Universidade de Brasília
Faculdade de Tecnologia

Integração de módulos de redes neurais em hardware em um sistema many-core: um estudo de caso

Gustavo de Medeiros Roarelli

TRABALHO DE GRADUAÇÃO

ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Brasília

2022

Universidade de Brasília
Faculdade de Tecnologia

Integração de módulos de redes neurais em hardware em um sistema many-core: um estudo de caso

Gustavo de Medeiros Roarelli

Trabalho de Graduação submetido como requisito parcial para obtenção do grau de Engenheiro de Controle e Automação

Orientador: Prof. Dr. Marcelo Grandi Mandelli

Brasília
2022

Medeiros Roarelli, Gustavo de.
M488i Integração de módulos de redes neurais em hardware em um sistema many-core: um estudo de caso / Gustavo de Medeiros Roarelli; orientador Marcelo Grandi Mandelli. -- Brasília, 2022.
84 p.

Trabalho de Graduação (Engenharia de Controle e Automação) -- Universidade de Brasília, 2022.

1. Sistemas Manycore. 2. Redes Neurais. 3. System C. 4. Memphis. I. Grandi Mandelli, Marcelo, orient. II. Título

Universidade de Brasília
Faculdade de Tecnologia

Integração de módulos de redes neurais em hardware em um sistema many-core: um estudo de caso

Gustavo de Medeiros Roarelli

Trabalho de Graduação submetido como requisito parcial para obtenção do grau de Engenheiro de Controle e Automação

Trabalho aprovado. Brasília, 24 de junho de 2022:

Prof. Dr. Marcelo Grandi Mandelli,
UnB/IE/CIC
Orientador

Prof. Dr. Marcus Vinicius Lamar,
UnB/IE/CIC
Examinador interno

Prof. Dr. Ricardo Pezzuol Jacobi,
UnB/IE/CIC
Examinador interno

Brasília
2022

Dedico este trabalho a todas as pessoas que o tornaram possível, em especial meus pais Arnaldo e Maria Liz, meus irmãos Otávio e Rafael, e minha namorada Amanda.

Gustavo de Medeiros Roarelli

Agradecimentos

Foi uma jornada. Inicialmente, agradeço meus pais, Arnaldo e Maria Liz, por terem provido tudo o que precisei para toda a minha tragetória até aqui. Livros, escola, computador, carro, me deixar e me buscar... se eu colocasse tudo aqui, ficaria maior que o próprio TG. Obrigado pelo carinho, pela paciência, pelos conselhos e pelo apoio incondicional. Além disso, agradeço aos meus irmãos Otávio e Rafael pelo companheirismo e por sempre estarem ao meu lado. Agradeço também à minha namorada Amanda pelo apoio, por cuidar de mim e por estar sempre comigo nas situações de estresse. Te amo muito.

É imprescindível deixar meus agradecimentos ao Professor Marcelo Grandi Mandelli. Obrigado por ter aceitado ser meu orientador. Muito obrigado pela paciência, ajuda nas mais diversas horas, compreensão e ensino. Nunca irei me esquecer disso.

Deixo também meus agradecimentos aos meus avós Luiz e Maria Luíza. Obrigado pelo amor, pela comida e pelos jogos de futebol de domingo à tarde.

Agradeço também aos meus avós Arnaldo (*in memorian*) e Teresa. Obrigado sempre pela hospitalidade e pelo carinho.

Agradeço ao meu querido primo Ivan, que é quase um irmão, que me acompanhou desde criança e sempre foi um ótimo amigo.

Agradeço aos meus parentes em Tatuí. Obrigado pelas alegrias, festas, comida boa e risadas. Da mesma forma, agradeço a todos os meus queridos parentes em Brasília.

Agradeço ao meu querido amigo Augusto (*in memorian*). Obrigado por ter crescido comigo e ter sido meu amigo. Sempre guardo lembranças preciosas nossas.

Agradeço a todos os professores que me marcaram positivamente, em especial o professor Daniel Rodrigues, que me fez gostar de exatas.

Agradeço ao meu chefe Cristhiano pela ajuda e compreensão no período final do curso, e a todos os colegas da Atlântico, incluindo Glênio, Hugo, Priscila, Nael, Rafa e Tiozão.

Agradeço aos meus amigos de longa data Morais e Ian e a todos os meus amigos do fut. Eu ainda vou, continuem me chamando.

Agradeço aos meus webamigos Bill, Alysson e Fark. Valeu pela companhia nos joguinhos.

Agradeço aos meus queridos amigos de curso e de escola Arthur, Gabriel e Zeca. Sem vocês eu tenho certeza que não teria conseguido.

Por fim, agradeço aos meus grandes amigos Lúcio, Hashirama, Dick, Duda, Henrique Dias, Leo, Lucas e Anny. Obrigado pela amizade preciosa que vocês me oferecem.

Gustavo de Medeiros Roarelli

“And carry on, it’s time to forget
The remains from the past, to carry on”
(André Matos)

Resumo

Este trabalho propõe um estudo de caso de implementação de um periférico de rede neural. Para isso, foi utilizada uma rede neural *feedforward* com duas entradas, uma camada escondida e uma saída, capaz de obter a saída de uma porta XOR. A metodologia utilizada para atingir o sistema proposto foi a da implementação de dois módulos no periférico, sendo estes chamados de Módulo de Rede Neural, que provê a saída da rede neural, e o Módulo de Interfaceamento, que envia e recebe pacotes provenientes da rede gerada pela plataforma Memphis. Além disso, a mesma rede neural foi implementada em *software*, na linguagem C, com objetivo de realizar comparações de desempenho. Para acoplar o periférico à Memphis, foram necessárias algumas adições ao *software* e ao *hardware* da mesma. O periférico de rede neural se mostrou funcional tanto isolado quanto acoplado à plataforma. Foi observado que, quando isolado, o periférico é capaz de enviar e receber pacotes e retornar a saída da rede neural. Quando acoplado à plataforma Memphis, o periférico tem seu tempo de execução individual de apenas 30 ciclos de clock para uma rede descongestionada. Ao comparar o desempenho do sistema gerado pela plataforma com o periférico retornando a saída da rede neural com o software em C implementado, notou-se que o tempo total de execução do sistema em software é, em média, 4% menor. Além disso, o *delay* de requisição da saída da rede neural no sistema em *software* é de 343 ciclos de clock, ao passo que esse tempo em *hardware* é de 1246 ciclos de clock em uma rede descongestionada com apenas 2 tarefas alocadas. Foi observado que essa diferença é decorrente principamente do tempo da chamada de sistema na API; do tempo de envio, recebimento e tratamento de pacotes; e do tempo de escalonamento entre tarefas. Dessa forma, concluiu-se que o desempenho entre o uso de uma implementação em *software* e *hardware* será relativo a complexidade da rede neural implementada.

Palavras-chave: Sistemas Manycore. Redes Neurais. System C. Memphis.

Abstract

This work proposes a case study of an implementation of a neural network peripheral. In order to do this, a feed-forward neural network with two inputs, one hidden layer and one exit, capable of obtaining the output of a XOR logic gate was used. The methodology used to achieve the proposed system was the implementation of two modules in the peripheral, called Neural Network Module, that provides the output value of the neural network, and the Interface Module, that sends and receives packets from the mesh generated by the Memphis platform. Moreover, the same neural network was implemented in software, in C language, for performance comparison purposes. To attach the peripheral to Memphis, some additions were necessary to the Memphis's software and hardware. The neural network peripheral proved to be functional not only isolated but also attached to the platform. It was observed that, when isolated, the peripheral is capable of sending and receiving packets and returning the output value of the neural network. When attached to Memphis, the peripheral has its individual execution time of only 30 clock cycles for a decongested mesh. When comparing the performance of the system generated by the platform when the peripheral returns the output value with the implemented software, it was observed that the total execution time of the system is, in average, 4% smaller. Furthermore, the requisition delay of the output of the neural network in software is 343 clock cycles, whereas in hardware is 1246 in a decongested mesh with 2 allocated tasks. It was observed that this difference is mostly due to the API system call time; sending, receiving and treating packets and the scheduling time between the tasks. Therefore, it is concluded that the performance between the implementation in software and in hardware is relative to the complexity of the neural network implemented.

Keywords: Manycore Systems. Neural Networks. System C. Memphis.

Lista de ilustrações

Figura 1 – Modelo de um Roteador numa NoC (Fonte: (CHEN; YANG; CHANG, 2009)) 21
Figura 2 – Comparação das dimensões PHIT e FLIT (Fonte: (ZEFERINO, 2003)) . 22
Figura 3 – Topologias de NoC Típicas (Fonte: (B. ABDALLAH, 2017)) 23
Figura 4 – Exemplo de uma NoC em malha 3 por 3, sendo R: roteadores, PEs: Elementos de Processamento e NI: Interface de Rede (Fonte: (B. ABDALLAH, 2017)) 23
Figura 5 – Arquitetura da Memphis (Fonte: (RUARO et al., 2019)) 27
Figura 6 – a): NoC Hermes 3 por 3, b): Arquitetura Interna do Roteador (Fonte: (CARARA, 2011)) 28
Figura 7 – Arquiteturas Tradicionais de NI com DMA e DMNI (Fonte: (RUARO, 2018)) 29
Figura 8 – Arquitetura DMNI (Fonte: (RUARO, 2018)) 29
Figura 9 – Código da Função send_packet() (Fonte: (RUARO, 2018)) 30
Figura 10 – MEF do módulo de envio (Fonte: (RUARO, 2018)) 30
Figura 11 – MEF do módulo de recebimento (Fonte: (RUARO, 2018)) 31
Figura 12 – Código da Função read_packer() (Fonte: (RUARO, 2018)) 32
Figura 13 – MEF do árbitro de Acesso à Memória (Fonte: (RUARO, 2018)) 32
Figura 14 – Estrutura de uma Mensagem na Memphis (Fonte: (RUARO et al., 2019)) 33
Figura 15 – Modelo de Funcionamento 34
Figura 16 – Fluxo de Troca de Mensagens Entre PEs (Fonte: (RUARO, 2018)) 34
Figura 17 – Modelo de Neurônio Artificial de Uma Entrada (Fonte: (DEMUTH et al., 2014)) 37
Figura 18 – Modelo de Neurônio Artificial de Múltiplas Entradas (Fonte: (DEMUTH et al., 2014)) 38
Figura 19 – Estrutura Geral de uma RNA (Fonte: (ALEXANDER, 2020)) 38
Figura 20 – Representação Gráfica do Sistema Proposto 39
Figura 21 – Rede Neural *Feedforward* Utilizada 41
Figura 22 – Funções de Ativação Aproximadas (Fonte: (MAKIUCHI, 2018)) 43
Figura 23 – Modelo do Pacote de Recebimento do Periférico 45
Figura 24 – Diagrama da MEF de Recebimento 45
Figura 25 – Modelo do Pacote de Envio do Periférico 46
Figura 26 – Diagrama da MEF de Envio 47
Figura 27 – Modelo de Mensagem de Entrada 49
Figura 28 – Modelo de Mensagem de Saída 49
Figura 29 – Diagrama da Comunicações entre o Periférico e uma Tarefa 55
Figura 30 – Trecho do Código do Arquivo de *Testcase* 57

Figura 31 – Formas de Onda dos Sinais da Rede Neural com Função de Ativação Tipo Linear em Verilog 61
Figura 32 – Formas de Onda dos Sinais da Rede Neural com Função de Ativação Tipo Linear em SystemC 61
Figura 33 – Formas de Onda dos Sinais da Rede Neural com Função de Ativação Tipo LogSig Unipolar em Verilog 62
Figura 34 – Formas de Onda dos Sinais da Rede Neural com Função de Ativação Tipo LogSig Unipolar em SystemC 62
Figura 35 – Arquitetura do Testbench do Sistema Periférico-MTER 63
Figura 36 – Valores Impressos no Terminal Após o Tratamento no MTER e no Periférico 65
Figura 37 – Funcionamento da Aplicação prod_cons 66
Figura 38 – Funcionamento da Aplicação MPEG 67
Figura 39 – Número de Ciclos Impresso no Terminal 67
Figura 40 – Trafego de Flits no Serviço NN_REQUEST 72
Figura 41 – Trafego de Flits no Serviço NN_SEND 72
Figura 42 – Rede Gerada pela Memphis Utilizada 83
Figura 43 – QRCode para acesso ao Repositório do Trabalho no GitHub 84

Lista de tabelas

Tabela 1 – Funções dos Módulos da Rede Neural Traduzida em SystemC 42
Tabela 2 – Descrição de Sinais do Periférico 44
Tabela 3 – Tempos de execução do módulo do periférico para diferentes posições de alocação da tarefa prod 68
Tabela 4 – Resultados do cenário de avaliação do tempo total de execução do sistema 69
Tabela 5 – Tempos de *delay* de *hardware* com várias tarefas no PE [0,1] 70
Tabela 6 – Tempos de *delay* de *hardware* com tarefas provenientes de aplicações diferentes acopladas num mesmo PE 71

Lista de Códigos

3.1 Trecho do Código do Módulo network_g 42
3.2 Portmap da Rede Neural no Periférico 47
3.3 Chamada de Sistema NNPeriph 48
3.4 Função init_Pcommunication 50
3.5 Função get_PERIPHERAL_free_position 50
3.6 Função peripheral_search_and_return 51
3.7 Código da Chamada de Sistema NNPeriph 52
3.8 Código da Função send_nn_request NN_SEND 53
3.9 Código do Serviço NN_SEND 54
3.10 Sinais que Ligam o Periférico ao GPPC 55
3.11 Sinais que Ligam o Periférico ao GPPC 55
3.12 Sinais de Interfaceamento com o Periférico 56
3.13 Implementação da Lista de Sensitividade do Periférico com o Roteador . . . 56
3.14 Projeto de Fiação do Periférico na Memphis 56
3.15 Função nn_peripheral 57
4.1 Método de Definição de Entradas do Testbench da Rede Neural 60
4.2 Método de Recebimento de Pacotes do MTER 62
4.3 Módulo do Testbench do Periférico 63
A.1 Código em C da Aplicação prod 80
A.2 Código em C da Aplicação cons 80

Lista de abreviaturas e siglas

API	Application Programming Interface	20
DMNI	Direct Memory Network Interface	20
GPPC	General Purpose Processing Core	20
MEF	Máquina de Estados Finita	20
Memphis	Many-corE Modeling Platform for Heterogenous SoCs	20
MMR	Memory-mapped Registers	20
MPE	Manager Processing Element	20
MTER	Módulo de Teste de Envio e Recebimento	60
NI	Network Interface	20
NoC	Network on Chip	20
PE	Processing Element	20
RNA	Rede Neural Artificial	20
SF	Store and Forward	20
SPE	Slave Processing Element	20
VCT	Virtual Cut Through	20
WH	Wormhole	20

Sumário

1 INTRODUÇÃO . . . 17
1.1 Objetivos . . . 18
1.1.1 Objetivos Específicos . . . 18
1.2 Estrutura do Trabalho . . . 18

2 FUNDAMENTAÇÃO TEÓRICA . . . 20
2.1 Sistemas *manycore* . . . 20
2.1.1 Rede Intrachip . . . 20
2.1.1.1 Topologias . . . 22
2.1.1.2 Chaveamento . . . 23
Chaveamento por Circuito . . . 24
Chaveamento por Pacote . . . 24
2.1.1.3 Controle de Fluxo . . . 25
2.1.1.4 Roteamento . . . 25
2.1.1.5 Técnicas de Arbitragem . . . 26
2.2 MEMPHIS . . . 26
2.2.1 Região de Processadores de Propósito Geral (GPPC) . . . 27
2.2.1.1 Roteador . . . 27
Sinais do Roteador . . . 28
2.2.1.2 DMNI . . . 29
Módulo de Envio . . . 30
Módulo de Recebimento . . . 31
Árbitro de Acesso à Memória . . . 32
2.2.2 Comunicação entre Tarefas . . . 32
2.2.3 Periféricos . . . 33
2.2.4 Modelo de Gerenciamento . . . 33
2.2.5 Comunicação Entre Módulos de Hardware . . . 33
2.2.6 Protocolo de Injeção de Aplicação Dinâmico . . . 34
2.3 System C . . . 35
2.4 Redes Neurais Artificiais . . . 36
2.4.1 Modelo de um Neurônio Artificial . . . 36
2.4.1.1 Neurônio Artificial de Uma Entrada . . . 37
2.4.1.2 Neurônio Artificial de Múltiplas Entradas . . . 37
2.4.2 Estrutura Geral de uma RNA . . . 38

3 METODOLOGIA . . . 39

3.1 Implementação do Periférico . . . 40
3.1.1 Implementação do Módulo de Rede Neural . . . 40
3.1.1.1 Processo de Tradução da Rede Neural . . . 41
3.1.2 Inplementação do Módulo de Interfaceamento . . . 43
3.1.2.1 Recebimento de Pacotes . . . 44
3.1.2.2 Envio de Pacotes . . . 46
3.1.2.3 Utilização de Diferentes Redes Neurais . . . 47
3.2 Alterações no Software da Memphis . . . 48
3.2.1 Adição de Novos Serviços de Pacotes . . . 48
3.2.2 Adição de Campos de Pacote . . . 48
3.2.3 Adições no Arquivo de API . . . 48
3.2.4 Módulo Peripheral Communications . . . 49
3.2.5 Alterações no Arquivo de Kernel Servo . . . 51
3.2.5.1 Função de Chamadas de Sistema . . . 51
3.2.5.2 Adição da Primitiva send_nn_request . . . 53
3.2.5.3 Função de Tratamento de Pacotes . . . 53
3.3 Alterações no Hardware da Memphis . . . 55
3.3.1 Adições no *Testbench* . . . 55
3.3.2 Adições no Arquivo Principal da Memphis . . . 56
3.4 Rede Neural em Formato de Software . . . 57

4 RESULTADOS . . . 60
4.1 Cenários de Validação do Periférico . . . 60
4.1.1 Validação do Módulo de Rede Neural . . . 60
4.1.2 Validação do Módulo de Interfaceamento . . . 62
4.1.2.1 Módulo de Teste de Envio e Recebimento . . . 62
4.1.2.2 Testbench do Sistema Periférico-MTER . . . 63
4.2 Avaliação do Periférico Acoplado à Memphis . . . 65
4.2.1 Aplicações e Cenários Utilizados . . . 66
4.2.2 Avaliação do Tempo de Execução do Módulo do Periférico . . . 67
4.2.3 Avaliação do Tempo Total de Execução do Sistema . . . 68
4.2.4 Avaliação do Tempo de *delay* de Requisição ao Periférico . . . 69
4.2.4.1 Avaliação do *Delay* com Aumento do Número de Tarefas Alocadas num mesmo PE 70
4.2.4.2 Comparação da Obtenção da Resposta da RNA em *Hardware* com *Software* . . . 71
4.2.5 Validação do Tráfego de Pacotes . . . 72

5 CONCLUSÕES . . . 73
5.1 Conclusão Final . . . 74
5.2 Trabalhos Futuros . . . 75

REFERÊNCIAS 76

APÊNDICES 79

APÊNDICE A – CÓDIGOS DAS APLICAÇÕES PROD E CONS . . . 80

ANEXOS 82

ANEXO A – CONFIGURAÇÃO PADRÃO DOS CENÁRIOS AVALIADOS 83

ANEXO B – QRCODE PARA ACESSO AO REPOSITÓRIO DO TRABALHO NO GITHUB 84

1 Introdução

Com o arrefecimento da Lei de Moore, estudos a respeito de transistores foram cada vez mais necessários. Consequentemente, avanços na tecnologia de transistores propiciaram o desenvolvimento de MCSoCs (do inglês, Manycore System-on-Chips), os quais integram dezenas a centenas de núcleo e elementos de processamento em um único chip (TOFIS; THEOCHARIDES; MICHAEL, 2011).

Tipicamente, um MCSoC tem a sua arquitetura composta por núcleos conectados por uma rede intrachip (VENKATARAMANI; CHAN; MITRA, 2019). Dependendo do uso desses núcleos, pode-se classificar MCSoCs entre os tipos homogêneos e heterogêneos. MCSoCs homogêneos têm seus núcleos idênticos e são flexíveis a uma variedade de aplicações. Por outro lado, MCSoCs heterogêneos utilizam diferentes tipos de núcleos, com foco na execução de um conjunto específico de aplicações.

Nos últimos anos, um aumento crescente na densidade energética e na demanda de desempenho levou a um maior uso de MCSoCs heterogêneos (WEICHSLGARTNER et al., s.d.) (PAGANI et al., s.d.). Isso se deve ao fato de MCSoCs heterogêneos suprirem melhor essas demandas em comparação com os MCSoCs homogêneos (LI; LOURI, 2020) (ARDA et al., 2019). Apesar de esse suprimento depender do conjunto de núcleos do sistema e do desempenho dos mesmos na execução do conjunto de aplicações, esses fatos são uma motivação para o uso e estudo de sistemas heterogêneos.

Entre os núcleos que podem integrar um MCSoC heterogêneo estão módulos de *hardware* que implementam redes neurais artificiais. Redes neurais artificiais (RNAs) demonstraram, nos últimos anos, um desempenho excelente em aplicações de diversas áreas, como processamento de sinais, padrões de redes sem fio e classificação de dados (ALEXANDER, 2020).

Dessa forma, a integração de uma RNA em uma plataforma MCSoCs é um ótimo estudo de caso para realizar comparações entre implementações em *software* versus *hardware* e estudar vantagens e desvantagens de tal implementação.

Neste trabalho, a plataforma MCSoCs Memphis foi usada como referência. Essa plataforma permite validar e realizar simulações de processadores *manycore* a nível lógico (RUARO et al., 2019)), que é ideal para a integração de uma RNA. Além disso, a Memphis tem a vantagem de disponibilizar ao projetista a possibilidade de trabalhar com a linguagem de descrição de *hardware* SystemC.

1.1 Objetivos

Este trabalho de graduação tem como objetivo geral a integração de núcleos de redes neurais descritos em *hardware* na plataforma MCSoC heterôgenea Memphis para estudos de caso. A rede neural utilizada neste trabalho é proveniente de trabalhos de cooperação entre outros docentes e discentes do Departamento de Ciência da Computação da Universidade de Brasília (CAETANO et al., 2019).

1.1.1 Objetivos Específicos

1. Aprendizagem de SystemC: Sabendo que a integração se dá através de SystemC, foi necessário um estudo da linguagem para realizar traduções, interfaceamentos e conexões entre módulos.

2. Tradução da Rede Neural para SystemC: Após a primeira etapa, é necessária a tradução da rede neural em Verilog fornecida.

3. Desenvolvimento de uma Interface para a Rede Neural: É preciso interfacear a rede neural para permitir a comunicação com elementos de processamento da Memphis.

4. Desenvolvimento de recursos que propiciem a comunicação entre tarefas e o módulo da rede neural: Para realizar a comunicação, é necessário realizar o controle de solicitações e adicionar funções e serviços que auxiliem este objetivo.

5. Avaliação de Resultados: Avaliar os resultados de forma que o que foi desenvolvido se aproxime do proposto.

6. Comparação *Software* x *Hardware*: Após a validação dos resultados, os módulos de *hardware* serão comparados com implementações em *software*, utilizando a linguagem C, que executam as mesmas funções de tais módulos. Esta comparação permitirá a avaliação de ganhos de desempenho entre as implementações de *software* e *hardware*.

1.2 Estrutura do Trabalho

O conteúdo deste trabalho é dividido em 5 capítulos, listados a seguir:

1. Introdução: Este capítulo apresenta a motivação por trás do tema e os objetivos buscados neste trabalho.

2. Fundamentação Teórica: Este capítulo tem como objetivo abordar conceitos importantes para o entendimento da metodologia por parte do leitor.

3. Metodologia: Este capítulo apresenta o desenvolvimento do projeto buscando atingir os objetivos apresentados no capítulo 1.

4. Resultados: Este capítulo mostra os resultados finais validados segundo o estudo de caso.

5. Conclusão: Este capítulo apresenta conclusões gerais a respeito dos resultados e enuncia trabalhos futuros.

2 Fundamentação Teórica

Neste capítulo, serão introduzidos conceitos e fundamentos teóricos para uma imersão do leitor no tema. A Seção 2.1 introduz conceitos essenciais de sistemas *manycore*. A Seção 2.2 mostra o funcionamento da plataforma Memphis. A Seção 2.3 enuncia alguns conceitos da linguagem de descrição de *hardware* SystemC. A Seção 2.4 apresenta o funcionamento e conceitos de Redes Neurais Artificiais.

2.1 Sistemas *manycore*

Um sistema *manycore* pode ser definido como um sistema que integra múltiplos núcleos de processamento (PEs, do inglês *processing element*) em uma único chip (TOFIS; THEOCHARIDES; MICHAEL, 2011). Uma outra definição para manycores é MPSoC (do inglês *Multiprocessor System-on-chip*). A mudança do termo "multi" para "many" surge devido a um crescimento do número de PEs num chip. Neste trabalho; será utilizado o termo *manycore* por se tratar de um sistema que abrange de dezenas a centenas de núcleos.

A arquitetura de um *manycore* varia de acordo com o tipo e objetivo da aplicação trabalhada. Logo, o uso e tipo de PEs também é variado. Com isso em mente, pode-se dividir sistemas *manycore* em dois tipos: homogêneo e heterogêneo. Um sistema heterogêneo tem como característica a utilização de diferentes arquiteturas de PEs (RUARO et al., 2019). Um sistema *manycore* homogêneo pode ser classificado por homogêneo simétrico e homogêneo assimétrico. Num sistema homogêneo simétrico, todos os PEs possuem a mesma arquitetura e organização e são multiplicados no sistema. Num sistema homogêneo assimétrico, todos os PEs possuem a mesma arquitetura, mas possuem organizações diferentes (RUARO et al., 2019).

Alguns autores afirmam que sistemas heterogêneos têm uma menor flexibilidade quando comparados aos homogêneos (WÄCHTER; BIAZI; MORAES, 2011). Entretanto, sistemas heterogêneos oferecem um melhor desempenho e uma melhor eficiência energética quando comparados aos homogêneos. Isso ocorre devido a um alto aproveitamento da especialização *on-chip* oferecida por esses sistemas (LI; LOURI, 2020). Devido a essa especialização, pode-se concluir que sistemas heterogêneos estão sendo cada vez mais usados nas mais diversas aplicabilidades.

2.1.1 Rede Intrachip

Com o objetivo de otimizar sistemas *manycore*, a infraestrutura de rede intrachip (Noc, do inglês *Network on Chip*) é bastante implementada. Em termos de infraestrutura

de comunicação, a NoC é geralmente vista como uma das mais proeminentes soluções de interconexões em circuitos integrados de grande escala, incluindo processadores *many-core* (ZHANG et al., 2009). Essa infraestrutura funciona a partir de comunicação entre PEs baseada em envio e recebimento de pacotes. Os pacotes geralmente possuem a seguinte estrutura:

- Cabeçalho: dispõe de informações necessárias para o roteamento e propagação da mensagem (SEPÚLVEDA FLÓREZ, 2011).
- Carga útil: carrega a informação a ser transferida (SEPÚLVEDA FLÓREZ, 2011).
- Terminador: carrega informações que são utilizadas para indicar o fim da mensagem e detecção de erros (SEPÚLVEDA FLÓREZ, 2011).

A NoC possui três blocos fundamentais em sua estrutura, sendo estes os enlaces, a interface de rede (NI, do inglês *Network Interfaces*) e os roteadores. Enlaces são os meios físicos através dos quais os pacotes transitam. Já a NI converte solicitações/respostas de transação em pacotes (ATIENZA et al., 2008). Por fim, os roteadores têm a função de roteamento. O roteador embutido em cada elemento de uma NoC tipicamente consiste em conexões de entrada e de saída, s e um comutador crossbar (CHEN; YANG; CHANG, 2009). A Figura 1 mostra um modelo de um roteador.

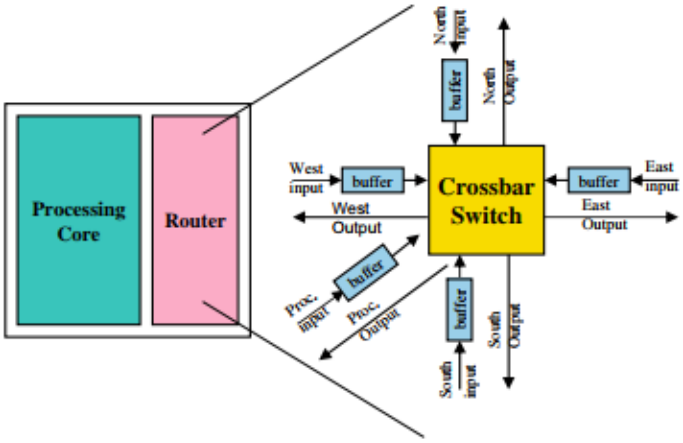

Figura 1 – Modelo de um Roteador numa NoC (Fonte: (CHEN; YANG; CHANG, 2009))

Tipicamente, NoCs são formadas por dois canais unidirecionais opostos. Um canal é responsável pelo envio e o outro pelo recebimento de pacotes. A unidade física que corresponde à largura desses canais é o PHiT (*PHisical unIT*. Pode-se dizer que PHIT indica o número de bits transmitidos paralelamente por um canal (MARINHO, 2018). Conclui-se que um conjunto de PHITs determina o controle de fluxo FLIT. Um pacote a ser transmitido é

composto por um conjunto de FLITs. A Figura 2 nos mostra uma comparação das dimensões PHIT e FLIT.

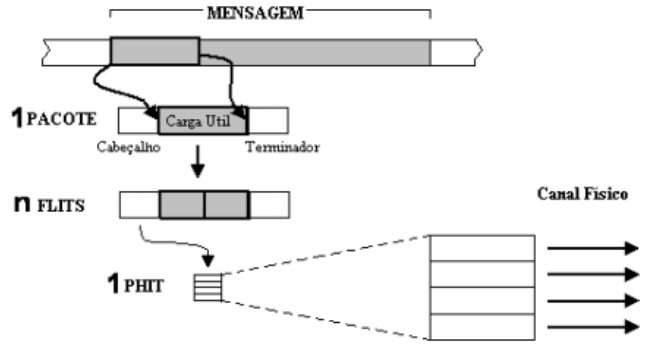

Figura 2 – Comparação das dimensões PHIT e FLIT (Fonte: (ZEFERINO, 2003))

A arquitetura de uma NoC é, geralmente, caracterizada por: sua topologia, roteamento, chaveamento, controle de fluxo e técnicas de arbitragem. Nas próximas sessões trataremos de cada propriedade, individualmente.

2.1.1.1 Topologias

É natural na otimização de uma rede o desenvolvimento ou implementação de um arranjo entre os seus respectivos elementos, ou seja, sua topologia. No caso de uma NoC, a topologia define como os seus roteadores e enlaces estão interconectados. A topologia é de caráter importante no projeto de um sistema porque define a distância de comunicação e a uniformidade entre os elementos supracitados.

A escolha da topologia depende de que forma influenciam suas vantagens e desvantagens. Pode-se assim dividir topologias entre regulares (padronizadas), mostradas em Figura 3 de a) a c) e irregulares (não padronizadas) mostrada em Figura 3 de d) a f). Normalmente, topologias regulares são preferíveis por conta de sua escalabilidade e o fato de o seu padrão poder ser usado repetidamente. Por outro lado, topologias irregulares são mais recomendadas quando a aplicação tem uma necessidade mais especifica. Entretanto, o tempo de desenvolvimento do projeto cresce com a irregularidade da topologia.

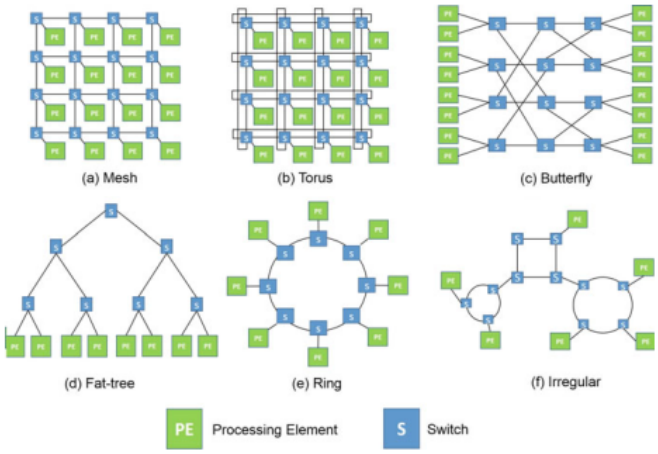

Figura 3 – Topologias de NoC Típicas (Fonte: (B. ABDALLAH, 2017))

A topologia mais utilizada em NoCs é a topologia em malha (*mesh*) por conta de um *trade-off* benéfico entre custo de implementação e desempenho (HOJABR et al., 2017). Numa malha, os nós são organizados em uma estrutura similar a uma grade (Figura 4) em que cada nó é conectado ao seu vizinho mais próximo referente à sua latitude ou longitude. Devido a essa estrutura de caráter regular e com um *layout* amigável, NoCs em malha beneficiam de uma facilidade de fiação e implementação a nível de circuito (B. ABDALLAH, 2017).

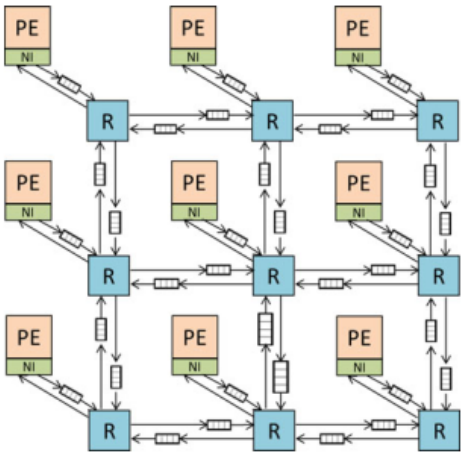

Figura 4 – Exemplo de uma NoC em malha 3 por 3, sendo R: roteadores, PEs: Elementos de Processamento e NI: Interface de Rede (Fonte: (B. ABDALLAH, 2017))

2.1.1.2 Chaveamento

A forma através da qual dados são transferidos de um canal de entrada para um de saída é determinada pelo chaveamento. O chaveamento é geralmente dividido em por circuito e por pacote.

Chaveamento por Circuito

O chaveamento por circuito tem por principal característica a determinação e a reservação prévias do caminho entre a origem e o destino. Isso é feito através de duas etapas. Na primeira etapa, o nó de origem injeta na rede um cabeçalho que contém o endereço do nó de destino e algumas informações de controle. Esse cabeçalho vai percorrendo a rede enquanto reserva os canais físicos através dos quais a mensagem trafegará. Se o canal estiver reservado para outra mensagem, o cabeçalho é bloqueado e entra em modo de espera até que o canal esteja disponível. Ao chegar no nó de destino, o cabeçalho envia a informação de reconhecimento obtida ao nó de origem, autorizando então o envio da mensagem. A segunda etapa consiste na transferência dos dados da mensagem.

Chaveamento por Pacote

O chaveamento por pacote é mais comum e mais utilizado em NoCs (B. ABDALLAH, 2017). O chaveamento por pacote é aqui subdividido em armazena e repassa (SF, do inglês *store and forward*), com transpasse virtual (VCT, do inglês *virtual cut through*) e *wormhole* (WH).

Chaveamento por Pacote Armazena e Repassa (*store and forward*) Nesse tipo de chaveamento por pacote, as mensagens são divididas em vários pacotes. Cada pacote citado é armazenado em um FIFO antes de ser enviado ao próximo roteador. É importante notar que o tamanho do é o mesmo tamanho do pacote, já que o envio é feito de forma a uma mensagem ser enviada de cada vez.

Chaveamento por Pacote *wormhole* O chaveamento por pacote *wormhole* é o método mais utilizado de chaveamento em NoCs (B. ABDALLAH, 2017). Nesse método, os pacotes são divididos em um número de FLITs. Cada FLIT ocupa um roteador, ou seja, n FLITs ocupam n roteadores. Isso dispensa a necessidade de s de grandes tamanhos, pois o tamanho necessário é o tamanho de um FLIT, ou seja, o menor tamanho possível. Entretanto, existe uma forte dependência entre o último FLIT de um pacote e o primeiro de um segundo pacote. Isso faz com que, caso um FLIT do primeiro pacote qualquer seja bloqueado, os FLITs do segundo pacote também ficam bloqueados. Consequentemente, o sistema entra em *deadlock*. Isso pode ser solucionado pelo uso de Canais Virtuais (*Virtual-Channels*) (B. ABDALLAH, 2017).

Chaveamento por Pacote com Transpasse Virtual (*virtual cut through*) Seria um intermediário entre WH e SF. Esse tipo de chaveamento também divide o pacote em FLITs, como no WH. Entretanto, temos um *buffer* com o tamanho de um pacote, conforme se tem no SF. Isso resolve, de certa forma, o problema o problema de bloqueamento do WH,

pois mais FLITs podem ser armazenados em um roteador evitando o bloqueamento e uma consequente entrada em *deadlock* do sistema. Porém, o tempo de atraso no VCT é maior quando comparado ao WH, o que não torna esse tipo de chaveamento uma solução imediata.

2.1.1.3 Controle de Fluxo

O controle de fluxo de uma NoC determina como recursos, como *buffers* e largura de banda dos canais, são alocados e como colisões de pacotes são resolvidas (B. ABDALLAH, 2017) ou mitigadas. Uma boa estratégia de controle de fluxo é essencial no projeto de uma NoC. Existem diversos tipos de controle de fluxo, como por Canal Virtual (que resolve o problema do chaveamento via WH (B. ABDALLAH, 2017)), liga e desliga (ON/OFF) e ACK/-NACK. Dois tipos de implementação de controle de fluxo são importantes no entendimento deste trabalho:

- Implementação via handshaking: uma mensagem é enviada ao receptor questionando se é possível o recebimento do próximo pacote ou FLIT.
- Implementação via créditos: um contador contido no receptor envia ao respectivo emissor informações de disponibilidade de espaço. Essas informações são atualizadas conforme esse receptor envia e recebe pacotes.

2.1.1.4 Roteamento

Um pacote/mensagem, para chegar ao seu destinatário, precisa percorrer um caminho. A escolha do caminho que esse pacote percorre depende do algoritmo de roteamento implementado. De acordo com Abdallah, algoritmos de roteamento podem ser classificados de acordo com as seguintes propriedades:

- Número de destinatários: de acordo com o número de nós de destino, algoritmos de roteamento podem ser separados entre unicast e multicast. Roteamento unicast é caracterizado pelo envio de uma única origem para um único destino. Roteamento multicast é caracterizado pelo envio de uma única origem para múltiplos destinos.
- Localidade de decisões de roteamento: de acordo com a localidade na qual as decisões de roteamento são feitas, algoritmos de roteamento (tanto multicast quanto unicast) podem ser classificados em centralizado, fonte e distribuído (ZEFERINO, 2003). No roteamento centralizado, os caminhos são escolhidos através de um controlador central na rede. No roteamento por fonte, o nó de origem define o caminho que o pacote seguirá antes de injetá-lo na rede. No roteamento distribuído, o caminho é definido pelos roteadores enquanto o pacote percorre a rede.
- Adaptabilidade: além das classificações nas propriedades já citadas, algoritmos de roteamento podem ser divididos por conta da sua adaptabilidade, podendo ser determi-

nísticos ou adaptativos. Algoritmos de roteamento determinísticos são implementados de forma que os caminhos da origem até o destino são estaticamente computados e serão sempre, no mínimo, similares. Já algoritmos de roteamento adaptativos são computados com o objetivo de o caminho ser adaptável conforme diferentes informações são obtidas, como o congestionamento entre canais e o estado da porta de saída.

- Minimalidade: de acordo com a minimalidade dos algoritmos de roteamento, estes podem ser divididos ainda entre mínimos e não-mínimos. Num algoritmo de roteamento mínimo, as mensagens sempre são roteadas pelo roteador mais próximo do destino e percorrem então o menor número de caminhos entre roteadores vizinhos possível. Já no não-mínimo, as mensagens podem ser roteadas por roteadores mais distantes. Isso pode ser determinado por uma regra ou restrição (B. ABDALLAH, 2017) ou simplesmente de forma randômica.

2.1.1.5 Técnicas de Arbitragem

A arbitragem tem papel essencial na resolução de conflitos. A principal função desse recurso é eleger através de algum parâmetro ou regra o próximo pacote ou FLIT a ser recebido e para onde ele vai. De forma análoga, o roteador efetua o controle de saída e o árbitro efetua o controle de entrada. A sua implementação pode se dar de forma centralizada ou distribuída (ZEFERINO, 2003). A arbitragem centralizada tem como cenário a implementação de mecanismos de roteamento e arbitragem em único módulo, o que faz com que atuem de forma dependente entre si. Já a distribuída tem como característica a independência entre o roteamento e a arbitragem. Cada uma das portas possui um módulo de roteamento associado ao seu canal de entrada e um de arbitragem associado ao seu canal de saída. Os mecanismos de arbitragem também são baseados em diferentes critérios, como prioridades estáticas, prioridades rotativas (*round-robin*), escalonamento por idade (*deadline*), FCFS (do inglês *first-come-first-served*), LRS (do inglês *least recently served*), dentre outros (ZEFERINO, 2003).

2.2 MEMPHIS

Esta seção busca descrever e detalhar o funcionamento da Memphis, um modelo *manycore* e framework para geração automática e validação de *manycore*s a nível lógico. Memphis é o acrônimo de "Many-corE Modeling Platform for Heterogenous SoCs".

A Memphis consiste em duas principais contribuições. A primeira é um framework para uma geração de um MPSoC com infraestrutura NoC que permite o projeto de um modelo de plataforma, que aborda uma região *manycore* homogênea cercada por periféricos. A segunda é uma integração acoplada do modelo de plataforma com ferramentas de debugging, permitindo traçar eventos de *hardware* e *software* simultaneamente. Em suma, a plataforma

permite um design lógico modular que permite a geração de *hardware* e a compilação de sistemas e aplicações.

Pode-se ver na Figura 5 a arquitetura da Memphis em detalhes. De imediato, nota-se que é um sistema *manycore* heterogêneo, baseando-se na definição dada na última seção. Contém duas regiões, sendo estas a de processadores de propósito geral (GPPC do inglês *General Purpose Processing Core*) representada em Figura 5 a) e a região de periféricos. Em b), tem-se a arquitetura de um PE.

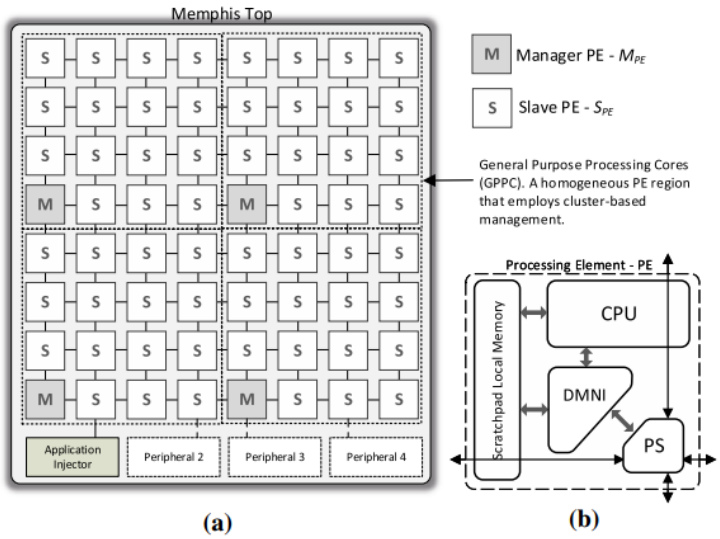

Figura 5 – Arquitetura da Memphis (Fonte: (RUARO et al., 2019))

2.2.1 Região de Processadores de Propósito Geral (GPPC)

O GPPC possui uma série de PEs homogêneos e simétricos. Cada PE (Figura 5 b) da região de GPPC possui um processador de arquitetura MIPS de 32 bits, uma memória local do tipo *scratchpad*, que guarda códigos e instruções, além de um roteador NoC (apresentado na Seção 2.2.1.1) e um módulo de Interface de Rede de Acesso de Memória Direto (DMNI, do inglês *Direct Memory Network Interface* (apresentado na Seção 2.2.1.2).

2.2.1.1 Roteador

O roteador adotado pela Memphis é um roteador NoC Hermes. A NoC Hermes é uma NoC com topologia de malha 2D, com chaveamento por pacotes do tipo *wormhole* e controle de fluxo baseado em crédito. Os roteadores possuem *buffers* de entrada, uma lógica de controle (*Switch Control* - a arbitragem e roteamento) compartilhada por todas as portas, um crossbar interno e até cinco portas bidirecionais (norte, sul, leste, oeste e local). A Figura 6 a) mostra um exemplo de NoC Hermes 3 por 3 e a Figura 6 b) mostra a arquitetura interna do roteador.

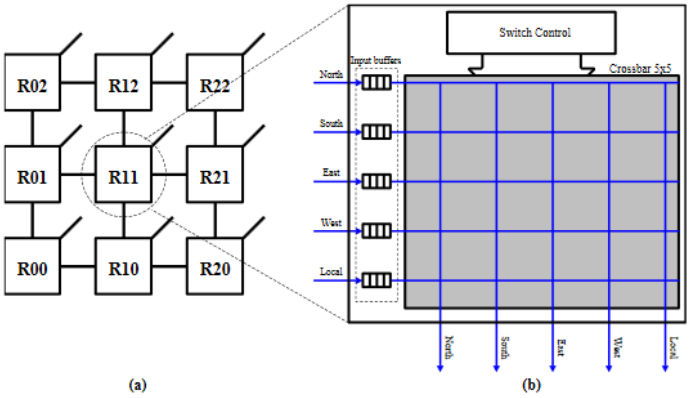

Figura 6 – a): NoC Hermes 3 por 3, b): Arquitetura Interna do Roteador (Fonte: (CARARA, 2011))

Essa NoC utiliza a arbitragem *Round-Robin*, que utiliza um esquema de prioridades dinâmicas. O algoritmo de roteamento é o XY. Esse algoritmo faz com que os pacotes da rede sejam enviados, primeiramente, horizontalmente até chegar a coordenada X do roteador destino. Em seguida, percorre a NoC verticalmente, até encontrar de fato o roteador destino na coordenada vertical Y indicada.

Sinais do Roteador

Os seguintes sinais são utilizados na comunicação entre roteadores na Memphis, dando suporte a um controle de fluxo baseado em créditos: rx (notifica dados a serem recebidos), tx (notifica dados a serem transmitidos), credit_in (notifica que o destino dispõe de créditos, ou seja, está apto a receber dados), credit_out (notifica que a origem dispõe de créditos, ou seja, está apta a receber dados), data_in (armazena o dado recebido) e data_out (armazena o dado a ser enviado). Com isso, o sinal rx da origem é conectado ao sinal tx do destino, o sinal credit_in da origem é conectado ao sinal credit_out do destino, e o sinal data_out da origem é conectado no sinal data_in do destino.

Por utilizar o chaveamento por pacotes do tipo *wormhole*, os dados de um pacote são enviados de um roteador a outro em flits. Assim, para ocorrer o envio de um flit, um roteador primeiramente verifica a existência de créditos, definido quando credit_in estiver em "1". Existindo créditos, o roteador define o sinal tx como "1" para indicar a transmissão e armazena o flit a ser enviado no sinal data_out. Já para o recebimento de um flit por parte de um roteador, primeiramente, é visto se o sinal rx está em "1", indicando o recebimento de dados. Cumprindo essa condição e, se houver créditos disponíveis, o valor recebido é lido do sinal data_in. Paralelamente, o roteador gera o sinal de credit_out, indicando a existência de créditos, baseado na disponibilidade de espaço em *buffer* para armazenamento do flit a ser recebido.

2.2.1.2 DMNI

Pode-se dizer que a DMNI é composta por uma fusão entre os módulos NI e Acesso de Memória Direto (DMA, do inglês *Direct Memory Access*). O principal objetivo da DMNI é prover uma interface especializada para *manycores* com sistema NoC que conecta diretamente o roteador com a memória interna usando um único módulo. A DMNI consegue, simultaneamente, receber e transmitir pacotes.

A Figura 7 mostra uma arquitetura tradicional de NI com DMA comparada à arquitetura DMNI utilizada na Memphis.

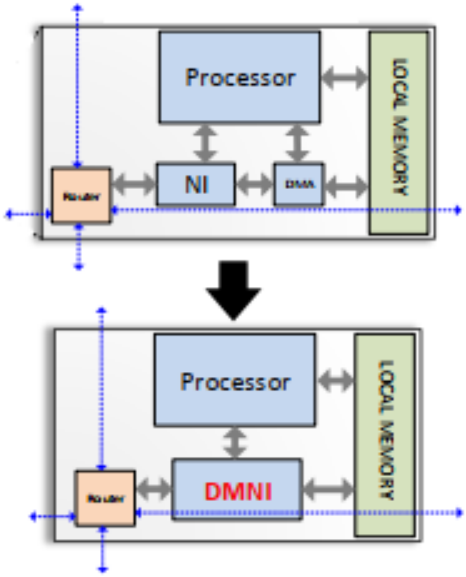

Figura 7 – Arquiteturas Tradicionais de NI com DMA e DMNI (Fonte: (RUARO, 2018))

A Figura 8 detalha a arquitetura DMNI. A DMNI possui três principais módulos: envio (*send)*, recebimento (*receive*) e árbitro (*Memory Acess Arbiter*). O módulo árbitro gerencia os acessos de memória para ambos os módulos (send e receive) e permite assim operações de envio e recebimento serem executadas simultaneamente. O kernel controla a DMNI através de registradores mapeados em memória (MMRs, do inglês *memory-mapped registers*).

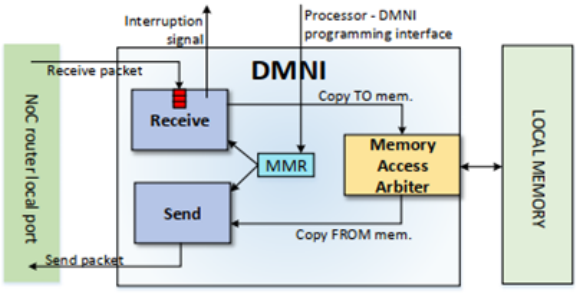

Figura 8 – Arquitetura DMNI (Fonte: (RUARO, 2018))

Módulo de Envio

O objetivo do módulo de envio é injetar um pacote na NoC. A característica principal desse módulo é possibilitar a transferência de dois blocos de memória como uma única transferência.

A Figura 9 enuncia a função send_packet() fornecida na DMNI API. Essa função está presente nos sistemas operacionais dos SPEs e MPEs, fazendo a interface de envio de pacotes através da DMNI fornecida na DMNI API. Essa função prevê o envio de um pacote composto apenas pelo cabeçalho da mensagem ou desse cabeçalho conjuntamente com a carga útil. Para isso, ela recebe como parâmetro, respectivamente, os tamanhos e os endereços iniciais de memória do cabeçalho (mem_size1 e mem_addr1) e da carga útil (mem_size_2 e mem_addr_2)

```
1. void send_packet(mem_size_1, mem_addr_1, mem_size_2, mem_addr_2){
2.     while (MemoryRead(DMNI_SEND_ACTIVE));
3.     MemoryWrite(DMNI_SIZE, mem_size_1);
4.     MemoryWrite(DMNI_ADDRESS, mem_addr_1);
5.     if (mem_size_2 > 0){
6.        MemoryWrite(DMNI_SIZE_2, mem_size_2);
7.        MemoryWrite(DMNI_ADDRESS_2, mem_addr_2);
8.     MemoryWrite(DMNI_OP, READ);
9.     MemoryWrite(DMNI_START, 1);
10.}
```

Figura 9 – Código da Função send_packet() (Fonte: (RUARO, 2018))

A função recebe, respectivamente, os primeiros e os segundos tamanhos e endereços de memória. Se a DMNI está transmitindo um pacote (DMNI_SEND_ACTIVE-1), o procedimento fica estagnado na linha 2 até que o módulo DMNI seja liberado. Nas linhas 3 e 4, o primeiro bloco de memória é configurado. Se uma mensagem tem um payload (opcional, conforme dito anteriormente), o segundo bloco de memória é configurado. Na linha 8, o tipo de operação é escrito. Na linha 9, a DMNI é autorizada a começar a transmissão do pacote.

A Figura 10 apresenta a Máquina de Estados Finita que controla o módulo de envio.

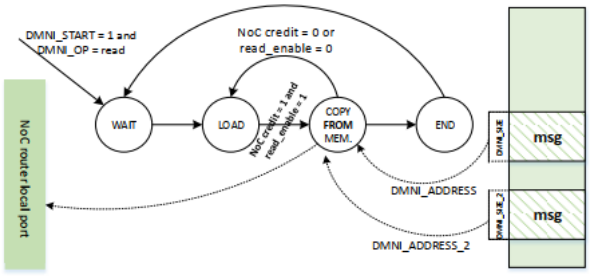

Figura 10 – MEF do módulo de envio (Fonte: (RUARO, 2018))

Inicialmente, a MEF espera a configuração dos MMRs (estado WAIT) pela função send_packet(). Quando as linhas 8 e 9 são executadas, a MEF vai para o estado LOAD, e a MEF envia um sinal send_active para o árbitro para requisitar acesso à memória. O estado LOAD verifica se a porta local do roteador pode receber dados (credit=1) e se o

árbitro permite uma operação de leitura (read_enable=1). Caso ambas as condições estejam satisfeitas (credit, read_enable=1), os dados são lidos da memória e injetados na porta local do rotador (estado COPY_FROM_MEM). Sempre que o árbitro ou a porta local desativem a transmissão, a MEF retorna ao estado LOAD. A MEF envia o primeiro bloco de memória e então muda o ponteiro de endereço para o segundo bloco de memória (se configurado) para transmitir os dados restantes.

Módulo de Recebimento

A Figura 11 detalha o funcionamento do módulo de recebimento. Contém duas MEFs e um *buffer* de 16 flits. O tamanho do é parametrizável na hora da definição do projeto. Quando um pacote chega no local da porta do roteador NoC, o estado HEADER lê o primeiro flit do pacote, interrompendo o processador. Em seguida, o estado PAYLOAD_SIZE lê o tamanho do payload e avança para o estado DATA, que lê os flits restantes do pacote. O recebe todos os flits. A NoC pára quando o fica cheio.

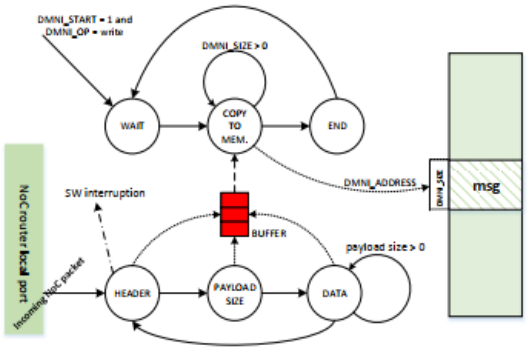

Figura 11 – MEF do módulo de recebimento (Fonte: (RUARO, 2018))

A Figura 12 enuncia o processo do segurador (handler) de interrupção, que é a função read_packet(). A função read_packet() escreve nos MMRs a quantidade de dados a receber (linha2), o endereço de memória a depositar no pacote (linha 3), a operação DMNI (linha 4) e um comando start (linha 5). O kernel espera a recepção completa do pacote (linha 6) para não haver nenhum problema na leitura da memória. Em seguida, o kernel executa as ações relacionadas ao serviço de pacote. As condições write e start, mostradas nas linhas 4 e 5 da função read_packet(), caso atendidas liberam o começo da MEF. A MEF transfere os dados depositados no para a memória local (estado COPY_TO_MRM). Para escrever na memória, a segunda MEF declara o sinal receive_active para o árbitro requisitar o acesso à memória. O árbitro pode liberar acesso à memória ao declarar o sinal write_enable. Se o árbitro não garantir acesso à memória, a MEF fica estagnada no estado COPY_TO_MEM

```
1. void read_packet (init_addr, packet_size)
2.       MemoryWrite(DMNI_SIZE, packet_size);
3.       MemoryWrite(DMNI_ADDRESS, init_addr);
4.       MemoryWrite(DMNI_OP, WRITE);
5.       MemoryWrite(DMNI_START, 1);
6.       while (MemoryRead(DMNI_RECEIVE_ACTIVE));
7. }
```

Figura 12 – Código da Função read_packer() (Fonte: (RUARO, 2018))

Árbitro de Acesso à Memória

O árbitro permite que os acessos da memória concorrente recebam e enviem pacotes. O PE pode então receber novos dados e concorrentemente injetar novos pacotes na NoC. Um árbitro RR (*round-robin*) habilita esse recurso ao controlar os sinais read_enable(send) e write_enable(receive). Um *timer* (DMNI_TIMER) controla a quantidade de tempo que cada módulo pode acessar a memória.

A Figura 13 mostra a MEF que controla o árbitro. O sinal round recebe o módulo para permitir acesso. As MEF dos módulos receive e send declaram os sinais send_active e receive_active, respectivamente. Quando o árbitro vai para o estado SEND, o sinal read_enable é declarado, habilidando o módulo de envio acessar a memória. A MEF fica nesse estado enquanto o sinal send_active é configurado, ou o *timer* expirou e outro módulo requisitou acesso à memória.

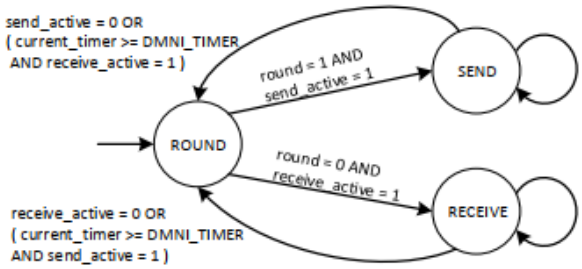

Figura 13 – MEF do árbitro de Acesso à Memória (Fonte: (RUARO, 2018))

2.2.2 Comunicação entre Tarefas

A comunicação entre tarefas é realizada através do envio e recebimento de mensagens. Isso é feito através de uma API de comunicação a nível de kernel na qual estão contidas diversas primitivas. Dentre estas, estão as primitivas Send e Receive. Essas primitivas chamam as rotinas writepipe() e readpipe(), respectivamente, que estão contidas no kernel de um PE servo (kernel *slave*). As tarefas consumidoras chamam a rotina Receive, ao passo que as produtoras chamam a rotina Send.

Cada kernel possui uma zona de memória chamada *pipe*, na qual são armazenadas as mensagens. A rotina writepipe() armazena uma mensagem proveniente da tarefa produtora no *pipe* utilizando o protocolo FIFO. Essa mensagem permanece lá até que seja solicitada pela rotina readpipe() e em seguida enviada para a tarefa consumidora.

2.2.3 Periféricos

Os periféricos provêm interface I/O ou aceleração de *hardware* para tarefas executadas na região da GPPC. Com os recursos de comunicação da Memphis, pode-se conectar periféricos nos PEs mais externos da rede. As portas dos roteadores desses PEs nas quais ocorre a conexão são aquelas que não estão sendo utilizadas, como, por exemplo, a porta norte de um PE que se encontra na posição mais acima possível da rede. Ou seja, são as portas mais externas. Dessa forma, os periféricos são sempre conectados nas portas das fronteiras da malha NoC.

Dois tipos de pacotes são trocados entre os PEs. Pacotes de dados são aqueles que são trocados por tarefas executadas em PEs. Pacotes periféricos são aqueles trocados entre a tarefa e um periférico. A NoC implementada diferencia pacotes de dados de pacotes periféricos.

A Figura 14 mostra uma flag no início da mensagem que diz ao NoC se o pacote é de dados ou periférico. Mais à frente será detalhado como ocorre a comunicação entre módulos de *hardware*.

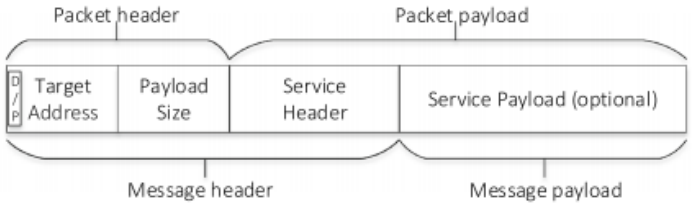

Figura 14 – Estrutura de uma Mensagem na Memphis (Fonte: (RUARO et al., 2019))

2.2.4 Modelo de Gerenciamento

A Memphis adota um gerenciamento descentralizado baseado em clusters. Esses clusters são regiões virtuais no GPPC, com uma série de processadores servos (SPEs) e um gerente (MPE). SPEs executam as tarefas das aplicações, ao passo que MPEs gerenciam os clusters.

A Figura 15 define o modelo de funcionamento. O gerenciamento é executado pelos sistemas operacionais (kernels), que operam nos próprios PEs.

2.2.5 Comunicação Entre Módulos de Hardware

A comunicação é feita através de mensagens unicast. O kernel gerencia a comunicação ao enviar e receber pacotes. Do ponto de vista do kernel, a mensagem tem:

- Cabeçalho da Mensagem (Message header): encapsula o cabeçalho do pacote e o tamanho do payload e adiciona outros campos de controle pelo kernel.

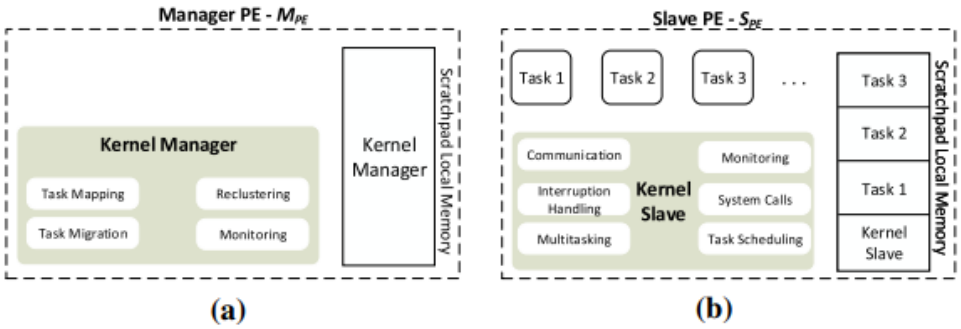

Figura 15 – Modelo de Funcionamento

- Carga útil da Mensagem (Message payload): campo opcional. Armazena dados usados em contexto do cabeçalho de serviço.

A Figura 16 representa o fluxo de troca de mensagem entre dois PEs diferentes.

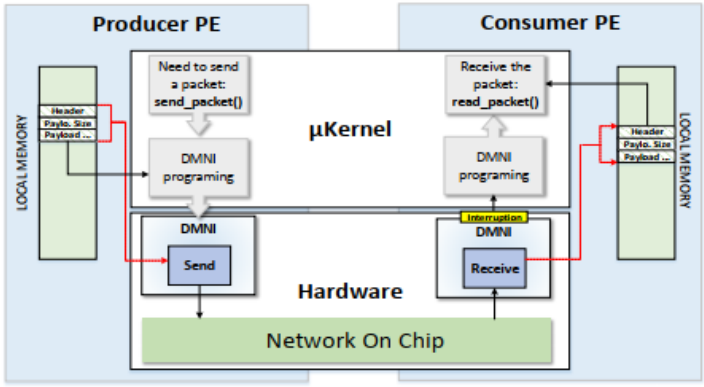

Figura 16 – Fluxo de Troca de Mensagens Entre PEs (Fonte: (RUARO, 2018))

Quando o kernel do PE produtor precisa enviar um pacote, ele chama a função send_packet() que programa a DMNI para começar o envio do pacote, copiando os dados da memória e transmitindo-os à NoC. Do lado do PE consumidor, quando a DMNI recebe um pacote ela interrompe o processador. O gerenciador (*handler*, em inglês) da interrupção chama a função read_packet(), que faz com que a DMNI leia o pacote ao copiar o pacote da NoC à memória do PE. Uma vez que o pacote é recebido, o kernel executa rotinas relacionadas ao conteúdo do pacote.

2.2.6 Protocolo de Injeção de Aplicação Dinâmico

Periférico responsável por simular a solicitação de execução de aplicações por parte de um usuário. Basicamente, são inseridas aplicações no sistema em tempos definidos.

O processo se inicia com um AppInj requisitando a execução de uma nova aplicação ao enviar uma mensagem “NEW APP REQUEST” a um MPE com o número de tarefas da aplicação. A seguir tem-se, passo a passo, o processo:

1. A mensagem “NEW APP REQUEST” é endereçada ao MPE do cluster zero, que carrega essa mensagem e seleciona um cluster para executar a aplicação. Somente um MPE carrega essas requisições.

2. O MPE seleciona o cluster de acordo com algum critério programado, enviando uma mensagem “APP ACK” ao AppInj com o endereço do MPE selecionado a receber essa mensagem.

3. O AppInj envia uma mensagem “APP DESCRIPTOR”, com o grafo de tarefa de aplicação no payload dessa mensagem. Ao receber essa mensagem, o MPE executa o mapeamento da tarefa de aplicação.

4. Depois do mapeamento, o MPE envia uma mensagem “APP ALLOCATION REQUEST” ao AppInj com as tuples ID da tarefa, endereço do SPE

5. O AppInj transfere o código do objeto da tarefa aos SPEs, enviando uma mensagem “TASK ALLOCATION”, que contém o código de objeto da tarefa no seu payload. Quando um SPE qualquer recebe a “TASK ALLOCATION", ele configura a DMNI para copiar o código de objeto da tarefa à página de memória selecionada.

2.3 System C

System C é uma linguagem de modelagem e projeto de sistemas baseada em biblioteca C++. Pode-se dizer que System C é um grande conjunto de C++, logo todo programa em C++ é um programa válido em System C (EMBECOSM, 2018). Com isso, conclui-se que a linguagem é amigável a diferentes programadores, além de possuir versatilidade e aplicabilidade tanto a nível *software* quanto a nível *hardware*. Hoje em dia, projetistas, ao utilizar System C, conseguem projetar e integrar sistemas complexos de eletrônica de forma rápida e, ao mesmo tempo, garantindo que o sistema final vai atender aos requerimentos de projeto e desempenho.

Projetistas podem desenvolver especificações executáveis, que são programas, em C++ que apresentam o mesmo comportamento do sistema em questão, permitindo assim uma fácil exploração do espaço do projeto (BANERJEE; SUR, 2014). Assim sendo, o System C adiciona três características que faltam em C++ básico, sendo estas concorrências, tempo e comportamento reativo (BANERJEE; SUR, 2014). Algumas vantagens de utilizar uma especificação executável são:

- Evita inconsistências e erros e garante a especificação do sistema de forma plena.
- Permite uma interpretação não ambígua do sistema.

- Valida a funcionalidade e o desempenho do sistema e permite a criação de modelos de desempenho do sistema sob estudo.
- Utilização de um test harness ou test bench de uma especificação executável dada que permite um teste rápido e preciso e verificação do modelo que está sendo testado

A descrição do System C consiste em um conjunto de PEs e módulos de *hardware* que intercambiam dados entre si através de canais bidirecionais ou unidirecionais. A sua biblioteca provê diversas classes de canais de comunicação embutidos, além de o usuário poder projetar seu próprio canal. Essas classes de canais de comunicação fazem parte da biblioteca estendida do System C, sendo elas:

- A classe C++ sc_module, adequada para definir módulos de *hardware* que contêm processos paralelos. Existe um mecanismo que define funções que modelam threads de controle dentro de classes sc_module.
- Duas classes, sc_port e sc_export para representar pontos de conexão com origem e destino de uma sc_module.
- Uma classe sc_interface para descrever os serviços de software requeridas por uma sc_port ou provida de uma sc_export.
- Uma classe sc_prim_channel para representar os canais de conexão de portas.

Essas classes também possuem um conjunto de classes derivadas delas mesmas para representar e conectar tipos de canais em comum utilizadas no projeto, como sinais, s e sistemas FIFOs (abreviação do inglês *first in first out*). Essa biblioteca estendida ainda dispõe de conjunto abrangente de tipos para representar dados em lógicas *2-state* e *4-state*.

2.4 Redes Neurais Artificiais

Redes Neurais Artificiais (RNAs) têm como objetivo emular o cérebro humano ao aplicar métodos matemático-estatísticos em sua estrutura (*hardware* ou *software*) afim de estimular o seu aprendizado próprio. Elas são formadas por fortes comunicações internas que trabalham juntas para resolver problemas ou tarefas específicas (ALEXANDER, 2020), contidas em sua estrutura. A habilidade de aprender é facilmente a característica mais importante de uma RNA. Com essa característica de aprendizado, RNAs executam corretamente tarefas sem que sejam previamente programadas (MAKIUCHI, 2018).

2.4.1 Modelo de um Neurônio Artificial

Existem diferentes arquiteturas de neurônios artificiais. A seguir, serão mostradas arquiteturas amplamente utilizadas nas mais diversas aplicações para um melhor entendi-

mento do seu funcionamento. Além das arquitetures que serão mostradas, existem outras mais robustas para atender aplicações mais complexas. Dessa forma, neurônios artificiais possuem na maioria dos casos os seguintes propriedades em sua estrutura:

- Um conjunto de enlaces de conexão, cada uma possuindo seu próprio peso;
- Somador, que soma as entradas multiplicadas pelo respectivo peso determinando a entrada da função de ativação;
- Bias, que aumenta ou diminui a entrada da função de ativação;
- Função de Ativação, determinada pelo projetista, que aplicada à sua entrada determina o sinal de saída;

2.4.1.1 Neurônio Artificial de Uma Entrada

Na Figura 17, tem-se um modelo de um neurônio de uma entrada. Duas entradas chegam ao somador. Uma entrada p é multiplicada pelo escalar de peso w para formar wp, que é repassada ao somador. A outra entrada de valor 1 é multiplicada pelo bias b e é repassada por fim ao somador. A partir do somador, temos o neurônio em si (*General Neuron*). A saída n do somador é a entrada de uma função de transferência, chamada de função de ativação, e tem como saída a, que é considerada como a saída do neurônio (DEMUTH et al., 2014).

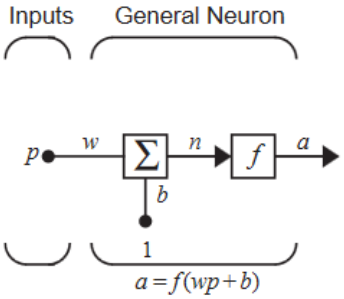

Figura 17 – Modelo de Neurônio Artificial de Uma Entrada (Fonte: (DEMUTH et al., 2014))

2.4.1.2 Neurônio Artificial de Múltiplas Entradas

Normalmente neurônios de uma entrada não são suficientes para o desenvolvimento de uma aplicação mais complexa. Por conta disso, existem neurônios de múltiplas entradas. A Figura 18 mostra esse tipo de neurônio. As múltiplas entradas que se tem são as entradas P_n multiplicadas por diferentes pesos W_n, sendo n o número da respectiva entrada no neurônio. O restante da estrutura mostrada na Figura 18 (*Multiple Input Neuron*), que é o próprio neurônio, funciona do mesmo modo da respectiva estrutura do neurônio de uma entrada. A saída n, neste caso, é representada pela forma matricial mostrada na Figura 18.

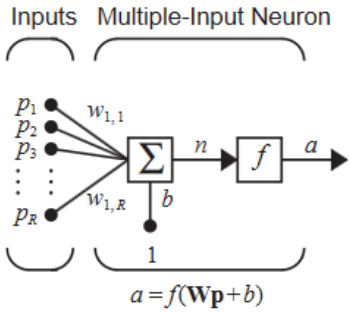

Figura 18 – Modelo de Neurônio Artificial de Múltiplas Entradas (Fonte: (DEMUTH et al., 2014))

2.4.2 Estrutura Geral de uma RNA

É fácil concluir que a estrutura das RNAs é semelhante às redes neurais biológicas. Ambas são compostas por camadas. Cada camada possui n entradas e m saídas. Existem diversos tipos de organização e configuração de camadas. Dentre estes, o tipo *feedforward* é amplamente utilizado. Uma rede neural *feedforward* de múltiplas entradas e uma saída é mostrada na Figura 19, e possui as seguintes camadas:

- Camada de entrada (*input layers*): recebe dados brutos introduzidos na rede (ALEXANDER, 2020).
- Camadas escondidas (*hidden layers*): a saída dessas camadas é determinada pela entrada, pelo peso da conexão entre eles e pelas camadas escondidas anteriores. Os neurônios enviam seus valores de saída para a função de ativação, e a saída desta é enviada para a próxima camada.
- Camadas de saída (*output layers*): Uma saída da camada de saída depende dos pesos da conexão entre as neurônios das camadas escondidas e desta saída.

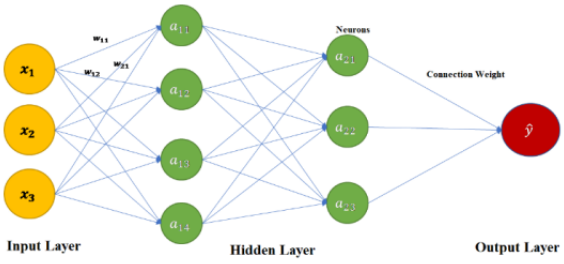

Figura 19 – Estrutura Geral de uma RNA (Fonte: (ALEXANDER, 2020))

3 Metodologia

Neste capítulo, será descrita a metodologia utilizada para atingir o objetivo proposto. Ou seja, serão detalhadas as etapas da implementação de um periférico de rede neural na plataforma Memphis. O sistema proposto é resumido pela representação gráfica mostrada na Figura 20.

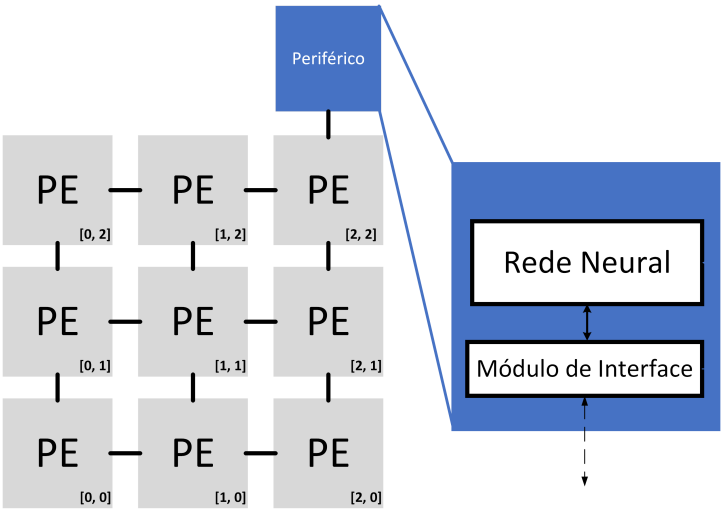

Figura 20 – Representação Gráfica do Sistema Proposto

Na Figura 20, PE indica um elemento de processamento qualquer da Memphis, o qual possui uma coordenada X e Y no eixo cartesiano (representada no canto inferior direito de cada PE). O periférico, representado em azul, tem a funcionalidade de permitir à Memphis o acesso aos recursos de uma rede neural. De modo geral, um periférico pode ser conectado a qualquer porta não utilizada nas bordas da rede, ou seja, as portas nortes dos PEs mais acima, nas portas sul dos PEs mais abaixo, na porte oeste dos PEs mais à esquerda a na porte leste dos PEs mais à direita. No exemplo da Figura 20, o periférico está conectado na porta norte do PE [2,2].

A Seção 3.1 apresenta o desenvolvimento do periférico, o qual é composto por dois módulos principais: Rede Neural e Módulo de Interfaceamento. Para esse trabalho, foi realizado um estudo de caso utilizando um módulo de rede neural capaz de obter a resposta de uma porta XOR, como descrito na Seção 3.1.1. Já o Módulo de Interfaceamento, detalhado na 3.1.2, é responsável por intermediar as conexões entre o módulo da rede neural e o roteador de um PE. Dessa forma, a seta dupla na Figura 20 representa a conexão entre o Módulo de Interfaceamento e o Módulo da Rede Neural e a seta dupla pontilhada representa a conexão entre o roteador do PE [2,2] e o Módulo de Interfaceamento.

Além disso, a Seção 3.2 apresenta as adaptações necessárias implementadas no

software da Memphis. Já a Seção 3.3 apresenta as adições necessárias implementadas no *hardware* da Memphis. Por fim, a Seção 3.4 mostra o processo de implementação da rede neural na linguagem C, a qual foi utilizada para fins de comparação entre a execução dessa rede neural em *hardware* e em software.

O sistema proposto visa permitir que uma aplicação utilize o periférico como uma ferramenta de obtenção da saída de uma RNA. Dessa forma, uma aplicação pode utilizar o periférico como forma de aceleração da obtenção da saída de uma rede neural comparada com a mesma implementada em *software*. Para que uma aplicação tenha acesso ao periférico, foi incluída uma nova primitiva na API do sistema. Além disso, a comnicação entre um PE e o periférico é feito através do envio e recebimento de pacotes. No periférico, o tratamento dos pacotes de solicitação do uso da rede é feito pelo Módulo de Interfaceamento descrito a seguir.

3.1 Implementação do Periférico

Conforme explicitado anteriormente, o periférico é composto por um Módulo de Rede Neural, descrito na seção 3.1.1 e um Módulo de Interfaceamento, detalhado na seção 3.1.2.

3.1.1 Implementação do Módulo de Rede Neural

Neste trabalho, foi feita a implementação de uma rede neural para estudos de caso. Essa implementação provém do trabalho de conclusão de curso "Desenvolvimento de rede neural artificial recorrente em FPGA para previsão online de oportunidades em transmissões oportunísticas em redes decomunicação wireless" (MAKIUCHI, 2018). Essa rede neural é do tipo *feedforward* e foi descrita na linguagem de descrição de *hardware* Verilog. A rede é representada pelo diagrama mostrado na Figura 21.

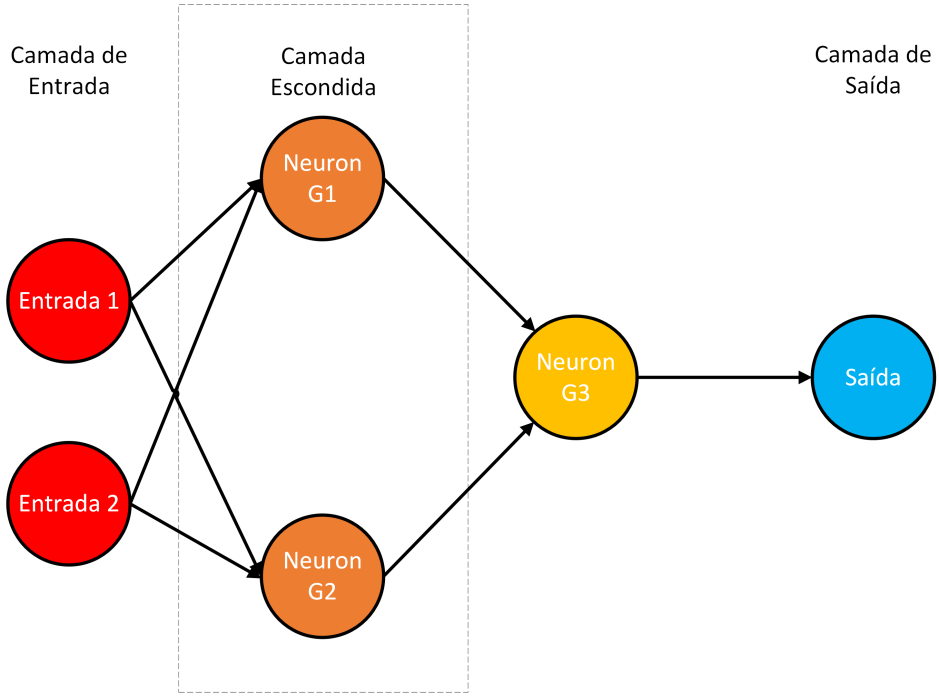

Figura 21 – Rede Neural *Feedforward* Utilizada

Como é visto na Figura 21, a RNA possui duas entradas (em vermelho). Essas entradas são de 16 bits em ponto fixo Q13. Além disso, possui uma camada escondida representada pelo retângulo pontilhado, e um neurônio que obtém a saída da camada escondida, representado pelo elemento em amarelo. Por fim, possui uma única saída também em ponto fixo Q13. Com isso, seus neurônios recebem como entrada seus respectivos valores de entrada determinados, seus pesos e o tipo de função de ativação que será utilizado.

Essa rede neural foi treinada para o caso de teste de implementação da porta XOR, ou ou-exclusivo. Dessa forma, os valores de saída absolutos tendem a ser próximos a 0 ou a 1, com essa proximidade variando de acordo com o valor de suas entradas.

3.1.1.1 Processo de Tradução da Rede Neural

Uma implementação em SystemC apresenta menor tempo de simulação quando comparado a Verilog. Com um menor tempo de simulação, busca-se obter decisões de projeto mais rapidamente, além de uma maior quantidade de testes de validação em um menor período de tempo. Além disso, a utlização de SystemC, por ser uma linguagem construída a partir de C++, facilita a transição do código do *hardware* para o *software*. Dessa maneira, pode-se mais facilmente comparar o custo entre uma implementação em *software* e *hardware* de uma mesma aplicação. Por esses motivos, a rede neural do trabalho supracitado (MAKIUCHI, 2018) foi traduzida para SystemC RTL para o uso neste trabalho. Neste caso, o processo foi feito manualmente. Para isso, foi necessário, inicialmente, separar os módulos em arquivos diferentes. A Tabela 1 mostra cada módulo e a sua função.

Tabela 1 – Funções dos Módulos da Rede Neural Traduzida em SystemC

Módulo	**Função**
nn	Módulo da Rede Neural Centralizada
network_g	Módulo de Conexão entre os Neurônios
neuron_g1	Módulo de Neurônio *Feedforward* da Camada Escondida
neuron_g2	Módulo de Neurônio *Feedforward* da Camada Escondida
neuron_g3	Módulo de Neurônio *Feedforward* a
activation_function	Módulo de Função de Ativação

Em seguida, é feito o interfaceamento separado para cada módulo com o objetivo de efetivar a comunicação entre estes. O módulo nn, inicialmente, centraliza a rede ao receber as entradas e direcioná-las ao módulo network_g e obter o redirecionamento da saída através do mesmo módulo.

Os módulos de neurônio neuron_g1 e neuron_g2 realizam o cálculo do seu estado interno através das entradas da rede neural. Já o módulo neuron_g3 realiza o cálculo do seu estado interno através da saída dos módulos neuron_g1 e neuron_g2.

O módulo network_g realiza as conexões entre os módulos de neurônio ao direcionar valores de entrada aos módulos neuron_g1, neuron_g2 e neuron_g3, receber a saída dos módulos neuron_g1 e neuron_g2 e direcioná-las ao módulo neuron_g3. Isso é visto no Código 3.1, em que Output1, Output2 e Output3 são as saídas dos módulos neuron_g1, neuron_g2 e neuron_g3, respectivamente, e o sinal Node direciona essas informações ao módulo neuron_g3. Em seguida, direciona a saída do módulo neuron_g à saída do módulo nn através do sinal Output.

```
1 void network_g::nodewrt(){
2   Node[0].write(0);
3   Node[1].write(Output1.read());
4   Node[2].write(Output2.read());
5   Node[3].write(Output3.read());
6
7   Output[1].write(Output3.read());
8 }
```

Código 3.1 – Trecho do Código do Módulo network_g

O módulo activation_function recebe o tipo de função de ativação a ser utilizado e os valores de estado interno dos módulos de neurônio. O módulo dispõe de diferentes funções de ativação aproximadas, que são representadas na Figura 22. Por fim, o módulo realiza os devidos cálculos e direciona a sua saída à saída do respectivo neurônio.

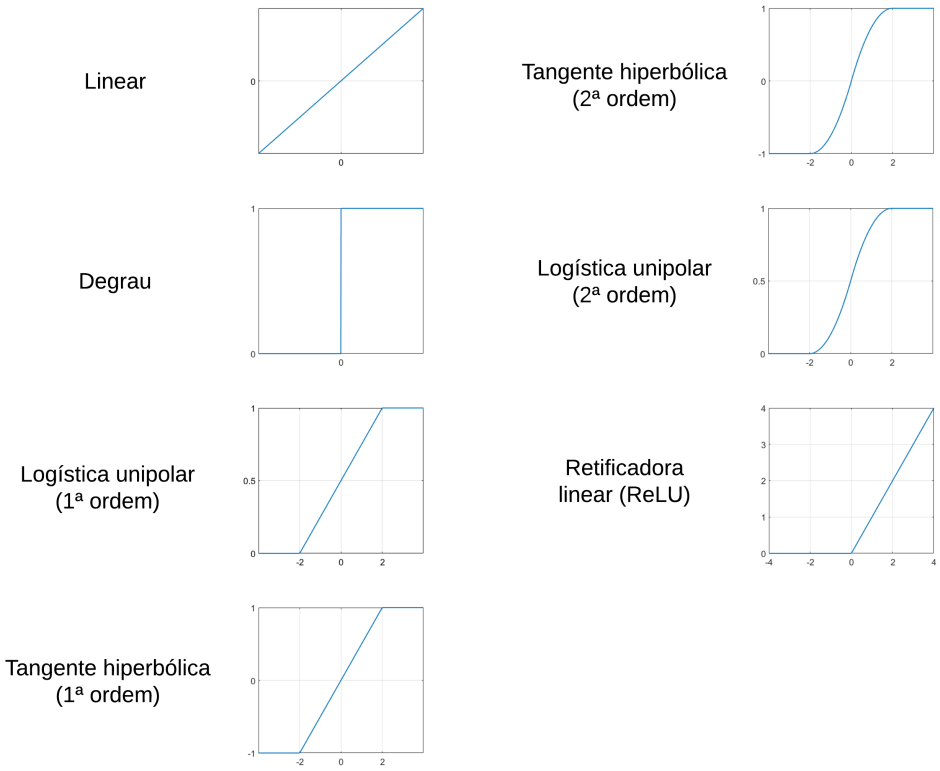

Figura 22 – Funções de Ativação Aproximadas (Fonte: (MAKIUCHI, 2018))

3.1.2 Inplementação do Módulo de Interfaceamento

A comunicação entre o periférico e os elementos da Memphis é orquestrada por um módulo de interface com os roteadores da rede presente no periférico. Esse módulo realiza a intermediação da conexão entre a rede neural e o elemento de processamento através dos sinais mostrados na Tabela 2.

Tabela 2 – Descrição de Sinais do Periférico

Sinal	Descrição
clock	Sinal global de clock
reset	Sinal global de reset
rx	Sinal que indica o recebimento de dados válidos
tx	Sinal que indica o envio de dados válidos
data_in	Sinal que contém os dados recebidos
data_out	Sinal que contém os dados enviados
credit_in	Sinal que indica que o elemento de processamento destino está apto a receber dados
credit_out	Sinal que indica que o periférico está apto a receber dados
sig_credit_out	Sinal que atualiza o valor do sinal credit_out
Entrada1	Sinal usado para enviar o valor da Entrada 1 da rede neural
Entrada2	Sinal usado para enviar o valor da Entrada 2 da rede neural
Saida1	Sinal usado para receber o valor da Saída da rede neural

Entre os sinais apresentados na Tabela 2, há os sinais utilizados na Memphis para o envio e recebimento de pacotes entre os roteadores. Os sinais tx, data_out e credit_in são responsáveis pela transmissão de dados. Já, os sinais rx, data_in e credit_out são responsáveis pelo recebimento de dados. Para implementar o controle de fluxo baseado por crédito implementado na Memphis, são utilizados os credit_in e credit_out. A Memphis também utiliza um chaveamento por pacotes do tipo wormhole, sendo pacotes enviados flit a flit. Dessa forma, o sinal data_in armazenará os flits de recebimento de pacote, enquanto o sinal data_out armazenará os flits de um pacote a ser enviado. Por fim, os sinal rx serve para indicar um recebimento de um flit de um pacote, enquanto o sinal tx alerta o roteador vizinho que um flit de um pacote será enviado.

Além dos sinais utilizados para comunicação, o periférico utiliza sinais para interfacear com as entradas (Entrada1 e Entrada 2) e saída (Saída1) da rede neural. Basicamente, o módulo de interface é responsável pelo recebimento de pacotes, com implementação detalhada na Seção 3.1.2.1, e pelo envio de pacotes, com implementação descrita na Seção 3.1.2.2, no periférico. Tanto a implementação de envio e recebimento de pacotes utilizam-se de Máquinas de Estado Finitas com sensibilidade no sinal de clock e no sinal de reset.

3.1.2.1 Recebimento de Pacotes

As solicitações realizadas ao periférico são feitas através de um pacote identificado pelo serviço NN_REQUEST. Como descrito anteriormente, um pacote é dividido em flits. Cada flit do pacote terá uma informação diferente baseada no serviço que ele implementa. A Figura 23 apresenta as informações contidas em cada flit de um pacote de serviço

NN_REQUEST. Como pode ser visto na figura, o serviço é identificado através do terceiro flit do pacote.

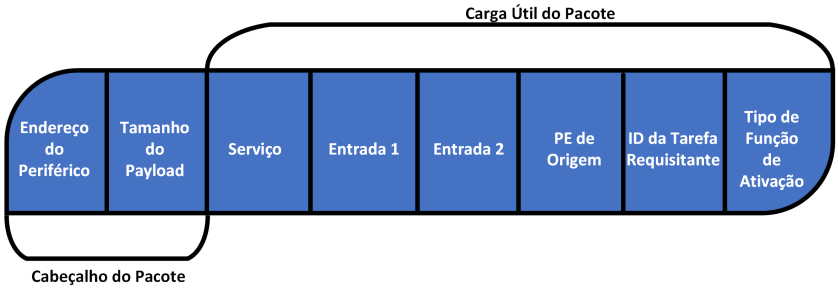

Figura 23 – Modelo do Pacote de Recebimento do Periférico

O módulo de interface implementa o recebimento desse tipo de pacote através da Maquina de Estados Finita de Recebimento. A MEF de Recebimento, primeiramente, utiliza vários estados para obter o conteúdo de cada flit de um pacote do tipo NN_REQUEST. Ao obter as informações das entradas da rede neural contidas no pacote, essa MEF as utiliza para interfaceamento com o módulo de rede neural para obtenção da saída. Após a obtenção da saída da rede, a MEF monta o pacote de resposta para que uma MEF de envio retorne ao PE que fez a solicitação ao periférico. A MEF de Recebimento conta com os sinais de clock, reset, rx e sig_credit_out para o seu funcionamento. Além disso, conta com a variável flit_counter, um contador que faz a contagem dos flits do payload, o qual facilita a obtenção ordenada do conteúdo de cada flit. O diagrama da Máquina de Estados de Recebimento é mostrado na Figura 24.

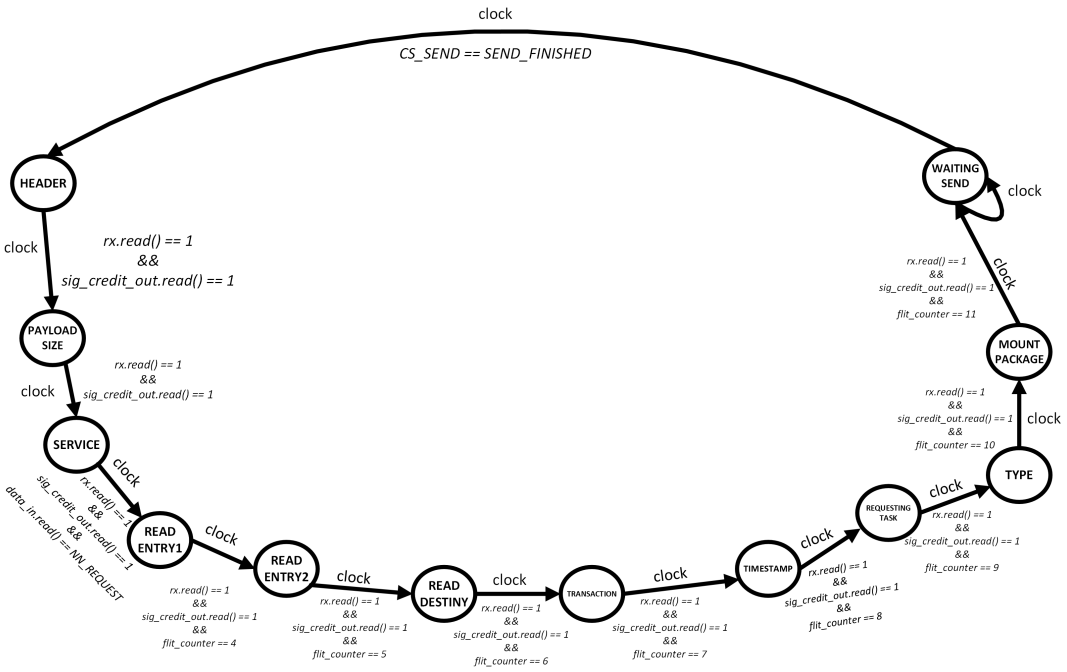

Figura 24 – Diagrama da MEF de Recebimento

Tem-se como estado inicial o estado HEADER. Nesse estado, é lido o cabeçalho do pacote de dados de recebimento. No estado PAYLOAD_SIZE é lido o tamanho do payload do pacote. O estado SERVICE verifica se o serviço do pacote, identificado por um número em hexadecimal, é de fato o serviço NN_REQUEST. Os estados READ_ENTRY1 e READ_ENTRY2 realizam a leitura das entradas 1 e 2, respectivamente, cujos valores serão os valores das entradas da rede neural. Já o estado READ_DESTINY realiza a leitura do destino do pacote de envio. Em seguida, passa-se para os estados TRANSACTION (sem conteúdo) e TIMESTAMP (carrega o *timestamp* de criação do pacote), que servem para a espera do recebimendo de dados arbitrários enviados pela Memphis que não tem utilidade para o periférico. No estado REQUESTING_TASK, é feita a leitura do id da tarefa que requisita o periférico. Logo depois, no estado TYPE, é feita a leitura do tipo de função de ativação da rede neural. Assim que chegam, os dados das entradas e do tipo de função de ativação são imediatamente colocados na rede neural. Sendo assim, como a saída da rede é obtida em um ciclo de clock, já é possível montar o pacote de envio inteiro. Isso é feito no estado seguinte, que é o estado MOUNT_PACKAGE, num vetor. Por fim, o estado WAITING_SEND espera o pacote ser enviado para retornar para o estado HEADER e o ciclo ser reiniciado. Esse estado habilita o ínicio do envio do pacote por parte da MEF de envio. Todas as leituras são feitas por meio do sinal data_in.

3.1.2.2 Envio de Pacotes

O envio do pacote de volta para o sistema é feito através de um pacote identificado pelo serviço NN_SEND. A Figura 25 apresenta as informações contidas em cada flit de um pacote de serviço NN_SEND.

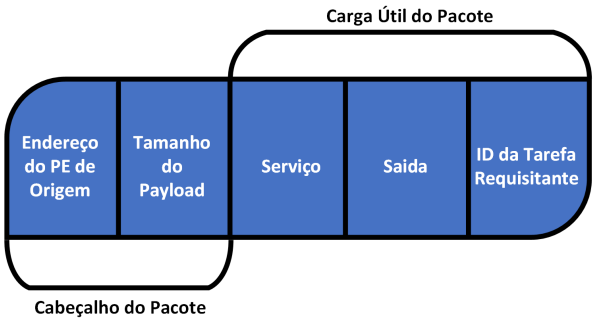

Figura 25 – Modelo do Pacote de Envio do Periférico

A MEF de Envio conta com os sinais credit_in e tx para o seu funcionamento. Além disso, conta com a variável packet_size, que controla o fluxo de flits enviados ao ser decrementada. A MEF é mostrada na Figura 26.

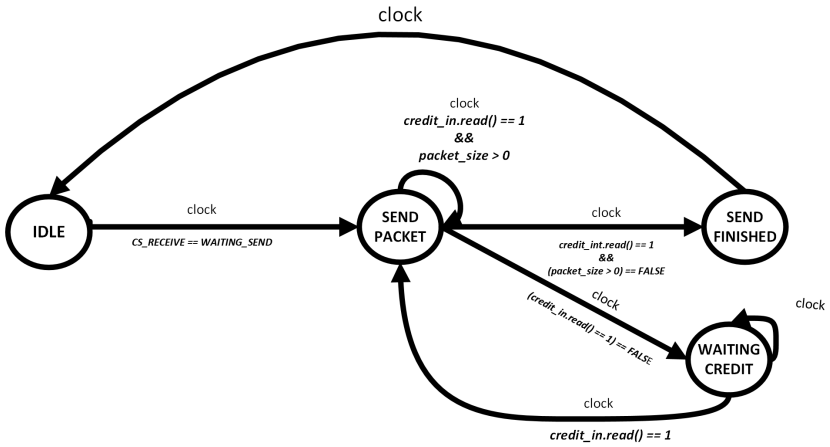

Figura 26 – Diagrama da MEF de Envio

Tem-se como estado inicial o estado IDLE. A MEF fica nesse estado até que o pacote de envio termine de ser montado, ou seja, quando a MEF de Recebimento chegar o estado WAITING_SEND. Caso isso aconteça, ocorre a transição para o estado SEND_PACKET. Nesse estado, um flit do pacote de envio montado é enviado a cada borda positiva de sinal de clock posição por posição até que todos os flits sejam enviados. Isso é feito ao decrementar uma variável que posssua o valor do tamanho do payload do pacote, que é lido no segundo estado da MEF de recebimento. Se em algum momento o PE destino não possa receber dados, ou seja, credit_in.read() é igual a 0, a MEF passa para o estado WAITING_CREDIT, em que permanece até que o PE destino esteja habilitado a receber dados. Nesse último caso, o transiciona-se de volta para o estado SEND_PACKET. Quando todos os flits são enviados, passa-se para o estado SEND_FINISHED. A passagem para esse estado permite que o ciclo do periférico seja reiniciado.

3.1.2.3 Utilização de Diferentes Redes Neurais

Esse estudo de caso trata de uma rede neural simples, com duas entradas e uma saída. O Código 3.2 apresenta um trecho do portmap do periférico no qual está presente a interface da rede neural.

```
1 sc_signal< sc_int< 16 > > Entrada1;
2 sc_signal< sc_int< 16 > > Entrada2;
3 sc_signal< sc_int< 16 > > Saida1;
4
5 sc_signal< sc_int< 4> > tipo;
```

Código 3.2 – Portmap da Rede Neural no Periférico

Assim sendo, é possível incluir outra rede neural nesse periférico. Para isso, informados os números de ciclos de clock para obtenção da resposta, pode-se implementar novos sinais de

entrada e saída. Para isso, no caso do *hardware*, são necessárias alterações nas MEFs.

3.2 Alterações no Software da Memphis

O *software* da Memphis não conta com todos os recursos para o funcionamento do periférico projetado quando se trata da comunicação entre tarefas e o periférico. Logo, é necessário realizar a adição desses recursos. Dessa forma, serão apresentados os recursos adicionados e as modificações no *software* da Memphis nas seções seguintes.

3.2.1 Adição de Novos Serviços de Pacotes

Neste trabalho, foram adicionados dois pacotes identificado pelos serviços: NN_REQUEST e NN_SEND. Esses serviços foram adicionados no arquivo de serviços services.h, o qual define códigos que identificam os serviços para a posterior utilização na estrutura do pacote. Cada pacote tem um campo que identifica o serviço através do seu código.

3.2.2 Adição de Campos de Pacote

A Memphis conta com um arquivo packet.h que implementa a estrutura de dados dos pacotes, dividido-os em campos (como o campo de serviço, por exemplo). Com isso, cada flit do pacote é correspondente a um campo, que são definidos para diferentes utilizações. Para o caso do uso do periférico, foi necessária a adição dos campos para a entrada 1, entrada 2, saída e o tipo de função de ativação. Com isso, os pacotes trabalhados funcionam conforme mostrado na Figura 23 e na Figura 25.

3.2.3 Adições no Arquivo de API

A Memphis possui um arquivo que implementa a API, a qual provê primitivas de chamadas de sistema para uso no código de tarefas de aplicações. Além disso, nesse arquivo é definida uma estrutura de dados chamada *Message*, utilizada para troca de mensagens entre tarefas (conforme apresentado na Seção 2.2.2), além de servir de argumento em algumas chamadas de sistema. Assim, foi implementada uma primitiva para uma nova chamada de sistema NNPeriph, a qual fica sendo chamada até que a tarefa receba a saída rede neural provinda do periférico. Isso é mostrado no Código 3.3.

```
1 #define NNPeriph(msg, out_msg) while(!SystemCall(NNPERIPH,
    (unsigned int*)msg, (unsigned int*)out_msg,0))
```

Código 3.3 – Chamada de Sistema NNPeriph

Nota-se a presença de dois argumentos: msg e out_msg,implementados em uma estrutura de dados do tipo *Message*. Essa estrura contém um vetor de inteiros de tamanho

pré-determinado (128 posições por padrão), além de um campo que define quantas posições desse vetor de inteiros foram utilizadas. A utilização dos argumentos implementados que usam essa estrutura de dados permite o envio e recebimento de um número parametrizável de dados através desse vetor de inteiros.

Assim, para o uso dessa primitiva, deve-se definir no vetor de inteiros do argumento msg as entradas da rede neural, além do tipo de função de ativação utilizada. Para o correto funcionamento da primitiva, a definição dessas informaçãoes deve ser feito em posições específicas do vetor, conforme visto no modelo da Figura 27. No caso da rede neural utilizada nesse trabalho, há duas entradas (entrada 1 e entrada 2). A entrada 1 deve implementada na posição 0, a entrada 2 na posição 1 e o tipo de função de ativação na posição 2 do vetor. Vale ressaltar que a utilização do vetor com tamanho parametrizável flexibiliza o uso dessa primitiva para redes neurais que possuam mais entradas, por exemplo.

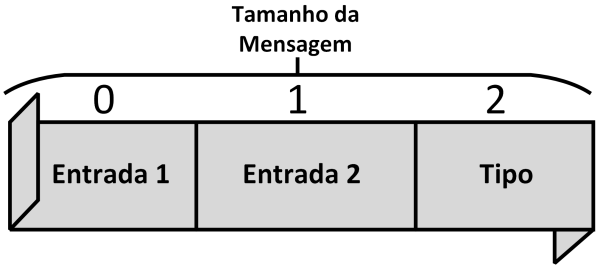

Figura 27 – Modelo de Mensagem de Entrada

Ao término da primitiva NNPeriph, o valor de saída da rede neural é encontrado na posição 0 do vetor do argumento out_msg. Isso pode ser visto na representação gráfica da Figura 28. Como mencionado anteriormente, a implementação utilizando esse vetor com tamanho parametrizável flexibiliza o uso da primitiva também para redes neurais que possuam mais saídas.

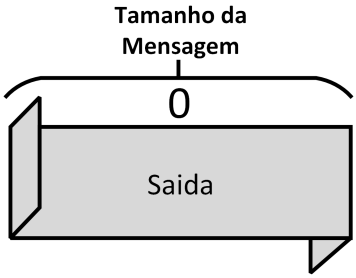

Figura 28 – Modelo de Mensagem de Saída

3.2.4 Módulo Peripheral Communications

Esse módulo tem como função criar um vetor para controlar as solicitações de uso do periférico. Cada posição do vetor armazena uma solicitação ao periférico através de uma

estrutura de dados chamada PeripheralRequests, que possui os seguintes campos:

- peripheral_location: armazena a localização do periférico na Memphis;
- requesting_task: armazena o id da tarefa requisitante;
- status_peripheral: armazena o estado de uma solicitação, o qual pode ser P_EMPTY (espaço de memória vazio, ou seja, sem solicitação) ou P_WAITING(esperando a resposta do periférico);
- message_ptr: ponteiro que aponta para o segundo argumento da primitiva NNPeriph (out_msg). Esse campo é utilizado para facilitar o retorno da saída da rede neural para a tarefa solicitante.

O módulo conta com algumas funções para o manejamento das posições do vetor. A função init_Pcommunication mostrada no Código 3.4 inicia o vetor, definindo todas as suas posições com estado de vazias P_EMPTY).

```
1  void init_Pcommunication(){
2    for(int i=0; i<PERIPHERAL_SIZE; i++){
3      peripheral[i].status_peripheral = P_EMPTY;
4    }
5  }
```

Código 3.4 – Função init_Pcommunication

O Código 3.5 apresenta a funcão get_PERIPHERAL_free_position. Essa função tem como objetivo procurar um espaço disponível no vetor, percorrendo-o até encontrar um espaço vazio (com estado P_EMPTY), coforme é visto nas linhas 3 a 6.

```
1  PeripheralRequests * get_PERIPHERAL_free_position(){
2    PeripheralRequests * peripheral_ptr;
3    for(int i=0; i<PERIPHERAL_SIZE; i++){
4      peripheral_ptr = &peripheral[i];
5      if (peripheral[i].status_peripheral == P_EMPTY){
6        return &peripheral[i];
7      }
8    }
9    return 0;
10 }
```

Código 3.5 – Função get_PERIPHERAL_free_position

Já o Código 3.6 apresenta a função peripheral_search_and_return, a qual possui como argumentos um id da tarefa requisitante (argumento requesting_task) e a localização do

periférico (argumento peripheral_location). Essa função tem como objetivo procurar espaços não vazios no vetor (com estado P_WAITING) cujo id da tarefa seja igual ao argumento requesting_task, e o periférico esteja na posição igual ao argumento peripheral_location . Ela percorre o vetor até encontrar o espaço ocupado correspondente ao id da tarefa e à localização do periférico que não esteja vazio, conforme visto nas linhas 4 a 6.

```
1 PeripheralRequests * peripheral_search_and_return(int
     requesting_task, int peripheral_location){
2   PeripheralRequests * peripheral_ptr;
3   for(int i=0; i<PERIPHERAL_SIZE; i++)
4     peripheral_ptr = &peripheral[i];
5     if ((peripheral_ptr->status_peripheral ==
         peripheral_ptr->status_peripheral == P_WAITING &&
         peripheral_ptr->requesting_task == requesting_task  &&
         peripheral_location == peripheral_ptr->peripheral_location){
6       return peripheral_ptr;
7     }
8   }
9   return 0;
```

Código 3.6 – Função peripheral_search_and_return

As funções desse módulo serão utilizadas nas alterações realizadas no kernel servo, como descrito na seção a seguir.

3.2.5 Alterações no Arquivo de Kernel Servo

No arquivo de Kernel Servo estão contidas funções para execução de tarefas da Memphis. Para o funcionamento do periférico, foi necessária a modificação da função de tratamento de pacotes (handle_packet), a adição da primitiva de envio de pacotes de serviço NN_REQUEST chamada send_nn_request e a modificação da função de chamadas de sistema (syscall). Isso será explicado nas sessões a seguir.

3.2.5.1 Função de Chamadas de Sistema

O Kernel Servo conta com a Função de Chamadas de Sistema (syscall), que tem como propósito tratar as chamadas de sistemas definidas pelas primitivas do arquivo de API. A chamada de sistema NNPeriph implementada tem o objetivo de mediar a comunicação entre uma tarefa e o periférico. Como explicado anteriormente, a tarefa chama a primitiva NNPeriph. É utilizado o argumento msg dessa primitiva para definir as entradas e o tipo de função de ativação, sendo recebido como retorno do periférico a saída da rede neural, contida no argumento out_msg da primitiva.

Como relatado acima, os argumentos msg e out_msg são estrutura de dados do tipo *Message*. Esses argumentos são movimentados confome mostrado no Código 3.7. A função do condicional da linha 2 é verificar se a DMNI está em processo de envio de pacotes. A chamada de sistema somente inicia sua execução caso a DMNI não esteja utilizada, para evitar problemas de deadlock. Inicialmente, a variável requesting_task recebe o id da tarefa requisitante, através do ponteiro current que aponta para a estrutura TCB, que guarda informações pertinentes da tarefa. Em seguida, procura-se se já existe a solicitação da tarefa requisitante ao periférico da rede neural através da função peripheral_search_and_return. Caso não seja encontrada, ou seja, pr seja nulo (linha 7), uma nova solicitação é criada, primeiramente, buscando-se uma posição livre no vetor de solicitações através da primitiva get_PERIPHERAL_free_position() (linha 8). Então, os campos da estrutura de dados daquela posição do vetor de solicitações são preenchidos, conforme mostrado nas linhas de 9 a 12. Na linha 14 utiliza-se um ponteiro msg_read que aponta para uma estrutura do tipo *Message*, que guarda o argumento msg da primitiva. Para obtenção desse argumento, é utilizado um ponteiro para a posição de memória dada pela soma do endereço do offset da tarefa (endereço de memória inicial de onde a tarefa está alocada) com o endereço da mensagem. Por fim, é chamada a primitiva send_nn_request, que realiza o envio de um pacote do tipo NN_REQUEST ao periférico com a solicitação da tarefa.

```
1  case NNPERIPH:
2    if ( MemoryRead(DMNI_SEND_ACTIVE) ){
3      return 0;
4    }
5    requesting_task =  current->id;
6    pr = peripheral_search_and_return(requesting_task,
       NEURAL_PERIPHERAL);
7    if (pr == 0){
8      pr = get_PERIPHERAL_free_position();
9      pr->peripheral_location = NEURAL_PERIPHERAL;
10     pr->requesting_task = requesting_task;
11     pr->status_peripheral = P_WAITING;
12     pr->message_ptr = arg1;
13
14     msg_read = (Message *)((current->offset) | arg0);
15     send_nn_request(requesting_task, msg_read);
16     schedule_after_syscall = 1;
17     return 0;
18   }
19   schedule_after_syscall = 1;
20   return 0;
```

Código 3.7 – Código da Chamada de Sistema NNPeriph

3.2.5.2 Adição da Primitiva send_nn_request

Essa função tem como objetivo de montar um pacote de serviço NN_REQUEST e enviá-lo para o periférico, conforme é visto no Código 3.8. O pacote é montado a partir de argumentos definidos na chamada de sistema NNPeriph. Na linha 3, o ponteiro p aponta para uma estrutura de um pacote. Nas linhas 3 a 9, é feito o tratamento da mensagem e são preenchidos os campos do pacote. A primitiva send_packet abstrai o envio do pacote para o periférico.

```
1  void send_nn_request(int requesting_task, Message *msg_read){
2    ServiceHeader *p;
3    p = get_service_header_slot();
4    p->header = NEURAL_PERIPHERAL;
5    p->service = NN_REQUEST;
6    p->entrada1 = msg_read->msg[0];
7    p->entrada2 = msg_read->msg[1];
8    p->requesting_task = requesting_task;
9    p->tipo = msg_read->msg[2];
10
11   send_packet(p, 0, 0);
12 }
```

Código 3.8 – Código da Função send_nn_request NN_SEND

3.2.5.3 Função de Tratamento de Pacotes

A função de tratamento de pacote na Memphis se chama handle_packet. O propósito dessa função é fazer o tratamento de pacotes recebidos, decidindo o que fazer com ele baseado nas informações contidas nos seus campos. Essa função identifica o serviço correspondente ao pacote através do campo service. Em seguida, faz o tratamento do pacote de acordo com o seviço identificado.

Nessa função foi íncluído o tratamento de pacotes do serviço NN_SEND, abordado na Seção 3.1.2.2. Os pacotes desse serviço são enviados pelo periférico como retorno à solicitação das tarefas, enviando a saída da rede neural.

No tratamento do serviço NN_SEND, mostrado no Código 3.9, são preenchidos os valores do argumento out_msg, utilizado na primitiva de chamada de sistema NNPeriph, para retorno do valor de saída da rede neural. Para isso, primeiramente, utiliza-se um ponteiro para a estrutura TCB para obter informações referentes à tarefa requisitante, conforme mostrado na linha 2. Isso é realizado para a obtenção do offset da tarefa requisitante, ou seja, o endereço inicial de memória dessa tarefa na memória.

Depois, a solicitação da tarefa requisitante ao periférico é procurada no vetor de solicitações (linha 3). Encontrando-se a solicitação, o argumento out_msg é obtido (linha 5) através de um ponteiro que soma do endereço do offset da tarefa requisitante com o endereço desse argumento guardado previamente no campo message_ptr da respectiva estrutura de dados da solicitação. O valor da saída da rede neural é obtido através de um campo do pacote NN_SEND proveniente do periférico e é armazenado no argumento out_msg (linha 7).

Por fim, na linha 8, é modificado o registrador de retorno de função para 1 na estrutura TCB, que guarda o contexto da tarefa. Isso é realizado para indicar que a resposta da rede neural foi obtida na primitiva da chamada de sistema NNPeriph. Lembrando que a tarefa fica bloqueada nessa primitiva até o retorno da resposta da rede neural. Na linha 10 é verificado se há alguma tarefa em execução. Caso não haja, é solicitado que o escalonador coloque outra tarefa em execução (linha 11).

```
1  case NN_SEND:
2    tcb_ptr = searchTCB(p->requesting_task_back);
3    pr = peripheral_search_and_return(p->requesting_task_back,
       NEURAL_PERIPHERAL);
4    if(pr->status_peripheral == P_WAITING){
5      msg_ptr = (Message *)((tcb_ptr->offset) | (pr->message_ptr));
6      msg_ptr->length = 1;
7      msg_ptr->msg[0] = p->saida;
8      pr->status_peripheral = P_EMPTY;
9      tcb_ptr->reg[0] = 1;
10     if (current == &idle_tcb){
11       need_scheduling = 1;
12     }
13
14   }
15   break;
```

Código 3.9 – Código do Serviço NN_SEND

Em seguida, é terminada a chamada de sistema. Um resumo do processo de comunicação entre uma tarefa e um periférico é mostrado na Figura 29.

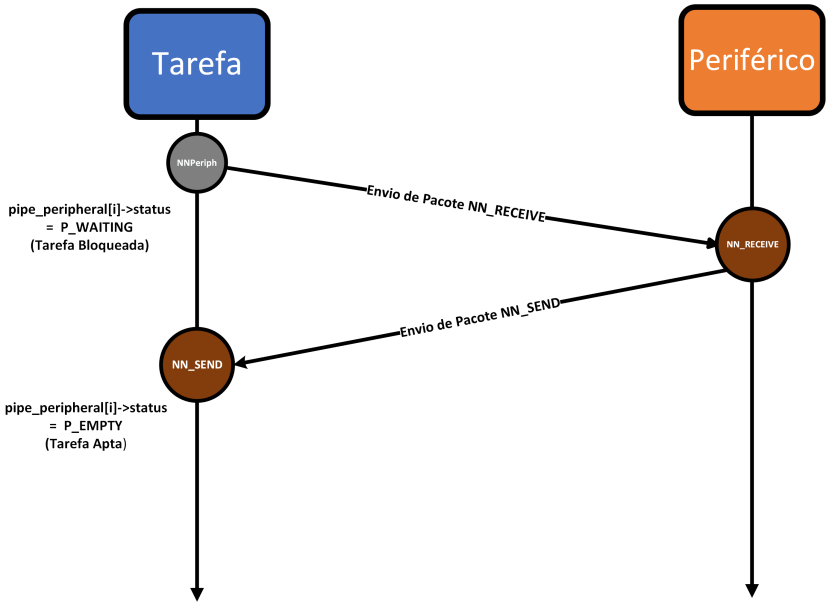

Figura 29 – Diagrama da Comunicações entre o Periférico e uma Tarefa

3.3 Alterações no Hardware da Memphis

Após desenvolver o periférico com a interface de um roteador da NoC Hermes, é necessário conectá-lo à Memphis. Isso é feito realizando mudanças no arquivo de *testbench* e no arquivo principal da Memphis.

3.3.1 Adições no *Testbench*

Inicialmente, criaram-se os sinais que ligam o periférico ao GPPC da Memphis no *testbench* da Memphis. Esses sinais são apresentados no Código 3.10, que é um trecho do código do *testbench*.

```
1 sc_signal<bool>    neural_peri_tx;
2 sc_signal<bool>    neural_peri_credit_i;
3 sc_signal<regflit>  neural_peri_data_out;
4 sc_signal<bool>    neural_peri_rx;
5 sc_signal<bool>    neural_peri_credit_o;
6 sc_signal<regflit>  neural_peri_data_in;
```

Código 3.10 – Sinais que Ligam o Periférico ao GPPC

Em seguida, foi devido instanciar o periférico no *testbench*. O instanciamento é mostrado no Código 3.11.

```
1 nnp = new neural_peripheral("Neural_Peripheral");
2   nnp->clock(clock);
3   nnp->reset(reset);
```

```
4   nnp->rx(neural_peri_tx);
5   nnp->data_in(neural_peri_data_out);
6   nnp->credit_out(neural_peri_credit_i);
7   nnp->tx(neural_peri_rx);
8   nnp->data_out(neural_peri_data_in);
9   nnp->credit_in(neural_peri_credit_o);
```

Código 3.11 – Sinais que Ligam o Periférico ao GPPC

3.3.2 Adições no Arquivo Principal da Memphis

Para conectar a interface do periférico ao roteador, é necessário, inicialmente, inserir no *portmap* presente no arquivo memphis.h a interface com o periférico. Isso é mostrado no Código 3.12.

```
1 sc_out< bool >      memphis_neural_peri_tx;
2 sc_in< bool >     memphis_neural_peri_credit_i;
3 sc_out< regflit >   memphis_neural_peri_data_out;
4
5 sc_in< bool >     memphis_neural_peri_rx;
6 sc_out< bool >      memphis_neural_peri_credit_o;
7 sc_in< regflit >    memphis_neural_peri_data_in;
```

Código 3.12 – Sinais de Interfaceamento com o Periférico

Em seguida, conecta-se a interface do periférico com o roteador. Deve-se inicialmente alterar a lista de sensitividade da Memphos. Essa alteração é mostrada no Código 3.13.

```
1
2 sensitive << memphis_neural_peri_tx;
3 sensitive << memphis_neural_peri_credit_i;
4 sensitive << memphis_neural_peri_data_out;
5 sensitive << memphis_neural_peri_rx;
6 sensitive << memphis_neural_peri_credit_o;
7 sensitive << memphis_neural_peri_data_in;
```

Código 3.13 – Implementação da Lista de Sensitividade do Periférico com o Roteador

Por fim, deve-se realizar a conexão no arquivo memphis.cpp da fiação do periférico com o roteador. Deve-se escolher no desenvolvimento do código a orientação da porta (norte, sul, leste e oeste) conforme visto no funcionamento da topologia da Memphis. O projeto de fiação é mostrado no Código 3.14.

```
1 if (i == NEURAL_PERIPHERAL && io_port[i] != NPORT) {
```

```
  p = io_port[i];
  memphis_neural_peri_tx
  .write(tx[NEURAL_PERIPHERAL][p].read());
  memphis_neural_peri_data_out
  .write(data_out[NEURAL_PERIPHERAL][p].read());
    credit_i[NEURAL_PERIPHERAL][p]
    .write(memphis_neural_peri_credit_i.read());

    rx[NEURAL_PERIPHERAL][p].write(memphis_neural_peri_rx.read());
    memphis_neural_peri_credit_o
    .write(credit_o[NEURAL_PERIPHERAL][p].read());
    data_in[NEURAL_PERIPHERAL][p]
    .write(memphis_neural_peri_data_in.read());
 }
```

Código 3.14 – Projeto de Fiação do Periférico na Memphis

Nesse caso, p é a porta do periférico, i a coordenada de um elemento de processamento e NEURAL_PERIPHERAL se trata de uma macro que contém uma referência ao periférico na Memphis. Essa referência é feita ao editar o arquivo yaml da Memphis que contém as informações de *testcase*. Para este trabalho, foi adicionado a este arquivo o trecho de código mostrado na Figura 30.

```
Peripherals:
 - name: NEURAL_PERIPHERAL
   pe: 2,2
   port: N
```

Figura 30 – Trecho do Código do Arquivo de *Testcase*

3.4 Rede Neural em Formato de Software

Para fins de estudo de caso, foi feito também a implementação da rede neural em verilog em linguagem C. Isso foi feito com o objetivo de inserir o código em C da rede neural em uma aplicação da Memphis. O processo foi simples, por conta da proximidade da linguagem SystemC com a linguagem C. Dessa forma, foram implementadas 3 funções em C que serão mostradas ou comentadas a seguir.

A função principal é a nn_peripheral, apresentada pelo Código 3.15, que recebe as entradas e o tipo de função de ativação, executa as instruções necessárias e retorna a saída.

```
int16_t nn_peripheral(int16_t Entrada1, int16_t Entrada2, int
   tipo){

    int Weight1[PESOS];
```

```
4       int Weight2[PESOS];
5       int Weight3[PESOS];
6
7       int32_t inner;
8
9       int16_t InnerState;
10      int16_t Node[NODES+1];
11
12      int16_t Saida;
13
14      assign_weights(&Weight1, &Weight2, &Weight3); //passa por
           "referencia" os pesos e coloca os valores
15
16      Node[0] = 0;
17
18      inner = UM*Weight1[0] + Entrada1*Weight1[1] +
           Entrada2*Weight1[2];
19      inner = inner >> 13;
20
21      InnerState = ((1 << 16) - 1) & (inner >> (0)); //pega os 16
           primeiros bits de inner
22      Node[1] = activation_function(InnerState, tipo);
23
24      inner = UM*Weight2[0] + Entrada1*Weight2[1] +
           Entrada2*Weight2[2];
25      inner = inner >> 13;
26
27      InnerState = ((1 << 16) - 1) & (inner >> (0));
28      Node[2] = activation_function(InnerState, tipo);
29
30      inner = UM*Weight3[0] + Node[1]*Weight3[1+INPUT] +
           Node[2]*Weight3[2+INPUT];
31      inner = inner >> 13;
32
33      InnerState = ((1 << 16) - 1) & (inner >> (0));
34      Node[3] = activation_function(InnerState, tipo);
35
36      Saida = Node[3];
37
38      return Saida;
39  }
```

Código 3.15 – Função nn_peripheral

Já a função assign_weights atribui os pesos da rede neural, e a função activation_function realiza as operações das funções de ativação implementadas de acordo com as seus argumentos.

4 Resultados

Neste capítulo são apresentados os resultados de validação individual do funcionamento do periférico implementado, vistos na Seção 4.1. Além disso, será feita a validação do sistema proposto com o periférico acoplado à plataforma Memphis na Seção 4.2. Nessa última seção também serão avaliadas métricas de desempenho relativas ao sistema proposto, como por exemplo, a comparação da implementação de rede neural em *hardware* e *software* e a validação do tráfego de flits na rede.

4.1 Cenários de Validação do Periférico

Essa seção tem como intuito avaliar individualmente o funcionamento do periférico desenvolvido nesse trabalho. Como descrito anteriormente, o periférico é composto por um módulo de rede neural, validado na Seção 4.1.1, e um módulo de interface, validado na Seção 4.1.2.

4.1.1 Validação do Módulo de Rede Neural

Nesta seção, serão mostrados os resultados provenientes do processo de verificação do funcionamento da rede neural traduzida. Como já informado, a rede neural se comporta como uma porta XOR, com entradas de 16 bits em ponto fixo Q13. Para verificar o funcionamento adequado da conversão, foi desenvolvido um testbench interfaceado com o módulo da rede neural. Uma vez que deseja-se observar o comportamento da saída de uma porta XOR, atribuiram-se valores próximos de 1 e 0 para as entradas. Isso foi feito no método input_define, com espaço de 100 nanosegundos entre atribuições de novos valores de entrada. Isso pode ser visto no Código 4.1.

```
1  void tb::input_define(){
2
3      tipo.write(1);
4
5      Entrada1.write("0x0000");
6      Entrada2.write("0x0000");
7
8      wait (100, SC_NS);
9
10     Entrada1.write("0x0000");
11     Entrada2.write("0x3FFF");
12
```

```
13      wait (100, SC_NS);
14
15      Entrada1.write("0xF000");
16      Entrada2.write("0x0000");
17
18      wait (100, SC_NS);
19
20      Entrada1.write("0xFFFF");
21      Entrada2.write("0xFFFF");
22
23  }
```

Código 4.1 – Método de Definição de Entradas do Testbench da Rede Neural

Para fins de comparação, foram realizadas simulações da RNA em Verilog e em SystemC. Essas simulações foram feitas para dois tipos de função de ativação. Inicialmente, Figura 31 mostra formas de onda resultante da simulação da rede neural descrita em Verilog, ao passo que a Figura 32 mostra formas de onda resultante da simulação da rede neural implementada nesse trabalho em SystemC. Ambas utilizam a função de ativação linear. Pode-se constatar que os mesmos valores de saída foram obtidos em ambas formas de onda.

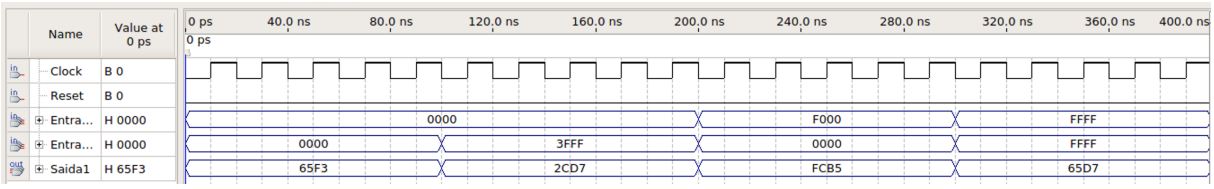

Figura 31 – Formas de Onda dos Sinais da Rede Neural com Função de Ativação Tipo Linear em Verilog

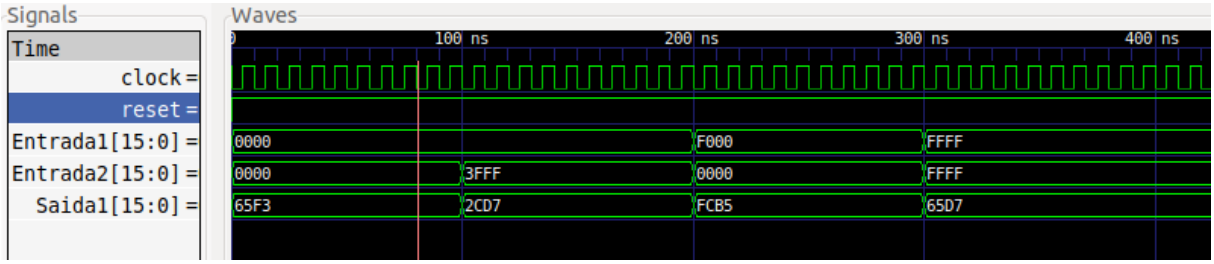

Figura 32 – Formas de Onda dos Sinais da Rede Neural com Função de Ativação Tipo Linear em SystemC

Além disso, pode-se ver que as formas de onda também são iguais para a RNA com o tipo de função de ativação sigmoide logística (LogSig) Unipolar com 1 segmento de reta na Figura 33 para a RNA em Verilog e na Figura 34 para a RNA em SystemC.

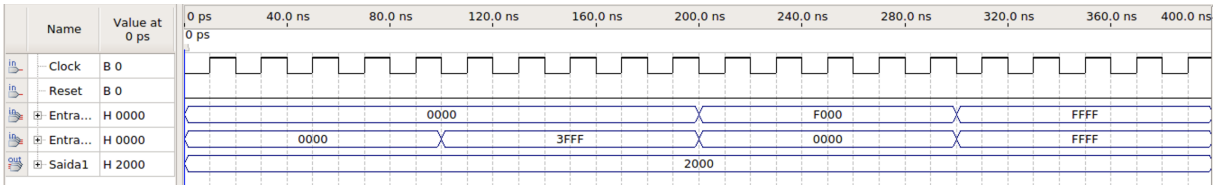

Figura 33 – Formas de Onda dos Sinais da Rede Neural com Função de Ativação Tipo LogSig Unipolar em Verilog

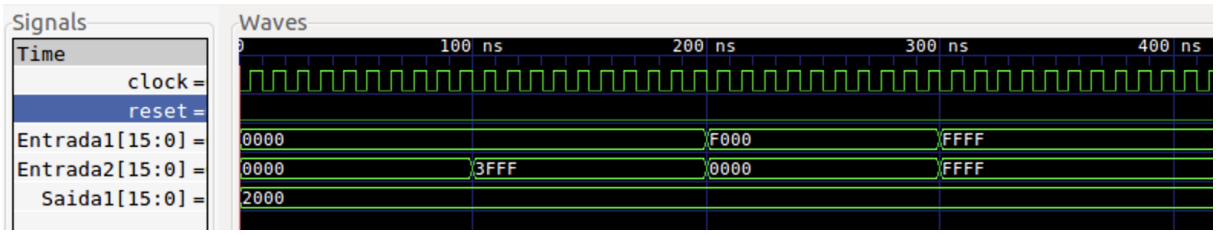

Figura 34 – Formas de Onda dos Sinais da Rede Neural com Função de Ativação Tipo LogSig Unipolar em SystemC

Nesse segundo teste, é importante comentar que os pesos da RNA fornecida em Verilog não são provenientes de treinamento para esse tipo de função de ativação. Logo, nota-se que as saídas obtidas após a aplicação desse tipo de função de ativação não valem para uma porta XOR. Dessa forma, esse teste tem como único objetivo validar somente a tradução desenvolvida para SystemC. Portanto, de acordo com os resultados acima, o Módulo de Rede Neural funciona corretamente e foi validado.

4.1.2 Validação do Módulo de Interfaceamento

Nesta seção, serão mostrados os métodos e resultados da validação do funcionamento do módulo de interfaceamento, responsável pelo recebimento e envio de pacotes no periférico. Para testar o módulo de interfacemento, inicialmente, foi desenvolvido um módulo que para envio e recebimento de pacotes ao periférico, com o objetivo de simular de forma simplificada o fluxo de pacotes da Memphis. Em seguida, foi feito um testbench para esse módulo.

4.1.2.1 Módulo de Teste de Envio e Recebimento

O Módulo de Teste de Envio e Recebimento (MTER) é conectado com o periférico. Os sinais deste módulo são os mesmos mostrados na Tabela 2.

Nesse módulo, existem dois métodos: um que envia e outro que recebe pacotes. Esse método que recebe pacotes também imprime o que foi recebido flit a flit, até chegar ao fim do pacote. O funcionamento do método que envia pacotes é o mesmo mostrado na MEF da Figura 26. Já o funcionamento do método que recebe pacotes é mostrado no Código 4.2.

```
1 void MTER::receive_nnpacket2(){
2   if (reset.read() == 1){
```

```
3       if (rx == 1 && counter<13){
4         cout << "Dado do flit recebido: " << data_in.read() << endl;
5         counter++;
6         }
7       }
8   }
```

Código 4.2 – Método de Recebimento de Pacotes do MTER

O interfaceamento do MTER com o periférico é feito através do testbench desenvolvido, como será visto a seguir.

4.1.2.2 Testbench do Sistema Periférico-MTER

No testbench desenvolvido para este estudo de caso, foi feito um interfaceamento entre o MTER e o periférico. Uma representação da arquitetura presente nesse testbench é mostrado na Figura 35.

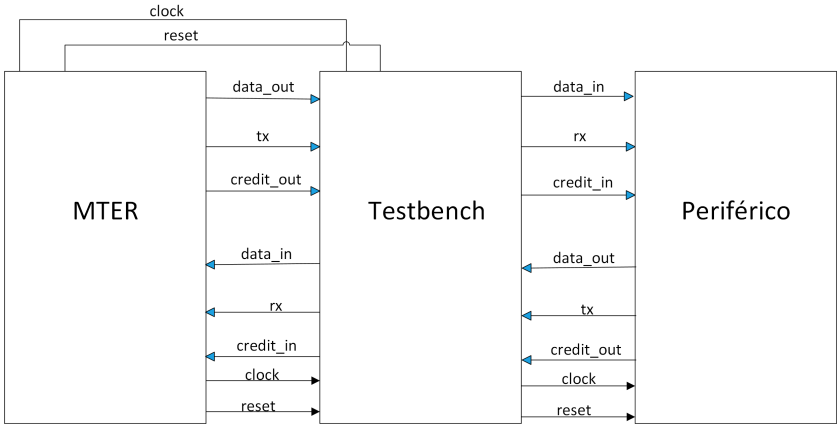

Figura 35 – Arquitetura do Testbench do Sistema Periférico-MTER

Nota-se que existe um ciclo de pacotes de envio e recebimento na conexão dos respectivos sinais data_in e data_out na troca de dados, rx e tx na validação do envio e recebimento de dados e credit_in e credit_out referente ao controlde de fluxo de dados. A conexão é feita através de sinais declarados no próprio testbench. Esses sinais fazem uma intermediação entre os sinais de entrada e saída dos módulos do periférico e do MTER. Isso pode ser visto no Código 4.3, que é um trecho do código do testbench.

```
1   SC_MODULE(tb_receive) {
2
3     sc_signal<bool >    clock;
4     sc_signal<bool >    reset;
5
6     sc_signal<bool >    tx_np_f;
```

```
7   sc_signal <bool >     tx_teste_send;
8   sc_signal<regflit >   data_out_teste_send;
9   sc_signal<regflit >   data_out_np_f;
10  sc_signal<bool >    credit_out_teste_send;
11  sc_signal<bool >    credit_out_np_f;
12
13  void ClockGenerator();
14  void resetGenerator();
15
16  neural_perif * TN;
17  MTER * TS;
18
19  SC_HAS_PROCESS(tb_receive);
20  tb_receive (sc_module_name name_) : sc_module(name_) {
21
22    TN = new np_f("np_f");
23    TN->clock(clock);
24    TN->reset(reset);
25    TN->rx(tx_teste_send);
26    TN->data_in(data_out_teste_send);
27    TN->credit_out(credit_out_np_f);
28    TN->tx(tx_np_f);
29    TN->data_out(data_out_np_f);
30    TN->credit_in(credit_out_teste_send);
31
32    TS = new teste_send("teste_send");
33    TS->clock(clock);
34    TS->reset(reset);
35    TS->rx(tx_np_f);
36    TS->data_in(data_out_np_f);
37    TS->credit_out(credit_out_teste_send);
38    TS->tx(tx_teste_send);
39    TS->data_out(data_out_teste_send);
40    TS->credit_in(credit_out_np_f);
41
42    SC_THREAD(ClockGenerator);
43    SC_THREAD(resetGenerator);
44  }
45 };
```

Código 4.3 – Módulo do Testbench do Periférico

Foi feita então a simulação ao enviar pacotes do MTER para o periférico de rede neural. O periférico recebe realiza o regimento desses pacotes, e imprime os dados destes. Por fim, obtém a saída e envia os dados de saída para o MTER, que imprime esses dados. As impressões são feitas através de mensagens de *debug*. Essas mensagens são mostradas na Figura 36.

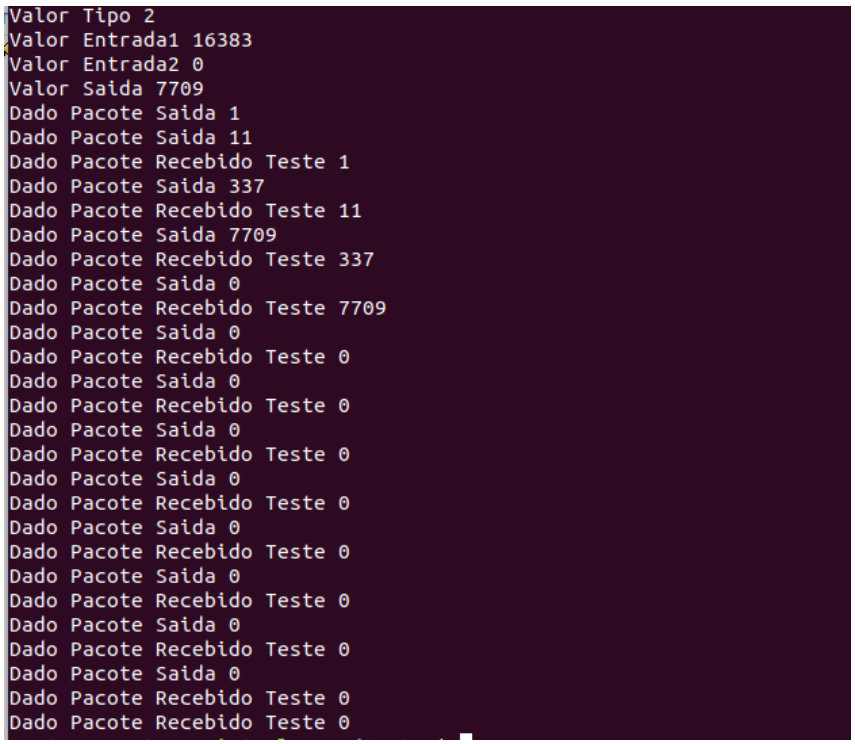

Figura 36 – Valores Impressos no Terminal Após o Tratamento no MTER e no Periférico

Os quatro primeiros valores consistem nos valores que o MTER envia ao periférico (tipo de função de ativação e entradas), seguidos da saída do Módulo de Rede Neural. Os dados rotulados "Dado de Saída"consistem nos dados que o periférico envia de volta ao MTER. Por fim, os dados rotulados "Dado Pacote Recebido Teste"marcam o recebimento dos dados por parte do MTER e consistem nos mesmos dados enviados pelo periférico.

4.2 Avaliação do Periférico Acoplado à Memphis

Esta seção visa avaliar a resposta do periférico de rede neural quando submetido a diferentes cenários de teste e aplicações. Estes cenários e aplicações são apresentados na Seção 4.2.1. Já as Seção 4.2.2 e a Seção 4.2.3 avaliam o periférico frente aos casos de teste apresentados anteriormente. Por fim, a Seção 4.2.5 valida o tráfego de pacotes na rede com o periférico acoplado.

4.2.1 Aplicações e Cenários Utilizados

Para as avaliações dos cenários de teste, foi utilizada como plataforma a Memphis com as seguintes configurações padronizadas: NoC malha 2D, algoritmo de roteamento XY, tamanho do flit de 32 bits, tamanho do pacote de 13 flits, modelo de descrição em SystemC e posição do PE gerente na posição [0,0], e tamanho de rede 3 por 3, seguindo o modelo da Figura 20. A Figura 42 da rede gerada pela Memphis está disponível no Anexo A.

Os cenários de teste foram determinados a partir da alteração dos arquivos my_scenario e my_testcase. Esses arquivos carregam informações que são utilizadas para gerar a Memphis. Para observar o comportamento do periférico frente a diferentes casos de teste, foram feitas algumas alterações nos arquivos supracitados. Dessa forma, foram alterados o número de tarefas por PE progressivamente, variou-se a posição onde as tarefas são alocadas na Memphis e adicionaram-se mais aplicações para serem executas paralelamente com o objetivo de sobrecarregar o sistema.

Para as avaliações dos casos de teste, foi implementada a aplicação prod_cons. Essa aplicação é composta por duas tarefas, denominadas prod (produtor) e cons (consumidor). Ela funciona de forma que a tarefa prod realiza inicialmente a chamada de sistema NNPeriph, solicitando a obtenção da saída da rede neural ao periférico. Para isso, são definidas as entradas e o tipo de função de ativação da rede neural em uma estrutura Message. Com isso, essas informações podem ser enviadas para a rede neural e assim obter a sua respectiva saída. A saída é armazenada em outra estrutura Message e enviada para a tarefa cons através da chamada de sistema Send. Por fim, a tarefa cons recebe a saída da rede neural através da chamada de sistema Receive. Os códigos dessas aplicações estão disponíveis no Apêndice (Código prod A.1 e Código cons A.2). A Figura 37 mostra uma representação gráfica do funcionamento dessa aplicação.

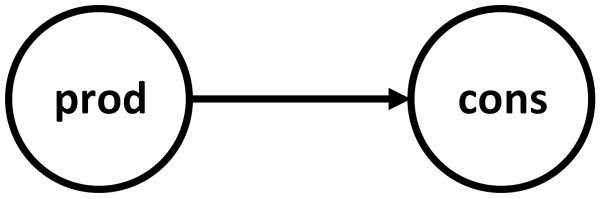

Figura 37 – Funcionamento da Aplicação prod_cons

Além disso, foi utilizada a aplicação chamada MPEG para sobrecarregar o sistema. Essa aplicação implementa um decodificar parcial MPEG. Nessa aplicação, existem 5 tarefas que recebem uma entrada e seguem um fluxo de pipeline até gerar uma saída. Esse funcionamento é mostrado na Figura 38.

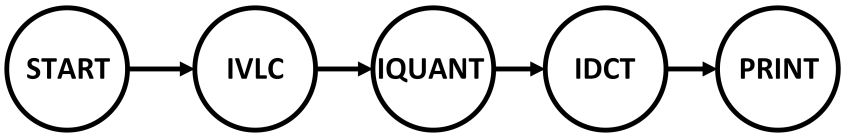

Figura 38 – Funcionamento da Aplicação MPEG

4.2.2 Avaliação do Tempo de Execução do Módulo do Periférico

Nesta seção, será avaliado o tempo de execução em ciclos de clock do periférico desde a chegada de um pacote de solicitação até o envio do pacote de resposta com saída da rede neural. Para isso, foi implementado um contador de clock no Módulo de Interface do periférico, o qual é iniciado assim que o primeiro flit do pacote de solicitação é recebido. A contagem de ciclos de clock é finalizada no momento que o último flit do pacote de resposta contendo a saída da rede neural é enviado. Ao finalizar a contagem, o tempo de execução do periférico, em número de ciclos de clock, é mostrado no terminal durante a execução do sistema, como mostrado na Figura 39.

```
my_scenario successfully generated!

        SystemC 2.3.3-Accellera --- Apr 22 2021 05:14:23
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Creating PE PE0x0
Creating PE PE1x0
Creating PE PE2x0
Creating PE PE0x1
gustavo@gustavo-VirtualBox:~/sandbox_memphis$ Creating PE PE1x1
Creating PE PE2x1
Creating PE PE0x2
Creating PE PE1x2
Creating PE PE2x2
App Injector requesting app prod_cons
Manager sent ACK
Loading task ID 0 to PE 0x2
Loading task ID 1 to PE 0x1
Ciclos de clock da tarefa 1 : 30
```

Figura 39 – Número de Ciclos Impresso no Terminal

Foram realizados vários cenários para avaliar o tempo de execução do periférico. Todos os cenários utilizam as mesmas configuração da plataforma Memphis descritas na Seção 4.2.1, executando a aplicação prod cons. A tarefa cons foi alocada sempre no PE de coordenadas [0,2]. Já a tarefa prod, que requisita o periférico, teve sua alocação variada de acordo com o cenário teste. A Tabela 3 apresenta todos os cenários, sendo que: a coluna à esquerda mostra a posição onde a tarefa prod foi alocada; ao passo que a coluna a direita mostra o tempo de execução do periférico em ciclos de clock.

Tabela 3 – Tempos de execução do módulo do periférico para diferentes posições de alocação da tarefa prod

Posição da Tarefa prod	Tempo de Execução do Periférico (Ciclos de Clock)
[0,1]	30
[1,0]	30
[2,0]	30
[1,1]	30
[1,2]	30
[2,1]	30
[2,2]	32

Nota-se que na Tabela 3 o tempo de execução do periférico permaneceu igual, com exceção do cenário onde a tarefa prod estava alocado na posição [2,2]. Isso pode ser explicado, pois nesse cenário a tarefa prod está alocada no mesmo PE onde está conectado o periférico. Uma possível explicação para isso seria causada pelo espaço de armazenamento em buffer reduzido, causando um tempo maior para o envio de pacote.

Também foi realizado um cenário em que sobrecarregou-se o sistema ao injetar a aplicação MPEG juntamente à aplicação prod_cons. A alocação foi feita de forma dinâmica na geração da Memphis, com a tarefa cons alocada no PE [0,2] e a tarefa prod alocada no PE [0,1]. Esse cenário resultou numa rede muito mais congestionada. Com isso, o tempo de execução do periférico aumentou de forma notória (107 ciclos). Isso ocorre por conta do maior congestionamento na rede do sistema, impedindo que seja finalizado o envio do pacote por parte do periférico.

4.2.3 Avaliação do Tempo Total de Execução do Sistema

Nesta seção, será avaliado o tempo de execução total do sistema, variando a posição de alocação de uma tarefa que requisita o periférico. É feita uma comparação do tempo de execução total do sistema com a implementação da RNA em *hardware*, através do periférico, e em *software*.

Foram realizados vários cenários para avaliar o tempo total de execução do sistema. Todos os cenários utilizam as mesmas configuração da plataforma Memphis descritas na Seção 4.2.1, executando a aplicação prod_cons. A tarefa cons foi alocada sempre no PE [0,2]. Já tarefa prod, que requisita o periférico, teve sua alocação variada de acordo com o cenário teste. A Tabela 4 apresenta todos os cenários, sendo que: a coluna à esquerda mostra a posição onde a tarefa prod foi alocada; já a coluna do meio mais à esquerda mostra o tempo total de execução do sistema, em ciclos de clock, quando é utilizada a implementação em *hardware*; a coluna do meio mais à direita mostra o tempo total de execução do sistema, em

ciclos de clock, quando é utilizada a implementação em *software*; e por fim, a coluna mais à direita mostra a redução percentual entre o tempo de execução em *software* comparado com o de *hardware*. O tempo total de execução é obtido através de um contador de ciclos de clock implementado no sistema.

Tabela 4 – Resultados do cenário de avaliação do tempo total de execução do sistema

Posição da Tarefa prod	**Tempo Total de Execução com RNA em *Hardware* (em ciclos de clock)**	**Tempo Total de Execução com RNA em *Software* (em ciclos de clock)**	**Redução do SW comparado ao HW (%)**
[0,1]	23732	22777	4,02
[1,0]	23498	22561	3,98
[2,0]	23546	22591	4,05
[1,1]	23658	22703	4,03
[1,2]	23656	22719	3,96
[2,1]	23688	22751	3,95
[2,2]	23686	22749	3,95

Alguns fatores que afetam o tempo de execução são a distância entre prod e cons, a distância entre prod e o PE Gerente e a distância entre prod e o periférico de rede neural. Essas distâncias aumentam o tempo de comunicação no sistema, aumentando o tempo de execução. Por exemplo, pode-se ver que a diferença no tempo de execução diminui quando prod se aproxima do periférico, mostrando, ainda que leve, a influência da distância do caminho dos dados até o periférico.

Nota-se também que o tempo de execução da implementação da RNA em *software* é sempre menor comparado a da implementação em *hardware* (em média 4%). Isso se deve, primeiramente, ao fato de a rede neural ser simples, não necessitando de muitos ciclos de clock para a execução em *software*, além da implementação em *hardware* necessitar do uso de chamadas de sistema, comunicação e tratamento de pacotes no sistema.

4.2.4 Avaliação do Tempo de *delay* de Requisição ao Periférico

Esta seção avalia o tempo de *delay* de requisição a um periférico por parte de uma tarefa da aplicação. É avaliado o tempo de *delay* tanto para a implementação da rede neural em *hardware*, através do periférico, como em *software*. No caso da implementação em *hardware*, considera-se o *delay* contado desde o tempo da chamada de sistema NNPeriph até o recebimento do pacote de resposta do periférico e consequentemente a obtenção da saída da rede neural. Já no caso da implementação em *software*, conta-se o tempo desde a chamada da função que implementa a RNA em *software* até a obtenção da sáida dessa rede. Na Se-

ção 4.2.4.1, foi avaliado somente o *delay* da implementação em *hardware* variando o número de tarefas alocadas em um mesmo PE. A Seção 4.2.4.2 compara o *delay* da implementação da RNA em *hardware* e *software*. A Seção 4.2.4.1 avalia apenas o *delay* da implementação em *hardware*, pois esta depende do envio e recebimento de pacotes pelo sistema, o qual é diretamente influenciado pela posição de alocação das tarefas. A implementação em *hardware* sofre alterações em *delay* devido a mudanças na alocação de tarefas. Em todas avaliações, foi utilizada a chamada de sistema GetTick() para calcular o *delay* em *hardware* e em *software*.

4.2.4.1 Avaliação do *Delay* com Aumento do Número de Tarefas Alocadas num mesmo PE

Neste cenário, foi testada a interferência no *delay* de requisição ao periférico com muitas tarefas alocadas num mesmo PE. Nesse caso, foram implementadas mais aplicações que são cópias da aplicação prod_cons. Com isso, tem-se várias tarefas prod e várias tarefas cons. As tarefas prod provenientes de diferentes aplicações então são alocadas progressivamente no PE [0,1]. Já as cons são alocadas no PE [0.2]. Dessa forma, pode-se avaliar como ocorre a interferência de diferentes tarefas provenientes de aplicações diferentes. A Tabela 5 apresenta os resultados para os diferentes cenários apresentados, com a coluna à esquerda apresentando o número de tarefas prod no PE [0,1] e a coluna à direita apresentando o *delay* de requisição associado.

Tabela 5 – Tempos de *delay* de *hardware* com várias tarefas no PE [0,1]

Número de Tarefas	***Delay* de Hardware**
1	1246
2	1442
3	1604
4	1733
5	1866
6	1999

Nota-se que o *delay* aumenta progressivamenter nesse caso ao aumentar o número de tarefas. Isso ocorre devido ao congestionamento resultante de mais solicitações ao periférico.

Além disso, foi avaliado o tempo de execução em *hardware* quando alocam-se as tarefas provenientes de uma mesma aplicação num mesmo PE. Isso foi feito ao inserir uma chamada de sistema NNPeriph na tarefa cons e alocá-la no mesmo PE que a tarefa prod. Este teste foi feito em diferentes PEs para avaliar se o caso era particular de um PE ou não. Com isso, alocaram-se a tarefa prod e a tarefa cons modificada no PE [0,1] e depois no PE [1,1]. Os resultados desse novo cenário são mostrados na Tabela 6.

Tabela 6 – Tempos de *delay* de *hardware* com tarefas provenientes de aplicações diferentes acopladas num mesmo PE

PE	***Delay* de *Hardware* Tarefa 1**	***Delay* de *Hardware* Tarefa 2**
[0,1]	2968	7213
[1,1]	3013	7528

Nota-se um aumento expressivo do tempo de execução em *hardware*. Isso ocorre por conta do escalonamento das tarefas, que faz com que as tarefas fiquem cada vez mais tempo bloqueadas conforme aumenta-se o número de tarefas alocadas num mesmo PE quando provenientes de uma mesma aplicação. Consequentemente, o tempo de obtenção da saída em *hardware* é também maior neste cenário do que no cenário anterior a este.

4.2.4.2 Comparação da Obtenção da Resposta da RNA em *Hardware* com *Software*

O tempo de obtenção da saída da rede neural em *software* é sempre de 343 ciclos de clock e em *hardware* é sempre 1246 ciclos de clock, em cenário normal. Esses tempos são calculados através da primitiva GetTick da API do sistema, que retorna o valor do número de ciclos passados até o momento da chamada. Como pode ser visto, o tempo em *software* é menor. O tempo de *hardware* é maior por conta dos motivos listados abaixo, independente do cenário.

- Tempo resultante da chamada de sistema NNPeriph da API;
- Tempo resultante do fato de chamada de sistema enviar um pacote de solicitação para o periférico, que percorre toda a rede até chegar ao periférico;
- Tempo de o periférico desempacotar o pacote, obter a resposta da rede neural e montar o pacote de saída
- Tempo de o pacote de saída percorrer toda a rede até chegar ao PE onde está a locada a tarefa que requisitante;
- Tempo de tratamento do pacote por parte do kernel do PE, que em seguida retorna o valor de saída da rede neural e o envia de volta para a tarefa;

Isto posto, pode-se afirmar que o tempo de execução em *hardware* é maior que em *software* nesse caso por se tratar de uma rede neural simples. Num caso em que a rede neural fosse mais robusta, com realimentação, por exemplo, o tempo de execução em *hardware* geralmente seria menor que o tempo em *software*.

4.2.5 Validação do Tráfego de Pacotes

Por fim, pode-se validar o tráfego de pacotes entre a rede e o periférico. Essa validação foi feita ao alocar a PE [0,1] e a tarefa cons ao PE [0,2]. A Memphis provê uma representação gráfica para o tráfego de flits na rede. Com isso, obtém-se essa representação para o serviço NN_REQUEST (Figura 40) e para o serviço NN_SEND (Figura 41).

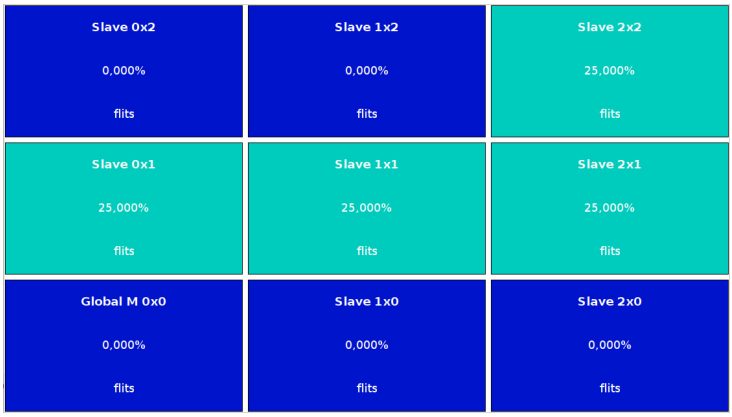

Figura 40 – Trafego de Flits no Serviço NN_REQUEST

Figura 41 – Trafego de Flits no Serviço NN_SEND

Isto posto, pode-se ver que o pacote chega ao periférico de rede neural através do serviço NN_REQUEST e retorna ao PE da tarefa através do serviço NN_SEND seguindo o algoritmo de roteamento XY, conforme esperado.

5 Conclusões

Esse trabalho realizou a proposta da implementação de um periférico de rede neural para ser integrado na plataforma Memphis. Para isso, foi realizado um estudo de caso desenvolvendo um periférico com uma rede neural capaz de obter a saída de uma porta XOR em sua estrutura. Essa rede neural é proveniente de trabalhos de cooperação do Departamento de Ciência da Computação da Universidade de Brasília e foi fornecida na linguagem de descrição de *hardware* Verilog. O periférico foi desenvolvido através da implementação de dois módulos projetados: o Módulo de Rede Neural e o Módulo de Interfaceamento. O Módulo de Rede Neural provê a saída da rede neural supracitada. Dessa forma, a RNA fornecida foi traduzida para a linguagem de descrição de *hardware* SystemC com o objetivo de buscar um menor tempo de simulação e uma melhor posterior transição do código de *hardware* para *software* para sua integração ao módulo em questão. Já o módulo de interfaceamento tem como objetivo envia e recebe pacotes do sistema. O periférico de rede neural teve seu funcionamento adequado à Memphis ao realizar algumas adições no código fonte da plataforma. Foram implementados e adicionados a nível de *software* então uma chamada de sistema para realizar requisições ao periférico, uma função que tem como objetivo gerar e enviar o pacote de recebimento do periférico, um serviço para realizar o tratamento de pacotes provenientes do periférico e um módulo de comunicações para auxiliar no funcionamento dos elementos descritos. Já a nível de *hardware* foram realizadas as conexões entre o periférico e roteadores de PEs da rede e o Gerenciador de Propósito Geral. Por fim, foi desenvolvida uma função em linguagem C que emula o funcionamento da RNA em questão para fins de comparação.

Esse periférico se mostrou funcional tanto isolado quanto acoplado à Memphis. Foram feitos inicialmente testes para o funcionamento da rede neural traduzida em SystemC, que se mostrou válida. Após acoplar a RNA traduzida ao Módulo de Rede Neural e desenvolver o Módulo de Interfaceamento, o periférico foi testado individualmente antes de ser incluído na estrutura da Memphis. Para isso, foi desenvolvido um módulo que emula de forma simples o envio e recebimento de pacotes, mostrando resultados válidos.

Foram feitos diversos casos de teste para avaliar o desempenho e funcionamento do periférico acoplado à Memphis. Para isso, foi utilizada a aplicação prod_cons numa rede 3 por 3 configurada configurada com algoritmo de roteamento XY, tamanho do flit de 32 bits, tamanho do pacote de 13 flits, modelo de descrição em SystemC e posição do PE gerente na posição [0,0]. Foram feitos então testes em diferentes cenários para avaliar o tempo de execução total do módulo do periférico, avaliar o tempo de execução total do sistema, avaliar os tempos de delay de requisição ao periférico, comparar com a obtenção em *hardware* da saída da rede neural com isto em sofware e por fim validar o tráfego de pacotes.

Com isso, o tempo de execução total do módulo do periférico foi avaliado ao variar a posição em que a tarefa prod é alocada, mostrando tempos de execução satisfatórios. O tempo de execução total do Sistema foi avaliado da mesma forma e mostrou que o desempenho do sistema não é muito influenciado pela distância entre o periférico e a tarefa requisitante. Já o delay de requisição ao periférico foi feito no cenário em que aumenta-se o número de tarefas requisitantes num mesmo PE. Foi visto que quando as tarefas são provenientes de aplicações diferentes, o delay aumenta por conta do congestionamento causado por um aumento de solicitações ao periférico. No caso em que as tarefas são provenientes de uma mesma aplicação, o delay aumenta substancialmente por conta do escalonamento entre tarefas.

Foi feita então a comparação entre o tempo de obtenção da saída da RNA em *hardware* com o mesmo tempo em *software*. Foi constatado que o tempo em *hardware* foi maior devido a diversos fatores como, por exemplo, o tempo de tratamento do pacote por conta do kernel do PE no qual é alocada a tarefa.

Então, foi validado o tráfego de flits na comunicação entre a rede e o periférico através de uma representação gráfica fornecida pela Memphis.

5.1 Conclusão Final

Primeiramente, é importante comentar que é possível utilizar o Módulo de Interface implementado para outros módulos de *hardware*, bastando somente realizar as devidas conexões nos portmaps.

Em segundo lugar, frente aos resultados obtidos, foi constatado que neste caso específico, em que se trata de uma rede neural simples, apesar de ser uma implementação em *hardware*, o tempo de execução em *software* foi menor. Isso ocorre porque o tempo de execução total em *hardware* engloba diversos fatores, como o tempo da chamada de sistema; a comunicação no sistema, incluindo o tempo de troca e tratamento de pacotes; e o tempo de escalonamento entre tarefas. Para exemplificar, o tempo de execução do periférico isoladamente é de somente 30 ciclos de clock para uma rede descongestionada. Para redes mais congestionadas, o periférico pode levar mais tempo devido ao recebimento e envio de pacotes.

Por outro lado, a implementação em *software* obteve um tempo de execução de 343 ciclos de clock e não sofre interferência de outros fatores. Porém, caso a rede neural tenha uma computação mais extensiva, pode-se chegar a um ponto em que o tempo de obtenção da saída em *software* supere a soma dos tempos citados anteriormente que aumentam o tempo de obtenção da resposta em *hardware*, como, por exemplo, o tempo resultante da chamada de sistema NNPeriph da API. Dessa forma, esses tempos tendem a ser insignificantes conforme o aumento da computação da rede neural. Ou seja, a relação entre a obtenção em *software* e

em *hardware* depende da complexidade da RNA.

5.2 Trabalhos Futuros

Como visto anteriormente, o potencial de um periférico de rede neural é mais explorado conforme aumenta-se a complexidade da rede. Sabendo disso, sugere-se para trabalho futuro utilizar redes neurais mais complexas. Além disso, é possível realizar a reconfiguração da rede neural durante a sua execução, como por exemplo, alterar seus pesos durante a execução. É interessante integrar outra rede neural para observar comno o sistema reage. Para explorar mais o comportamento do periférico, também é importante dar continuidade a este trabalho ao realizar testes com outras aplicações.

Referências

ALEXANDER, D. **Neural Networks: History and Applications**. Nova Science Pub Inc, abr. 2020. ISBN 978-1-53617188-4. Citado nas pp. 17, 36, 38.

ARDA, S. E.; NK, A.; GOKSOY, A. A.; MACK, J.; KUMBHARE, N.; SARTOR, A. L.; AKOGLU, A.; MARCULESCU, R.; OGRAS, U. Y. A simulation framework for domain-specific system-on-chips: work-in-progress. In: CODES/ISSS '19: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS COMPANION. New York, NY, USA: Association for Computing Machinery, out. 2019. P. 1–2. ISBN 978-1-45036923-7. DOI: 10.1145/3349567.3351719. Citado na p. 17.

ATIENZA, D.; ANGIOLINI, F.; MURALI, S.; PULLINI, A.; BENINI, L.; DE MICHELI, G. Network-on-Chip design and synthesis outlook. **Integration**, Elsevier, v. 41, n. 3, p. 340–359, mai. 2008. ISSN 0167-9260. DOI: 10.1016/j.vlsi.2007.12.002. Citado na p. 21.

B. ABDALLAH, A. **Advanced Multicore Systems-on-Chip Architecture, On-Chip Network, Design-Springer**. 1. ed.: Springer Nature Singapore Pte Ltd, 2017. Citado nas pp. 23–26.

BANERJEE, A.; SUR, B. **SystemC and SystemC-AMS in Practice**. Cham, Switzerland: Springer International Publishing, 2014. ISBN 978-3-319-01146-2. DOI: 10.1007/978-3-319-01147-9. Citado na p. 35.

CAETANO, M. F.; MAKIUCHI, M. R.; FERNANDES, S. S.; LAMAR, M. V.; BORDIM, J. L.; BARRETO, P. S. A Recurrent Neural Network MAC Protocol Towards to Opportunistic Communication in Wireless Networks. In: 2019 16TH INTERNATIONAL SYMPOSIUM ON WIRELESS COMMUNICATION SYSTEMS (ISWCS). IEEE, ago. 2019. P. 63–68. DOI: 10.1109/ISWCS.2019.8877272. Citado na p. 18.

CARARA, E. A. **Serviços de comunicação diferenciados em sistemas multiprocessados em chip baseados em redes intra-chip**. 2011. Tese (Doutorado) – Pontifícia Universidade Católica do Rio Grande do Sul. Disponível em: <https://repositorio.pucrs.br/dspace/handle/10923/1593>. Citado na p. 28.

CHEN, Y.-J.; YANG, C.-L.; CHANG, Y.-S. An architectural co-synthesis algorithm for energy-aware Network-on-Chip design. **J. Syst. Archit.**, North-Holland, v. 55, n. 5, p. 299–309, mai. 2009. ISSN 1383-7621. DOI: 10.1016/j.sysarc.2009.02.002. Citado na p. 21.

DEMUTH, H. B.; BEALE, M. H.; DE JESS, O.; HAGAN, M. T. **Neural Network Design**. Martin Hagan, set. 2014. DOI: 10.5555/2721661. Citado nas pp. 37, 38.

EMBECOSM. **2.1. What is SystemC**. Fev. 2018. [Online; accessed 25. 05 2021]. Disponível em: <https://www.embecosm.com/appnotes/ean1/html/ch02s01.html>. Citado na p. 35.

HOJABR, R.; MODARRESSI, M.; DANESHTALAB, M.; YASOUBI, A.; KHONSARI, A. Customizing Clos Network-on-Chip for Neural Networks. **IEEE Trans. Comput.**, IEEE, v. 66, n. 11, p. 1865–1877, jun. 2017. ISSN 1557-9956. DOI: 10.1109/TC.2017.2715158. Citado na p. 23.

LI, Y.; LOURI, A. ALPHA: A Learning-Enabled High-Performance Network-on-Chip Router Design for Heterogeneous Manycore Architectures. **IEEE Trans. Sustainable Comput.**, IEEE, p. 1, mar. 2020. ISSN 2377-3782. DOI: 10.1109/TSUSC.2020.2981340. Citado nas pp. 17, 20.

MAKIUCHI, M. R. **Desenvolvimento de rede neural artificial recorrente em FPGA para previsão online de oportunidades em transmissões oportunísticas em redes de comunicação wireless**. Dez. 2018. 81 f., il. Trabalho de Conclusão de Curso (Bacharelado em Engenharia Mecatrônica). Disponível em: <https://bdm.unb.br/handle/10483/22068>. Citado nas pp. 36, 40, 41, 43.

MARINHO, A. B. **HeMPS-V : um MPSoC com processador de arquitetura RISC-V**. Ago. 2018. 135 f. Trabalho de Conclusão de Curso (Bacharelado em Engenharia da Computação). Disponível em: <https://bdm.unb.br/handle/10483/25346>. Citado na p. 21.

PAGANI, S.; CHEN, J.-J.; SHAFIQUE, M.; HENKEL, J. **Advanced Techniques for Power, Energy, and Thermal Management for Clustered Manycores**. Cham, Switzerland: Springer International Publishing. ISBN 978-3-319-77479-4. Disponível em: <https://link.springer.com/book/10.1007/978-3-319-77479-4>. Citado na p. 17.

RUARO, M. **Self-adaptive QOS at communication and computation levels for many-core system-on-chip**. 2018. Tese (Doutorado) – Pontifícia Universidade Católica do Rio Grande do Sul. Disponível em: <https://repositorio.pucrs.br/dspace/handle/10923/11785>. Citado nas pp. 29–32, 34.

RUARO, M.; CAIMI, L. L.; FOCHI, V.; MORAES, F. G. Memphis: a framework for heterogeneous many-core SoCs generation and validation. **Des. Autom. Embed. Syst.**, Springer US, v. 23, n. 3, p. 103–122, dez. 2019. ISSN 1572-8080. DOI: 10.1007/s10617-019-09223-4. Citado nas pp. 17, 20, 27, 33.

SEPÚLVEDA FLÓREZ, M. J. Projeto de estruturas de comunicação intrachip baseadas em NoC que implementam serviços de QoS e segurança. **Biblioteca Digital de Teses e Dissertações da Universidade de São Paulo**, Biblioteca Digital de Teses e

Dissertações da Universidade de São Paulo, jul. 2011. DOI: 10.11606/T.3.2011.tde-04112011-140055. Citado na p. 21.

TOFIS, C.; THEOCHARIDES, T.; MICHAEL, M. K. FPGA-Based Laboratory Assignments for NoC-Based Manycore Systems. **IEEE Trans. Educ.**, IEEE, v. 55, n. 2, p. 180–189, jul. 2011. ISSN 1557-9638. DOI: 10.1109/TE.2011.2159795. Citado nas pp. 17, 20.

VENKATARAMANI, V.; CHAN, M. C.; MITRA, T. Scratchpad-Memory Management for Multi-Threaded Applications on Many-Core Architectures. **ACM Trans. Embedded Comput. Syst.**, Association for Computing Machinery, v. 18, n. 1, p. 1–28, fev. 2019. ISSN 1539-9087. DOI: 10.1145/3301308. Citado na p. 17.

WÄCHTER, E. W.; BIAZI, A.; MORAES, F. G. HeMPS-S: A homogeneous NoC-based MPSoCs framework prototyped in FPGAs. In: 6TH INTERNATIONAL WORKSHOP ON RECONFIGURABLE COMMUNICATION-CENTRIC SYSTEMS-ON-CHIP (RECOSOC). IEEE, jun. 2011. P. 1–8. DOI: 10.1109/ReCoSoC.2011.5981498. Citado na p. 20.

WEICHSLGARTNER, A.; WILDERMANN, S.; GLASS, M.; TEICH, J. **Invasive Computing for Mapping Parallel Programs to Many-Core Architectures**. Singapore: Springer Nature. ISBN 978-981-10-7356-4. Disponível em: <https://link.springer.com/book/10.1007/978-981-10-7356-4>. Citado na p. 17.

ZEFERINO, C. A. **Redes-em-Chip : arquiteturas e modelos para avaliação de área e desempenho**. 2003. [Online; accessed 25. 05 2021]. Disponível em: <https://lume.ufrgs.br/handle/10183/4179>. Citado nas pp. 22, 25, 26.

ZHANG, L.; HAN, Y.; XU, Q.; LI, X. w.; LI, H. On Topology Reconfiguration for Defect-Tolerant NoC-Based Homogeneous Manycore Systems. **IEEE Trans. Very Large Scale Integr. VLSI Syst.**, IEEE, v. 17, n. 9, p. 1173–1186, jun. 2009. ISSN 1557-9999. DOI: 10.1109/TVLSI.2008.2002108. Citado na p. 21.

Apêndices

APÊNDICE A – Códigos das Aplicações prod e cons

```
#include <api.h>
#include <stdlib.h>
#include "prod_cons_std.h"
#include "../../include/memphis_pkg.h"
Message msg, out_msg;
int main()
{

  int i;
  unsigned int delay_hard;


  Echo("Inicio da aplicacao prod");

  msg.msg[0] = 0x0000;
  msg.msg[1] = 0x3FFF;
  msg.msg[2] = 2;
  delay_hard = GetTick();
  NNPeriph(&msg, &out_msg);
  delay_hard = GetTick() - delay_hard;
  Echo("Delay de Hardware:");
  Echo(itoa(delay_hard));

  Send(&out_msg, cons);

  Echo("Fim da aplicacao prod");
  exit();

}
```

Código A.1 – Código em C da Aplicação prod

```
#include <api.h>
#include <stdlib.h>
#include "prod_cons_std.h"

```

```
5  Message out_msg;
6
7  int main()
8  {
9      Receive(&out_msg, prod);
10
11     Echo("Fim da aplicacao cons");
12   exit();
13 }
```

Código A.2 – Código em C da Aplicação cons

Anexos

ANEXO A – Configuração Padrão dos Cenários Avaliados

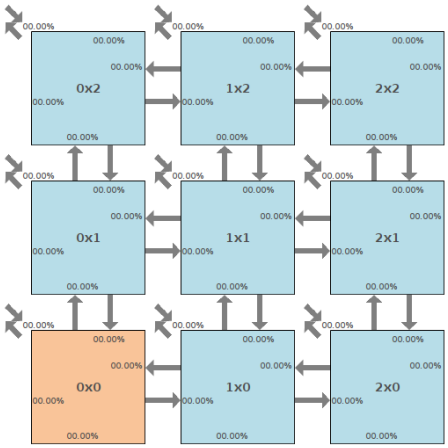

Figura 42 – Rede Gerada pela Memphis Utilizada

ANEXO B – QRCode para Acesso ao Repositório do Trabalho no GitHub

Figura 43 – QRCode para acesso ao Repositório do Trabalho no GitHub