125x2	8124	133	0.0072	181.16	0.9058
Diabetes2	8x24x2	768	161	0.0060	144.58	0.0556
Gene3	120x4x2x3	3175	172	0.0055	269.95	0.2999
Horse1	58x4x3	364	89	0.0102	248.98	0.3557

Uma comparação entre o neuroprocessador proposto e alguns *neurohardwares* encontrados na literatura e disponíveis comercialmente é apresentada na tabela 5.5. Apesar de pouco informativo, o MCPS foi a única fonte de informação de desempenho utilizada na comparação uma vez que esta é a única informação disponibilizada na maioria dos trabalhos publicados e nas especificações de fabricantes. A partir do MCPS, foi calculado o CPCPU, para obter uma informação da qualidade da paralelização.

Tabela 5.5: Comparações do neuroprocessador proposto com outros *neurohardwares*

Neurohardware	N^o de UPs	Freqüência (MHz)	MCPS	CPCPU
Philips Lneuro-2.3 [Duranton 1996]	12	60	720	1.000
Neuricam NC3001 [Neuricam 1999]	32	30	1000	1.042
RC Module NM6403 [Chevtchenko et al. 1998]	64	50	1200	0.375
SAND/1 [Dias et al. 2003]	4	50	200	1.000
CogniMem_1k [CogniMem 2008]	1024	27	80000	2.894
KARN [Rajah e Hani 2004]	100	100	5079	0.508
RAPTOR 2000 [Omondi e Rajapakse 2006]	384	105	38877	0.964
Pentium IV [Omondi e Rajapakse 2006]	2	2400	205	0.043
Neuroprocessador Proposto	128	100	6348.4	0.934

Os resultados de desempenho do processamento de uma topologia de duas camadas foi adotada na comparação. A configuração escolhida foi 128x64x4 com saída linear, um caso possível de configuração para reconhecimento de padrões. Pela tabela 5.5 pode ser observado que, além da flexibilidade, o *hardware* proposto possui também um bom desempenho, próximo ao apresentado pelos *neurohardwares* apresentados na tabela, com exceção de [CogniMem 2008], que é o neuroprocessador de melhor desempenho disponível comercialmente até a presente data.

5.3 Características Físicas

Para avaliar a influência da variação dos parâmetros do neuroprocessador nas características físicas do projeto, foram realizadas diversas sínteses lógicas. Como a síntese fornece uma estimativa da potência, área, quantidade de portas lógicas e temporização (atrasos), é possível analisar como estas características variam de acordo com a variação dos parâmetros. Foi utilizada uma biblioteca de células na tecnologia CMOS de 90nm da SMIC e o Encounter RTL Compiler, ferramenta de síntese da Cadence.

A tabela 5.6 e o gráfico da figura 5.1 mostram a variação do número de UPs em diversas sínteses realizadas para a frequência de 100 MHz, precisão de 16 bits e com núcleo duplo. A partir do gráfico da figura 5.1, pode ser percebido que a relação da potência e da área em função do número de UPs é quase linear. Estes resultados são referentes apenas ao caminho de dados e às logicas de controle, visto que os blocos de memória não são gerados durante a síntese.

Tabela 5.6: Sínteses para diversos números de UPs

N^o de UPs	Portas Lógicas	Slack (ns)	Potencia Estática (mW)	Potência Dinâmica (mW)	Potência Total (mW)	Área (μm^2)
8	14795	891.3	0.41	6.41	6.81	9608.8
16	24123	1013.7	0.65	9.19	9.84	14657
32	42979	409.0	1.13	12.28	13.41	24859
64	80743	175.9	2.09	19.34	21.43	45126
128	155318	375.2	4.01	37.00	41.00	85347

A tabela 5.7 apresenta os resultados da síntese na qual foi obtida a maior frequência de operação suportada pelo neuroprocessador, fixando o número de UPs em 128. A frequência máxima obtida foi em 200MHz e foi encontrada após diversas tentativas de síntese com frequências altas, descrescendo o ciclo do relógio até conseguir fazer o sistema funcionar sem problemas de *Setup and Hold*. As análises de *Setup and Hold* são realizadas pela ferramenta de síntese. Melhores resultados podem ser obtidos para outros números de UPs.

Tabela 5.7: Resultados da síntese para maior frequência de operação suportada

Característica	Resultado
Portas Lógicas	188390
Slack	0.0 ns
Potencia Estática	4.36 mW
Potência Dinâmica	53.16 mW
Potência Total	57.52 mW
Área	90612 μm^2

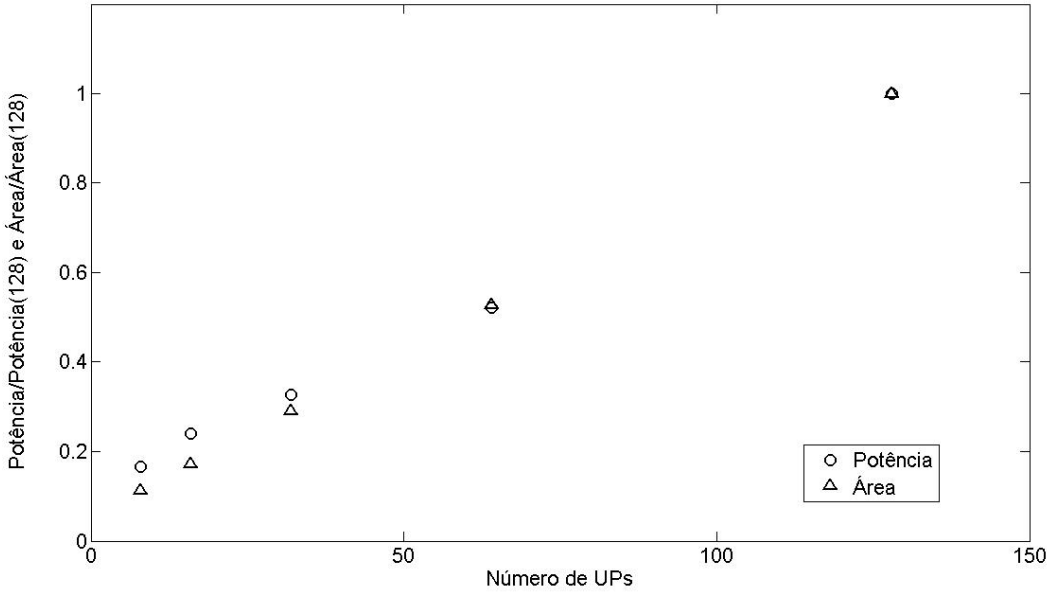


Figura 5.1: Área e potência em função do número de UPs

Capítulo 6

Conclusões

Neste trabalho a concepção da arquitetura de *hardware* e o projeto em circuito integrado de um neuroprocessador flexível para redes do tipo MLP foram apresentados. Desempenho, escalabilidade, flexibilidade e portabilidade foram as principais questões consideradas durante o projeto, o qual foi desenvolvido como um núcleo de IP, permitindo a integração em dispositivos de aplicação específica ou em SoCs.

O *kernel*, que é o módulo fundamental de processamento do sistema, é constituído por um módulo de controle, um caminho de dados e um módulo de memórias. O caminho de dados possibilita a seleção dos recursos utilizados em cada implementação, tais como o número de UPs e o número de núcleos de processamento. No caminho de dados, pode ser usado um núcleo, para processar uma camada por vez, ou dois núcleos, para processar duas camadas simultaneamente.

No caso da aplicação demandar três camadas processadas simultaneamente, dois *kernels* podem ser concatenados para atingir esse requisito de desempenho. Em casos de virtualização, o *kernel* é reutilizado visando processar a RNA parte por parte.

A organização de memória apresentada mostra vantagens com relação às possibilidades de configurações de topologias de RNA, que podem variar consideravelmente sem consequências para as UPs. Outra vantagem é que, por conta da divisão da memória em blocos, não existirão blocos muito extensos de modo a dificultar o *layout*. Desvantagens são percebidas para a prototipagem do *hardware* em FPGA, podendo as memórias armazenadas em um único módulo dificultar a utilização dos recursos.

O desempenho do neuroprocessador foi avaliado, utilizando métricas encontradas na literatura e uma nova métrica apresentada neste trabalho, a CPCPU. Esta métrica mostrou-se apropriada para comparar a eficiência das arquiteturas sem a influência do número de UPs utilizadas e da frequência de operação. Também foi realizada uma comparação entre o neuroprocessador proposto e outros encontrados na literatura. Mesmo não conseguindo superar o melhor dos neuroprocessadores encontrados, o *hardware* proposto figurou entre os melhores. Como também há requisitos de flexibilidade e área, a arquitetura não pôde ser projetada para atingir o máximo do desempenho, apresentando algumas limitações de recursos.

Um exemplo de *layout* do neuroprocessador foi realizado para avaliar a viabilidade física do projeto. A área útil utilizada foi de 4mm x 4mm para uma versão do neuroprocessador com 128 UPs, precisão de 16 bits para dados e pesos sinápticos, memória de LUT com 6144 posições, dois núcleos e 16384 palavras de memória de pesos (128 blocos

de 128 palavras). Para esta implementação, o projeto mostrou-se plenamente factível, não apresentando problemas de congestionamento.

Finalmente, sugere-se um estudo continuado da arquitetura, visando reduzir o impacto das limitações de recursos no desempenho do sistema, sendo esta uma das maiores dificuldades encontradas no desenvolvimento do trabalho. A aplicação de técnicas para redução do consumo pode ser incluída na continuação do projeto. Também são sugeridas a implementação do *hardware* em CI e a utilização em uma aplicação de reconhecimento de padrões.

Referências Bibliográficas

Altera (2009), *Avalon Streaming Interface*, Altera Corporation, San Jose, USA.

Bergeron, Janick (2000), *Writing Testbenches - Functional Verification of HDL Models*, Springer.

Bishop, Christopher M. (1995), *Neural Networks for Pattern Recognition*, Clarendon Press, Birmingham, UK.

Chevtchenko, P.A., D.V. Fomine, V.M. Tchernikov, e P.E. Vixne (1998), 'Using of microprocessor nm6403 for neural net emulation', *Workshop on virtual intelligence / dynamic neural networks* .

CogniMem (2008), *CogniMem Evaluation Base board*, Recognetics Inc, Loveland, USA.

Demuth, Howard, Mark Beale e Martin Hagan (2008), *Neural Network Toolbox™6: User's Guide*, The MathWorks™, Natick, MA.

Dias, Fernando Morgado, Ana Antunes e Alexandre Manuel Mota (2003), 'Artificial neural networks: a review of commercial hardware'.

Duranton, Marc (1996), L-neuro 2.3: a vlsi for image processing by neural networks, *em* 'Microelectronics for Neural Networks, 1996., Proceedings of Fifth International Conference on', IEEE Computer Society, pp. 157–160.

Girau, Bernard (2000), 'Fpna: Interaction between fpga and neural computation', *International Journal of Neural Systems* **10**(3), 243–259.

Haykin, Simon (1999), *Neural Networks - A Comprehensive Foundation*, Prenctice Hall, Hamilton, Ontario, Canada.

Heemskerk, Jan N. H. (1995), 'Overview of neural hardware', Capítulo 3 em Neurocomputers for Brain-Style Processing. Design, Implementation and Application, PhD Thesis. Disponível via ftp: `ftp.mrc-apu.cam.ac.uk/pub/nn/murre/neurhard.ps`.

Ienne, Paolo (1993), Architectures for neuro-computers: Review and performance evaluation, Relatório Técnico 93/21, Swiss Federal Institute of Technology, Lausanne.
URL: *http://ftp.funet.fi/pub/sci/neural/neuroprose/ienne.nnarch.ps.Z*

Ienne, Paolo (1997), Digital connectionist hardware: Current problems and future challenges, *em* J.Mira, R.Moreno-Díaz e J.Cabestany, eds., 'Biological and Artificial Computation: From Neuroscience to Technology', Springer, Berlin, pp. 688–713.

Keating, Michael e Pierre Bricaud (2002), *Reuse Methodology Manual for SoC Designs*, Kluwer Academic Publishers, New York.

Kemsley, D. H., T. R. Martinez e D. M. Campbell (1992), 'A survey of neural network research and fielded applications', *In International Journal of Neural Networks: Research and Applications* pp. 123–133.

Lindsey, Clark S. e Thomas Lindblad (1998), 'Review of hardware neural networks: A user's perspective', Disponível em `http://www1.cern.ch/NeuralNets/nnwInHepHard.html`.

Madani, Kurosh (2006), Industrial and real world applications of artificial neural networks: Illusion or reality?, *em* J.Braz, H.Araújo, A.Vieira e B.Encarnação, eds., 'Informatics in Control, Automation and Robotics I', Springer, Netherlands, pp. 11–26.

McCulloch, W. S. e W. Pitts (1943), 'A logical calculus of the ideas immanent in nervous activity', *Bulletin of Mathematical Biophysics* pp. 127–147.

Minsky, M. e S. Papert (1969), *Perceptrons*, MIT Press, Cambridge.

Neuricam (1999), *NC3001 TOTEM - Digital Processor for Neural Networks - Data Sheet*, Neuricam, Trento, Italy.

Omondi, Amos R. e Jagath C. Rajapakse (2006), *FPGA implementation of neural networks*, Springer, Dordrecht.

Patterson, David A. e John L. Hennessy (2005), *Computer Organization and Design: The Hardware/Software Interface*, Elsevier, San Franciso, USA.

Prechelt, Lutz (1994), Proben1 - a set of neural network benchmark problems and benchmarking rules, Relatório técnico, Universitat Karlsruhe, Karlsruhe.
URL: *http://page.mi.fu-berlin.de/~prechelt/Biblio/1994-21.ps.gz*

Rajah, Avinash e Mohamed Khalil Hani (2004), 'Asic design of a kohonen neural network microchip', *ICSE2004 Proceedings* .

Rosenblatt, Frank (1962), *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Spartan Books, New York.

Rumelhart, D. E., G. E. Hinton e R. J. Williams (1986), 'Learning representations by back-propagating errors', *Nature* pp. 533–536.

Samarasinghe, Sandhya (2006), *Neural Networks for Applied Sciences and Engineering*, Taylor and Francis Group, New York.

Scheffer, Louis, Luciano Lavagno e Grant Martin (2006), *EDA for IC Implementation, Circuit Design, and Process Technology*, CRC Press, New York.

van Keulen, Edwin, Sel Colak, Heini Withagen e Hans Hegt (1994), 'Neural network hardware performance criteria', *IEEE International Conference on Neural Networks* **3**, 1955–1958.

Vitabile, S., V. Conti, F. Gennaro e F. Sorbello (2005), 'Efficient mlp digital implementation on fpga', *8th Euromicro conference on Digital System Design* .

Widrow, Bernard, David E. Rumelhart e Michael A. Lehr (1994), 'Neural networks: Applications in industry business and science', *Communications of ACM* **27**(3).

Xiu, Liming (2007), *VLSI Circuit Design Methodology Demystified: A Conceptual Taxonomy*, IEEE Press, Hoboken.

Yun, Seok Bae, Young Joo Kim, Sung So0 Dong e Chong Ho Lee (2002), 'Hardware implementation of neural network with expansible and reconfigurable architecture', *9th International Conference on Neural Infomation Processing* **2**.

Apêndice A

Código-fonte em Verilog da UP

Este apêndice apresenta o código-fonte em Verilog da UP do neuroprocessador apresentado, conforme o esquemático da figura 4.9.

```
//    module mac
//    autor Igor Dantas
//    versao 1.0
//    data 20-01-2009
//    Este modulo executa o processamento individual de um neuronio
//    fazendo as multiplicacoes e somas relativas. Nao inclui a
//    funcao de ativacao.

`timescale 1ns/1ns

module mac
#(parameter MEM_WIDTH = 8,
parameter DATA_WIDTH = 8,
parameter MAX_N_INP = 511,
parameter ACTV_INPUT_WIDTH = 8)

(input clk,
input rst_n,
input sink_sop,
input sink_valid,
input update_out,
input signed [DATA_WIDTH-1:0] data_in,
input signed [MEM_WIDTH-1:0] mem_data,
output reg signed [ACTV_INPUT_WIDTH-1:0] data_out,
input [5:0] acc_max_width);

localparam ACC_WIDTH = log2(MAX_N_INP+1)+MEM_WIDTH+DATA_WIDTH-1;

// Sinais internos
reg signed [ACC_WIDTH-1:0] acc;
```

```
31 reg signed [DATA_WIDTH+MEM_WIDTH-1:0] mult;
32 reg bypass_sum;
33 reg sink_valid_1dly;
34 reg sink_valid_2dly;
35 wire signed [ACC_WIDTH-1:0] next_data_out;
36
37 // Truncando dados para enviar para activaction function
38 assign next_data_out = acc >>> (acc_max_width-ACTV_INPUT_WIDTH);
39
40 // Unidade de multiplicacao
41 always @ (posedge clk) begin
42    if(~rst_n)
43       mult <= 0;
44    else if(sink_valid | sink_valid_1dly) begin
45       if(sink_sop)
46          mult <= mem_data * (2**(DATA_WIDTH-1)-1);
47       else
48          mult <= mem_data * data_in;
49    end
50 end
51
52 // Acumulador
53 always @ (posedge clk) begin
54    if(~rst_n)
55       acc <= 0;
56    else if(sink_valid | sink_valid_2dly) begin
57       if(bypass_sum)
58          acc <= mult;
59       else
60          acc <= acc + mult;
61    end
62 end
63
64 // Registrador de saida. Atualizado apenas
65 // no fim da operacao
66 always @ (posedge clk) begin
67    if(~rst_n)
68       data_out <= 0;
69    else if(update_out)
70          data_out <= next_data_out[ACTV_INPUT_WIDTH-1:0];
71 end
72
73 always @ (posedge clk) begin
74    if(~rst_n) begin
```

```
75          bypass_sum <= 0;
76          sink_valid_1dly <= 0;
77          sink_valid_2dly <= 0;
78      end
79      else begin
80          if(sink_valid)
81              bypass_sum <= sink_sop;
82
83          sink_valid_1dly <= sink_valid;
84          sink_valid_2dly <= sink_valid_1dly;
85      end
86  end
87
88  `include log2.v
89
90  endmodule
```

Apêndice B

Folha de Especificações de Bloco de Memória

A seguir, a folha de especificações para uma memória de duas portas com 128 palavras de 16 bits, criada pelo gerador de memórias da SMIC para a tecnolgia CMOS de 90nm. Essa memória foi utilizada na implementação de exemplo apresentada no capítulo 4. A folha de especificações também foi criada pelo gerador de memória.

Dual-Port Synchronous SRAM
128 Words X 16 Bits, Mux 4 Instance
SMIC LOGIC90G 90nm Process

Overview

The dual-port synchronous SRAM is optimized for speed and density. The memory is designed to take full advantage of SMIC's 90nm LOGIC90G CMOS process.

The storage array is composed of six-transistor bit cells with fully static circuitry. The SRAM operates at a voltage of 0.9V to 1.1V and a junction temperature range of -40°C to 125°C.

Instance Settings

Parameter	Setting
Instance Name	sram_dp_128
Process	LOGIC90G
Words	128
Bits	16
Mux	4
Write Mask	off
Extra Margin Adjustment	on
Redundancy	off
Soft Error Repair	none
BIST Muxes	off
Output Drive	6
Power Routing Type	rings
Ring Width	2µm
Horizontal Ring Layer	MET3
Vertical Ring Layer	MET4
Top Metal	MET5-9
Frequency	1.0 MHz

Description

The dual-port synchronous RAM is a fully static memory with write enable (WENA, WENB), chip enable (CENA, CENB), address (AA, AB), data in (DA, DB) and data out (QA, QB) pins. The RAM is self-timed and consumes the minimum amount of power for read or write operations.

All synchronous inputs are latched on the rising-edge of the clock signal. When CENA is low and WENA is high the memory will read. When CENA and WENA are both low the word on the DA will be written to the memory and it will appear at the outputs (write-through).

When CENA is high the memory is deselected and forced into a low-power standby mode. Stored data is fully retained but memory access is disabled for data read or data write, the existing data outputs continue to drive their previous values.

The Extra Margin Adjustment allows you to adjust the width of the self timing pulse.

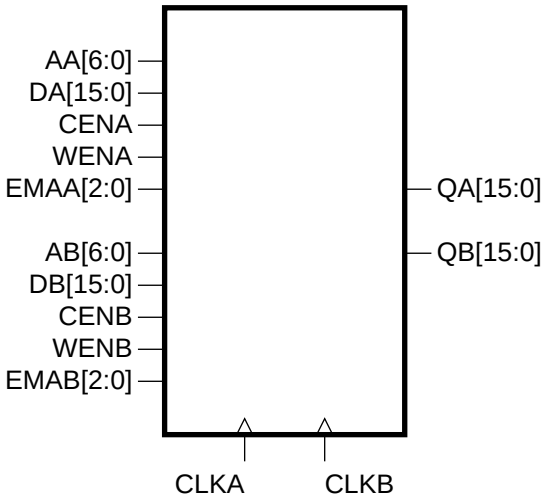
Refer to the users manual for a more detailed description of memory operation.

Physical Dimensions (units = µm)

Parameter	Size
Core Width	236.5
Core Height	134.4
Footprint Width	254.2
Footprint Height	152.2

The footprint area includes the core area and user defined power routing and pin spacing.

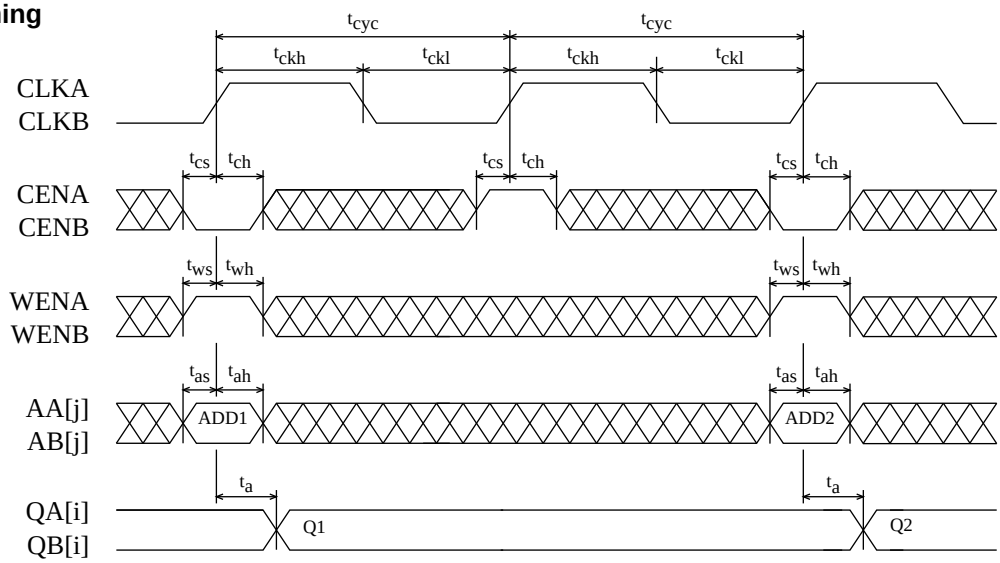
Symbol



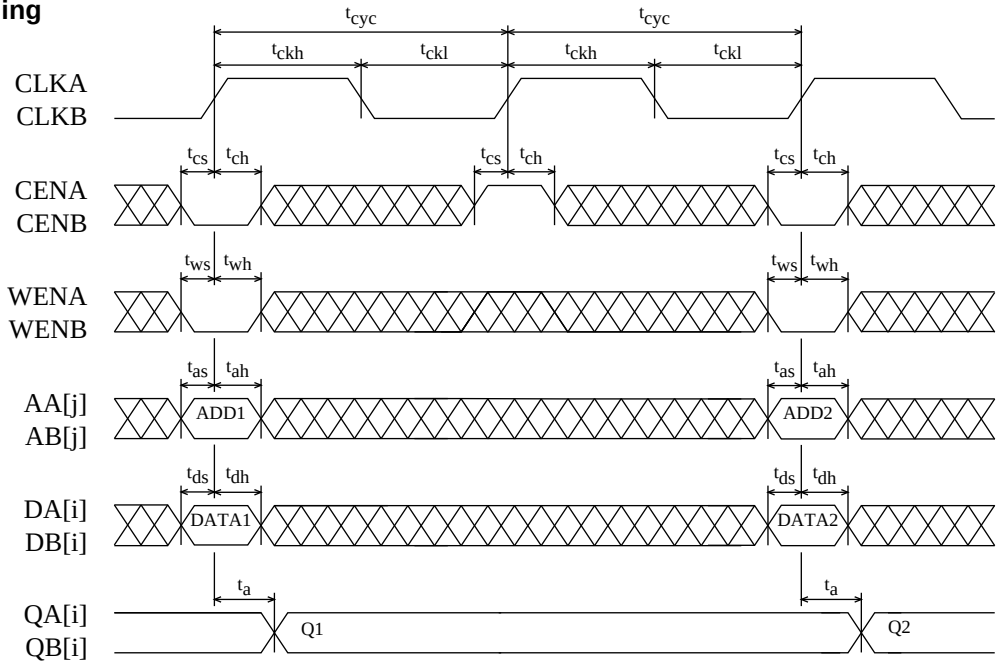
Pin Description

Pin	Description
AA[6:0], AB[6:0]	Port A & B Addresses (AA[0],AB[0] = LSB)
DA[15:0], DB[15:0]	Port A & B Data Inputs (DA[0],DB[0] = LSB)
CLKA, CLKB	Port A & B Clocks
CENA, CENB	Port A & B Chip Enables
WENA,WENB	Port A & B Write Enables (Active low)
QA[15:0], QB[15:0]	Port A & B Data Outputs (QA[0],QB[0] = LSB)
EMAA[2:0], EMAB[2:0]	Port A & B Margin Adjustment (EMAA[0],EMAB[0] = LSB)

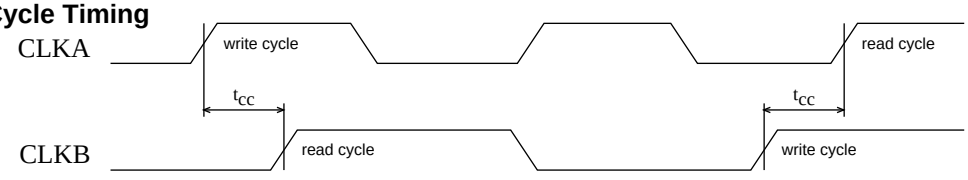
Read Cycle Timing



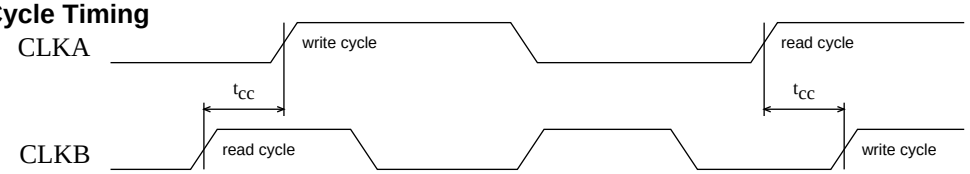
Write Cycle Timing



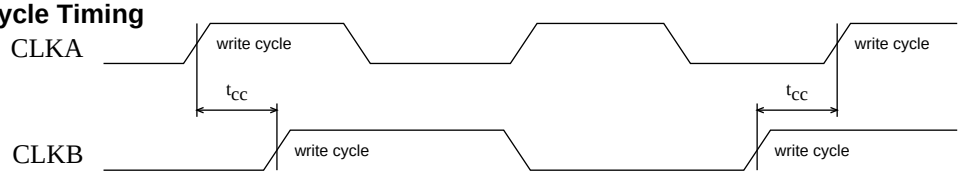
Write to Read Cycle Timing



Read to Write Cycle Timing



Write to Write Cycle Timing



Timing (units = ns)

The timing tables show values measured from the output threshold to the input threshold. The input pins are driven by standard slews. The slews and thresholds vary depending upon the process corner.
The timing tables values are applicable to both A and B ports of the memory even though only the A side is shown.

Pin	Symbol	Fast Process 1.1V, -40°C		Fast Process 1.1V, 125°C		Typical Process 1.0V, 25°C		Slow Process 0.9V, 125°C	
		Min	Max	Min	Max	Min	Max	Min	Max
Cycle time	t_{cyc0}	0.649		0.784		0.993		1.687	
Access time[1,2]	t_{a0}	0.514		0.619			0.936		1.610
Address setup	t_{as}	0.206		0.260		0.332		0.575	
Address hold	t_{ah}	0.000		0.000		0.000		0.000	
Data setup	t_{ds}	0.106		0.121		0.152		0.271	
Data hold	t_{dh}	0.000		0.000		0.000		0.000	
Chip enable setup	t_{cs}	0.221		0.265		0.335		0.602	
Chip enable hold	t_{ch}	0.000		0.000		0.000		0.000	
Write enable setup	t_{ws}	0.211		0.259		0.302		0.497	
Write enable hold	t_{wh}	0.000		0.000		0.000		0.000	
Clock high	t_{ckh}	0.038		0.044		0.062		0.093	
Clock low	t_{ckl}	0.226		0.267		0.366		0.677	
Clock rise slew	t_{ckr}		1.000		1.000		1.000		1.000
Output load factor[3]	K_{load}		0.441		0.516		0.754		1.170

[1] Output delays and a load dependency (Kload) which is used to calculate: *TotalDelay = FixedDelay + (Kload x Cload).*
[2] Access time is defined as the longest possible delay to valid output for the typical and slow corners, and the shortest possible delay for the fast corners.
[3] The output load factor units are ns/pF.

Cycle and Access Timing for Different Values of Extra Margin Adjustment (units = ns)

Pin	Symbol	Fast Process 1.1V, -40°C		Fast Process 1.1V, 125°C		Typical Process 1.0V, 25°C		Slow Process 0.9V, 125°C	
		Min	Max	Min	Max	Min	Max	Min	Max
Cycle time EMAA=0	t_{cyc0}	0.649		0.784		0.993		1.687	
Cycle time EMAA=1	t_{cyc1}	0.799		0.967		1.227		2.096	
Cycle time EMAA=2	t_{cyc2}	0.946		1.150		1.460		2.540	
Cycle time EMAA=3	t_{cyc3}	1.016		1.232		1.574		2.764	
Cycle time EMAA=4	t_{cyc4}	**		**		**		**	
Cycle time EMAA=5	t_{cyc5}	**		**		**		**	
Cycle time EMAA=6	t_{cyc6}	**		**		**		**	
Cycle time EMAA=7	t_{cyc7}	**		**		**		**	
Access time EMAA=0	t_{a0}	0.514		0.619			0.936		1.610
Access time EMAA=1	t_{a1}	0.664		0.802			1.171		2.019
Access time EMAA=2	t_{a2}	0.812		0.985			1.404		2.463
Access time EMAA=3	t_{a3}	0.881		1.067			1.517		2.687
Access time EMAA=4	t_{a4}	**		**			**		**
Access time EMAA=5	t_{a5}	**		**			**		**
Access time EMAA=6	t_{a6}	**		**			**		**
Access time EMAA=7	t_{a7}	**		**			**		**
EMAA setup	t_{emas}	0.649		0.784		0.993		1.687	
EMAA hold	t_{emah}	0.649		0.784		0.993		1.687	

**Illegal setting of EMAA for this corner.

Pin Capacitance (units = fF)

Pin	Fast Process 1.1V, -40°C	Fast Process 1.1V, 125°C	Typical Process 1.0V, 25°C	Slow Process 0.9V, 125°C
AA,AB	58.420	59.430	57.630	58.680
DA,DB	1.638	1.662	1.589	1.591
CLKA,CLKB	117.300	114.300	111.900	109.300
CENA,CENB	39.700	40.250	39.650	40.970
WENA,WENB	45.520	46.190	45.380	46.780
EMAA,EMAB	35.900	36.390	35.760	36.820

Power (current units = mA)

Pin	Fast Process 1.1V, -40°C	Fast Process 1.1V, 125°C	Typical Process 1.0V, 25°C	Slow Process 0.9V, 125°C
AC Current (EMAA=0)[1,4]	9.87E-3	1.31E-2	8.43E-3	7.57E-3
AC Current (EMAA=1)[1,4]	9.99E-3	1.31E-2	8.52E-3	7.63E-3
AC Current (EMAA=2)[1,4]	1.02E-2	1.33E-2	8.67E-3	7.74E-3
AC Current (EMAA=3)[1,4]	1.03E-2	1.33E-2	8.75E-3	7.79E-3
AC Current (EMAA=4)[1,4]	1.04E-2	1.35E-2	8.94E-3	7.91E-3
AC Current (EMAA=5)[1,4]	1.05E-2	1.36E-2	9.03E-3	8.01E-3
AC Current (EMAA=6)[1,4]	1.06E-2	1.36E-2	9.07E-3	8.08E-3
AC Current (EMAA=7)[1,4]	1.06E-2	1.36E-2	9.10E-3	8.14E-3
Read AC Current (EMAA=0)[4]	7.64E-3	1.09E-2	6.54E-3	5.93E-3
Read AC Current (EMAA=1)[4]	7.69E-3	1.09E-2	6.58E-3	5.97E-3
Read AC Current (EMAA=2)[4]	7.75E-3	1.09E-2	6.63E-3	6.00E-3
Read AC Current (EMAA=3)[4]	7.78E-3	1.09E-2	6.66E-3	6.02E-3
Read AC Current (EMAA=4)[4]	7.82E-3	1.10E-2	6.72E-3	6.06E-3
Read AC Current (EMAA=5)[4]	7.85E-3	1.10E-2	6.74E-3	6.10E-3
Read AC Current (EMAA=6)[4]	7.86E-3	1.10E-2	6.76E-3	6.12E-3
Read AC Current (EMAA=7)[4]	7.86E-3	1.10E-2	6.76E-3	6.14E-3
Write AC Current (EMAA=0)[4]	1.21E-2	1.53E-2	1.03E-2	9.20E-3
Write AC Current (EMAA=1)[4]	1.23E-2	1.54E-2	1.05E-2	9.29E-3
Write AC Current (EMAA=2)[4]	1.26E-2	1.56E-2	1.07E-2	9.48E-3
Write AC Current (EMAA=3)[4]	1.27E-2	1.57E-2	1.08E-2	9.56E-3
Write AC Current (EMAA=4)[4]	1.30E-2	1.60E-2	1.12E-2	9.76E-3
Write AC Current (EMAA=5)[4]	1.32E-2	1.62E-2	1.13E-2	9.93E-3
Write AC Current (EMAA=6)[4]	1.33E-2	1.62E-2	1.14E-2	1.00E-2
Write AC Current (EMAA=7)[4]	1.34E-2	1.63E-2	1.14E-2	1.01E-2
Peak Current	53.28	45.49	33.13	18.58
Deselected Current[2,4]	3.02E-3	5.08E-3	2.58E-3	2.32E-3
Standby Current[3]	5.47E-2	4.18	6.05E-2	2.33E-1

** Illegal setting of EMAA for this corner.

1 The AC current value assumes 50% read and write operations, where all addresses and 50% of input and output pins switch at the user defined frequency of 1.0MHz. It is assumed that EMAA pins do not switch.

2 The deselected current assumes the memory is deselected, all addresses switch, and 50% of input pins switch at the user defined frequency of 1.0MHz. The logic switching component of deselected power becomes negligibly small if the input pins are held stable by externally controlling these signals with chip select. It is assumed that EMAA pins do not switch.

3 The standby current value is independent of frequency and assumes all inputs and outputs are stable.

4 The standby current component is not included in this value.

Clock Noise Limit

Symbol	Fast Process 1.1V, -40°C		Fast Process 1.1V, 125°C		Typical Process 1.0V, 25°C		Slow Process 0.9V, 125°C	
	Pulse Width	Voltage	Pulse Width	Voltage	Pulse Width	Voltage	Pulse Width	Voltage
CLKA, CLKB	10.0ns	0.3V	10.0ns	0.2V	10.0ns	0.3V	10.0ns	0.3V

The clock noise limit is the maximum voltage allowed (for the indicated pulse width) that does not cause an unintentional memory cycle or other memory failure.

Supply Noise Limit (units = V)

Pin	Fast Process 1.1V, -40°C	Fast Process 1.1V, 125°C	Typical Process 1.0V, 25°C	Slow Process 0.9V, 125°C
Power	0.11	0.11	0.10	0.09
Ground	0.11	0.11	0.10	0.09

The power and ground noise limit is the maximum supply voltage transition that is allowed without causing a memory failure.