final - julho/2020

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA
BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

INTERFACES DE COMUNICAÇÃO PADRONIZADAS PARA PERIFÉRICOS EM SISTEMAS MANY-CORE

DANIEL CORREA MUNHOZ

Trabalho de Conclusão apresentado como requisito parcial à obtenção do grau de Bacharel em Engenharia de Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Doutor Fernando Gehm Moraes
Co-Orientador: Prof. Doutor Marcelo Ruaro

Porto Alegre
2020

INTERFACES DE COMUNICAÇÃO PADRONIZADAS PARA PERIFÉRICOS EM SISTEMAS MANY-CORE

RESUMO

Os sistemas Many-core comunicam-se com periféricos ou componentes que podem não utilizar o mesmo protocolo de comunicação utilizado internamente ao Many-core. Uma forma de mitigar este problema é o desenvolvimento de uma interface de comunicação que desacopla os protocolos de comunicação do Many-core, dos protocolos dos periféricos. A adoção de protocolos padronizados permite estabelecer conexões com núcleos de propriedade intelectual de terceiros. O objetivo geral deste trabalho de conclusão de curso (TCC) é desenvolver a infraestrutura de software e hardware para que um sistema Many-Core que utiliza NoC como meio de interconexão possa se comunicar com periféricos, que utilizam interfaces padronizadas. Especificamente, foi desenvolvido um suporte de comunicação entre o Many-core e periféricos de terceiros que implementam o protocolo de comunicação *Wishbone*. Este TCC tem por ênfase a parte de desenvolvimento do software, envolvendo a criação de novas funções no sistema operacional, que fornecem uma interface de acesso abstraída entre as aplicações do usuário e o periférico *Wishbone*. O hardware da interface de comunicação é desenvolvido em System-C, permitindo a validação dos eventos do protocolo padrão adotado.

Palavras-Chave: Interface de Comunicação, Sistemas Many-core.

PERIPHERAL'S STANDARD COMMUNICATION INTERFACES FOR MANY-CORE SYSTEMS

ABSTRACT

Many-core systems communicate with peripherals or components that may not use the same communication protocol used by the Many-core. One way to mitigate this problem is the development of a communication interface that decouples the Many-core communication protocol with the peripherals' protocols. The adoption of standardized protocols allows establishing connections with third parties' intellectual property cores. The overall goal of this end-of-term work (TCC) is to develop the software and hardware infrastructure so that an NoC-based Many-Core system can communicate with peripherals, which use standardized interfaces. Specifically, the objective was to develop the communication support between many-core and third-party peripherals that implement the *Wishbone* communication protocol. This TCC focus on the software development part, involving the creation of new functions in the operating system, which allow an abstracted access interface between the user's applications and the *Wishbone* peripheral. The communication interface hardware is developed in System-C, allowing the validation of the events of the standard protocol adopted.

Keywords: Network Interface, Many-core Systems.

LISTA DE FIGURAS

LISTA DE SIGLAS

API – Application Programming Interface

HEMPS – Hermes Multiprocessor System on Chip

IP – Intelectual Property

MEMPHIS – Many-core Modeling Platform for Heterogenous SoCs

MPSOC – Multiprocessor System-on-Chip

NI – Network Interface

NOC – Network-on-Chip

PE – Processing Element

SOC – System-on-Chip

SUMÁRIO

1. INTRODUÇÃO

Seguindo a evolução da tecnologia, de acordo com a previsão da Lei de Moore [6], a redução do tamanho dos transistores e a densidade destes na mesma área de silício, resultou em projetos de SoCs (*Systems-On-Chip*) com mais de um processador. A revolução gerada pela chegada dos MPSoCs (*Multi-processor system-on-chip*) é abordada como a terceira revolução habilitada pela da Lei de Moore [4].

No contexto deste trabalho, um SoC com diversos núcleos de propriedade intelectual (3PIP - Third Party IPs) ou um MPSoC com diversos processadores, são denominados genericamente por *Many-cores*. Assume-se NoCs (*Networks-on-Chip*) como meio de interconexão entre os processadores. Adota-se NoCs devido à escalabilidade no que se refere à largura de banda e paralelismo na comunicação.

Um sistema Many-core pode comunicar-se com periféricos ou componentes externos que não utilizam o mesmo protocolo de comunicação que as mensagens internas do sistema. Uma interface de comunicação padronizada permite realizar estas conexões, sem expor os protocolos internos do Many-core. Uma das faces da interface troca mensagens utilizando o protocolo do Many-core, enquanto que sua outra face utiliza um determinado protocolo padronizado. Este trabalho apresenta uma interface de comunicação que converte o protocolo de comunicação próprio do Many-core Memphis [9] no protocolo Wishbone [8].

O controle na utilização dos recursos compartilhados é importante em sistemas Many-core para evitar problemas de concorrência e monopolização na comunicação com os periféricos. A interface de comunicação atua no controle de acesso aos periféricos. Impedir a comunicação de determinados elementos de processamentos (PEs) quando o periférico está comunicando-se com um determinado PE é fundamental para garantir a integridade dos eventos de comunicação.

1.1 Motivação

Sistemas embarcados integrados frequentemente exigem conexões com núcleos de terceiros. No entanto, a infraestrutura e os protocolos de comunicação podem não ser compatíveis, fazendo-se necessário o uso de lógica de adaptação. Para que um Many-core consiga se comunicar de forma a não expor seus protocolos internos, pode-se adotar o uso de uma interface capaz de traduzi-los em protocolos padronizados. Esta interface, além de prover um conjunto de serviços padronizados, fica responsável por estabelecer a comunicação entre o periférico e um PE.

O desenvolvimento de serviços de comunicação, tanto no nível de software quanto de hardware, exige conhecimentos adquiridos durante todo o período de graduação, por exemplo: Sistemas Operacionais, Redes de Comunicação, Projeto de Sistemas Integrados, Sistemas Embarcados. Além de gerar oportunidades de aperfeiçoamento nestas áreas.

1.2 Objetivos

O objetivo geral deste trabalho é desenvolver a infra-estrutura para que um sistema Many-core que utiliza NoC como meio de interconexão possa se comunicar com periféricos de terceiros que utilizam uma interface de comunicação padronizada.

Os objetivos específicos, destacados na Figura 1, incluem:

1. Dominar o sistema operacional do Many-core Memphis [9];
2. Dominar o desenvolvimento de *drivers* para a comunicação com periféricos - estes *drivers* são serviços que permitem que PEs consigam se comunicar com periféricos;
3. Validar a troca de mensagens entre os PEs e um periférico através da interface de comunicação;

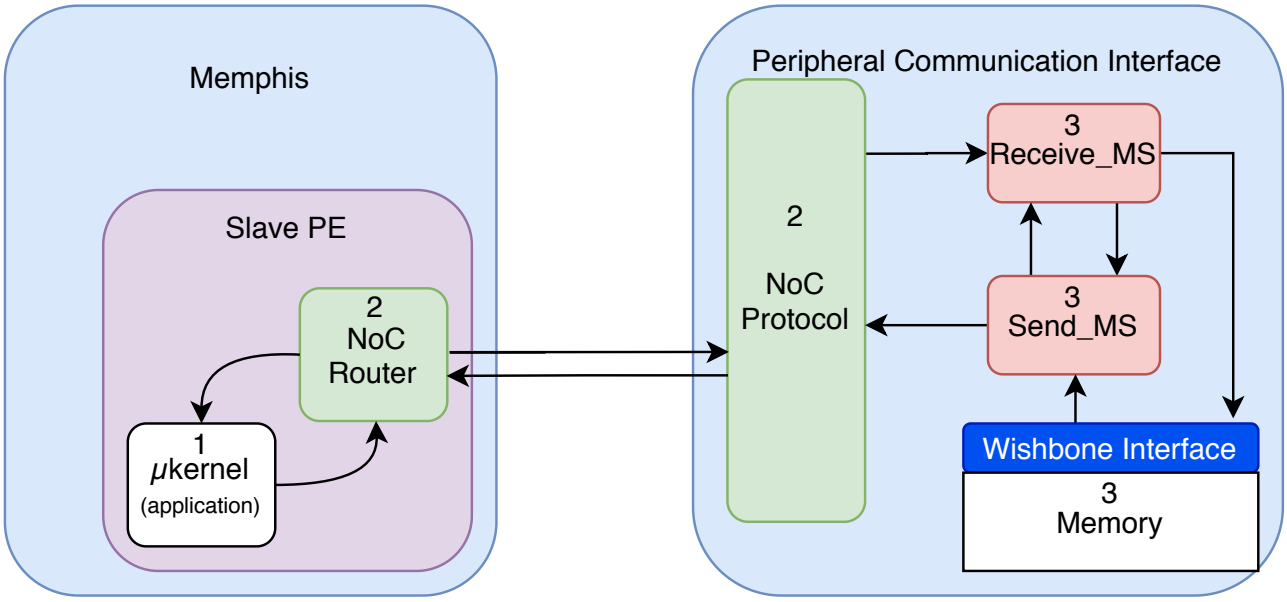

Figura 1 – Visão geral do desenvolvimento deste projeto.

1.3 Trabalho Proposto

A proposta deste trabalho é desenvolver um conjunto de serviços para o μkernel (Seção 2.2.3) dos PEs da Memphis [9]. Estes serviços habilitarão a comunicação do Many-core com periféricos de terceiros que implementem um protocolo de comunicação padronizado através de uma interface.

Através desta interface, uma tarefa de aplicação, executando em um determinado PE, pode realizar solicitações de serviços para um periférico. Ela desempenha um papel fundamental, pois é a interface de comunicação que conversa com os dois mundos, portanto, os serviços desenvolvidos para o μkernel devem ser capazes de atuar em conjunto com a interface, permitindo que as aplicações aloquem e façam uso de periféricos de terceiros através da mesma.

A interface de comunicação também deve adicionar mecanismos de controle de acesso para a alocação de periféricos. Ela permite que PEs façam uso exclusivo dos recursos alocados nesta, protegendo a comunicação através do descarte de pacotes oriundos de outros PEs. Os serviços desenvolvidos para o μkernel também devem ser capazes de atuar em conjunto com a interface, permitindo que as aplicações aloquem e façam uso de periféricos de terceiros através da mesma.

1.4 Estrutura do Documento

O conteúdo do presente trabalho está distribuído ao longo dos próximos cinco capítulos. A fundamentação teórica, Capítulo 2, descreve os estudos realizados para o desenvolvimento deste trabalho. O Capítulo 3 descreve a visão geral do projeto. O Capítulo 4 apresenta o conjunto de serviços que são utilizados para estabelecer a conexão com os periféricos. O Capítulo 5 explica o funcionamento da interface de comunicação desenvolvida. O Capítulo 6 sumariza o trabalho desenvolvido, comparando a proposto original frente ao desenvolvimento realizado, resumindo os temas explorados no desenvolvimento e trabalhos futuros para aprimorar o projeto.

2. FUNDAMENTAÇÃO TEÓRICA

Este Capítulo é organizado em três seções, necessárias para o desenvolvimento deste TCC. A Seção 2.1 apresenta o conceito de sistemas Many-cores. A Seção 2.2 apresenta o sistema Many-core Memphis, plataforma adotada no presente TCC. A Seção 2.3 apresenta conceitos relacionados a Wishbone, protocolo de comunicação utilizado pela interface desenvolvida.

2.1 Sistemas Many-core

Um Many-core é composto por vários núcleos de Propriedade Intelectual (Intelectual Property) (IP), cujo objetivo é executar diversas aplicações paralelamente. No nível de processamento, os Many-cores são classificados em três grandes grupos:

- Heterogêneos ou assimétricos: utilizam diferentes arquiteturas de elementos de processamento (PEs - do inglês *Processing Elements*), como GPPs, GPUs, DSPs e ASIPs.
- Homogêneos simétricos: todos os PEs têm a mesma arquitetura e organização e são replicados no sistema permitindo o suporte de processamento paralelo multi-tarefa.
- Homogêneos assimétricos: todos os PEs têm a mesma arquitetura, porém organização diferente, como, por exemplo, a arquitetura big.LITLLE [3].

Many-cores heterogêneos visam aplicações específicas, pois os seus IPs são especializados de acordo com a aplicação. Por outro lado, Many-cores homogêneos visam aplicações de uso geral, como aplicações multimídia e de telecomunicações, e utilizam a replicação de componentes idênticos (PEs) para aumentar a capacidade de processamento. Many-cores homogêneos são uma tendência, devido à possibilidade de alcançar um melhor equilíbrio de carga de processamento e tolerância a falhas devido à redundância de componentes.

NoC [1] é a infra-estrutura de comunicação atualmente mais adotada em Many-cores de alta complexidade para interligar os IPs, dissociando a computação da comunicação. NoCs são compostas por roteadores e enlaces. Em NoCs os roteadores têm a função de definir o caminho para cada pacote trocado entre um determinado par origem-destino, e os enlaces têm a função de interligar os roteadores e de ligar cada roteador com o seu respectivo IP. A NoC adotada na Memphis [9] é a Hermes [7].

As principais características de uma NoC são topologia, algoritmo de encaminhamento, controle de fluxo e modo de chaveamento [7]. Quanto à topologia, destaca-se a

topologia 2D-mesh, devido à sua facilidade de implementação e escalabilidade. Quanto ao algoritmo de roteamento, os algoritmos baseados em curvas (*turn-model*: XY, West-first, etc.) são utilizados principalmente, devido aos baixos requisitos de área, caminho mínimo e ausência de *deadlock*. Exemplo de método de controle de fluxo é o baseado em créditos, que proporciona uma menor latência em comparação com o método *handshake*. O modo de chaveamento mais utilizado é o modo de chaveamento de pacotes do tipo *wormhole*, uma vez que requer a menor quantidade de espaço em buffers e evita a reserva de recursos da rede durante um grande período de tempo como chaveamento de circuitos.

2.2 MEMPHIS

Memphis [9] é um Many-core desenvolvido no laboratório GAPH (Grupo de Apoio ao Projeto de Hardware). A arquitetura MEMPHIS possui uma estrutura de processamento homogênea e simétrica, um periférico de injeção de aplicações, e suporte à conexão de periféricos nas bordas externas da NoC.

A Figura 2(a) mostra uma visão geral da arquitetura, com a região de processamento de propósito geral (GPPC - *General Purpose Processing Cores*), e periféricos conectados na extremidade sul. A estrutura dos PEs na GPPC contém (Figura 2(b)):

- CPU: processador plasma (arquitetura MIPS 32-bits);
- Memória local (do tipo *Scratchpad* [11]), para armazenamento de código e instruções;
- DMNI: um componente que realiza as funções de DMA (*Direct Memory Access*), e NI (*Network Interface*) [10]. Ele permite que o roteador da NoC acesse a memória interna, sem intervenção do processador;
- Roteador (PS): roteador da rede Hermes [7].

2.2.1 Modelo de Aplicação de Usuário

As aplicações de usuário na Memphis são compostas por tarefas comunicantes. As aplicações são carregadas no sistema dinamicamente, ou seja, não há conhecimento prévio das aplicações que irão executar. Cada tarefa de uma dada aplicação possui um ID único que é gerado de forma crescente conforme novas aplicações (consequentemente, tarefas), são carregadas no sistema.

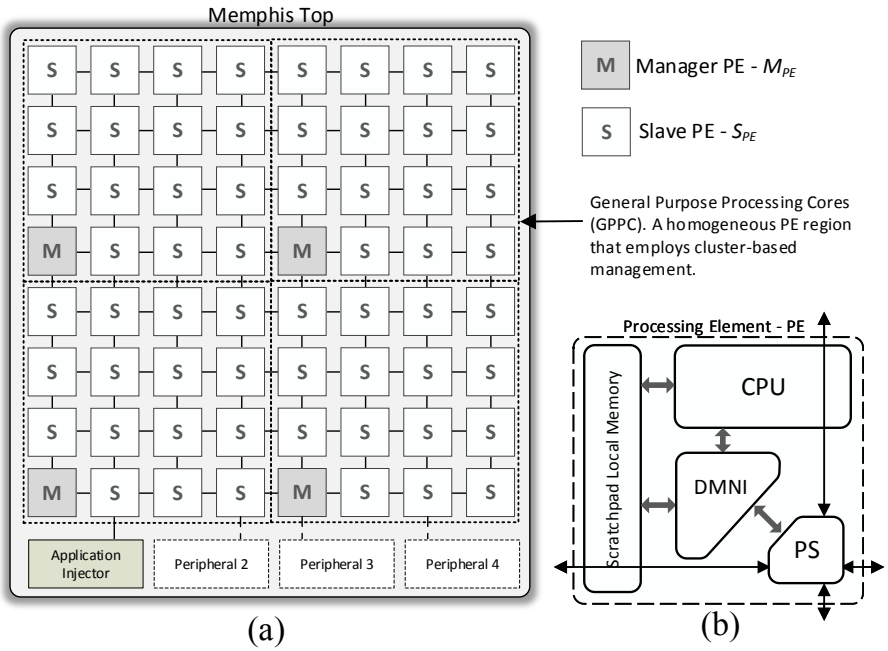

Figura 2 – Visão geral da arquitetura da Memphis [9].

2.2.2 Troca de mensagens

A estrutura dos pacotes que trafegam pela NoC está representada pela Figura 3. O cabeçalho possui dois flits, *target address* e *payload size*. O *target address* informa o destino do pacote. Uma *flag* (D/P) neste flit determina se o pacote é de dados, de tarefas ou dados de periféricos. O *payload size* determina o tamanho da carga útil do pacote.

Do ponto de vista do sistema operacional, uma mensagem contém: *message header* e *message payload*. O *message header* contém os dois flits de cabeçalho, e um terceiro flit – *service header*, que identifica o tipo de mensagem[1] (recepção de mensagem, mapeamento de tarefas, requisição de mensagem, etc.). O *payload* da mensagem é opcional, podendo conter, por exemplo, informação da aplicação ou código objeto de uma tarefa.

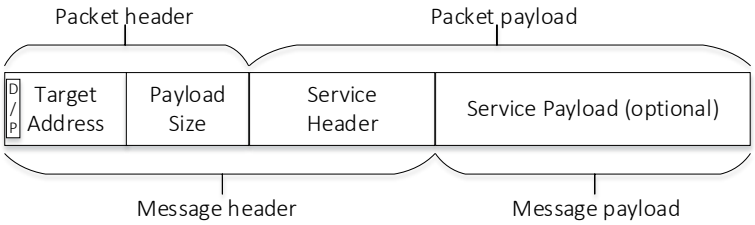

Figura 3 – Estrutura do pacote transmitido através da rede [9].

Cada processador do MPSoC executa um sistema operacional simples (μkernel) com suporte à execução multitarefa e uma API de comunicação *unicast*. Essa API é dis-

[1] https://www.inf.pucrs.br/hemps/docs/HeMPS_Messages.pdf

ponibilizada pelo μkernel para que as tarefas possam enviar e receber pacotes. A Figura 4 apresenta o fluxo para enviar e receber um pacote entre dois PEs diferentes.

As tarefas do usuário se comunicam através de troca de mensagens, assumindo envio não bloqueante e recepção bloqueante. A tarefa de origem (**S**) grava o pacote em uma área de memória do μkernel de origem e continua sua execução. A tarefa de destino (**T**) envia uma solicitação de pacote, tendo sua execução bloqueada. A tarefa **S** envia a mensagem solicitada após a sua produção. A vantagem desse método é que a rede não é bloqueada por pacotes aguardando seu consumo, reduzindo o congestionamento.

Quando o μkernel do PE executando **S** está pronto para enviar o pacote (i.e., mensagem produzida), ele chama a função *send_packet()*, que programa o DMNI para começar a enviar o pacote da memória. No lado do consumidor, quando o DMNI recebe um pacote, ele interrompe o processador. O tratamento de interrupção chama a função *read_packet()* que programa o DMNI para ler o pacote. Depois que o pacote é completamente recebido, o μkernel executa as ações relacionadas ao conteúdo do pacote. Por exemplo, se o pacote contiver dados para uma tarefa do usuário, o corpo do pacote (mensagem no contexto da tarefa do usuário) será gravado no espaço de memória da tarefa.

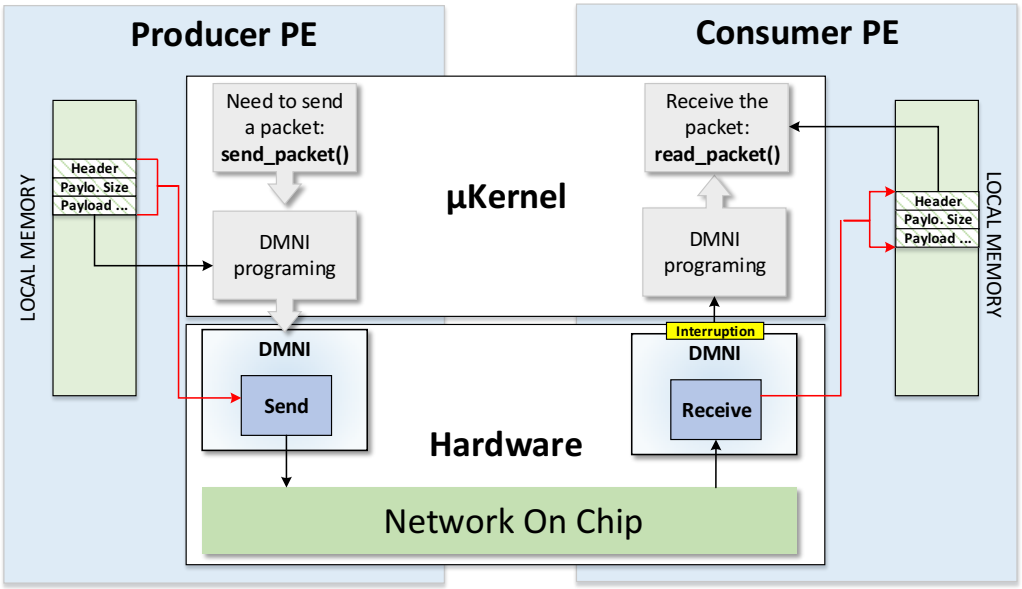

Figura 4 – Processo de transferência de informação entre dois PEs [10].

2.2.3 μkernel

O Many-core Memphis possui dois μkernels: (*i*) μkernel de gerência (*kernel Manager*); (*ii*) μkernel de controle de tarefas (*kernel Slave*).

O μkernel de gerência executa nos processadores **M** (Figura 2). Como mostrado na Figura 5, os processadores de gerência executam tarefas no nível de sistema, como ma-

peamento de tarefas, migração de tarefas, reclustering [2], e monitoramento de atendimento de requisitos de qualidade de serviço.

Os processadores que executam tarefas de usuário são denominados processadores escravos ($S_{PE}s$). Dentre as ações executadas pelo μkernel *slave* cita-se: comunicação entre tarefas, tratamento de interrupções, execução multitarefa (escalonamento), monitoramento, e chamadas de sistema.

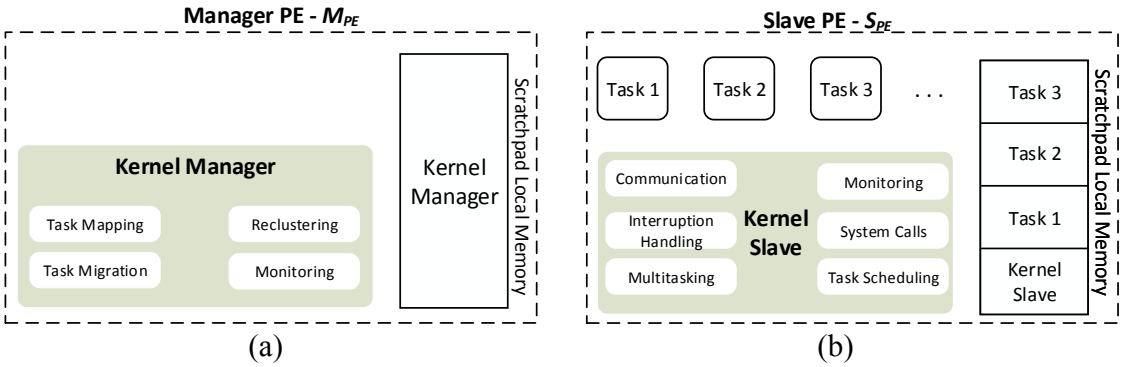

Figura 5 – Estrutura do μkernel [9].

2.3 Interfaces de Comunicação Padronizadas - *Wishbone*

Uma interface de comunicação é um hardware e/ou software capaz de conectar dois componentes, traduzindo a comunicação de um componente para o outro e permitindo a troca de informações entre eles. Desta forma, a adoção de interfaces de comunicação padronizadas facilita a reutilização de IPs, diminuindo o tempo de projeto.

O protocolo de comunicação utilizado neste projeto é Wishbone[2] [8]. O protocolo é similar a um barramento de microprocessador, oferecendo uma solução de integração flexível que pode ser adaptada para uma aplicação específica. Também oferece uma variedade de ciclos de barramento e tamanhos de barramento, e permite que produtos sejam desenvolvidos para uma variedade de fornecedores. Entre suas características estão:

- Interface lógica para IPs simples e compacta, que exige poucas portas lógicas;
- Modos simples de transferência de dados, como, por exemplo:
  - modos de leitura/escrita;
  - ciclo de transferência de blocos;
  - ciclo RMW (read-modify-write), usado para operações atômicas, como em *semáforos*. Durante a primeira metade do ciclo, é realizada uma única transferência

[2]https://opencores.org/cdn/downloads/wbspec_b3.pdf

de dados de leitura. Durante a segunda metade do ciclo, é realizada uma transferência de dados de gravação.

- Barramento de dados de até 64-bits e protocolo baseado em *handshake* que é exemplificado na Figura 6. O sinal STB_O do mestre se mantém ativo até que o escravo responda com ACK_I. Se os dois sinais se manterem ativos, são feitas transmissões em rajadas;
- Utiliza modelo de comunicação Mestre-Escravo.

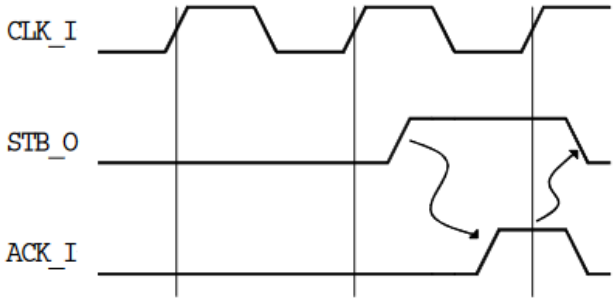

Figura 6 – Protocolo handshake dos sinais de controle Wishbone [8].

Ao aderir o modelo Mestre-Escravo, é preciso determinar um *Wishbone Master* e um *Wishbone Slave*, pois a comunicação só pode ser iniciada pelo mestre. A Figura 7 apresenta os principais sinais de interconexões entre um módulo com interface Wishbone mestre com um módulo Wishbone escravo. O bloco *SYSCON* representa os sinais de controle que são necessários para manter o sincronismo na comunicação. *USER DEFINED* é um bloco que representa um conjunto de sinais de definições de usuário que, por exemplo, podem diferenciar se o protocolo irá assumir transmissões simples ou em rajada. Os dois componentes na comunicação possuem sinais em comum e outros que são específicos. Os sinais que eles possuem em comum são:

– **CLK_I:** Sinal de relógio, um dos sinais de controle utilizado para sincronizar os dois componentes.

– **RST_I:** Sinal de *reset*, um dos sinais de controle utilizado para sincronizar a inicialização da comunicação de ambos os lados.

– **DAT_I:** Dado de entrada.

– **DAT_O:** Dado de saída.

Os sinais específicos do mestre incluem :

– **ACK_I:** Sinal de confirmação de leitura/escrita do dado.

– **ADR_O:** Endereço do dado de saída.

– **CYC_O:** Indica que um ciclo de transmissão vaíldo está em progresso. O sinal é ativado durante a primeira transmissão e fica ativo até o último dado ser transmitido.

– **STB_O:** Indica um ciclo de transferência de dados válido. Para cada *strobe output*, o wishbone escravo precisa responder com um ACK.

– **WE_O:** O *write enable* é o sinal que indica se será uma escrita ou leitura. Nível lógico alto significa escrita, enquanto nível lógico baixo significa leitura.

Os sinais específicos de escravo incluem:

– **ACK_O:** Indica o fim da transmissão de um dado.

– **ADR_I:** Vetor que indica o endereço a ser acessado.

– **CYC_I:** Indica que um ciclo de transmissão de dados válidos está em progresso.

– **STB_I:** Indica que o escravo em questão foi selecionado pelo mestre e que haverá transmissão de dados.

– **WE_I:** Indica se o acesso ao escravo será de escrita ou leitura.

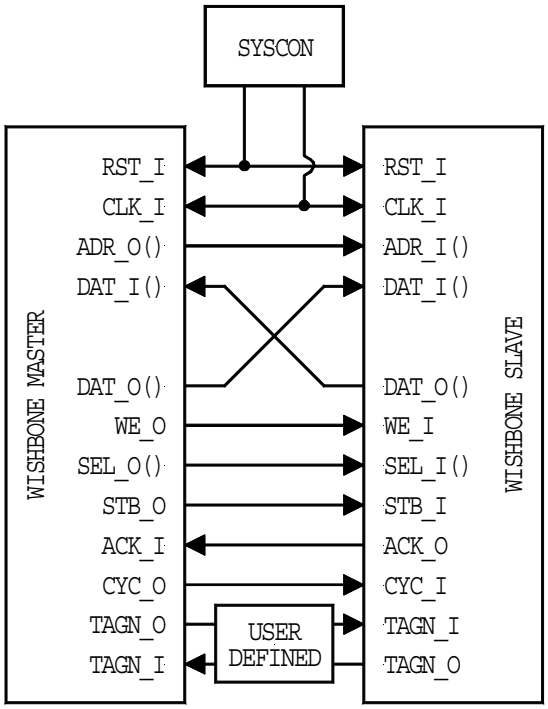

Figura 7 – Conexão padrão de uma interface Wishbone [8].

A Figura 8 é um exemplo de transmissão de escrita. Para a transmissão ser realizada, o mestre aciona os sinais STB_O, CYC_O e WE_O em nível lógico alto, além de colocar as informações e o endereço onde ele gostaria de escrevê-las em DAT_O e ADR_O, respectivamente. ACK_I em nível lógico alto representa a recepção dos dados pelo *wishbone slave*. Enquanto STB_O e ACK_I se mantiverem em nível lógico alto, os dados continuarão a ser transmitidos.

A Figura 9 exemplifica os sinais de controle em uma leitura, seguido de uma transmissão de escrita. A principal diferença entre os sinais nesta operação de leitura, compa-

rada a escrita, está no sinal WE_O que está em nível lógico baixo. A Figura também mostra, após a leitura ser feita, a escrita ser executada no mesmo endereço, por isso ADR_O mantém o mesmo valor durante toda a operação.

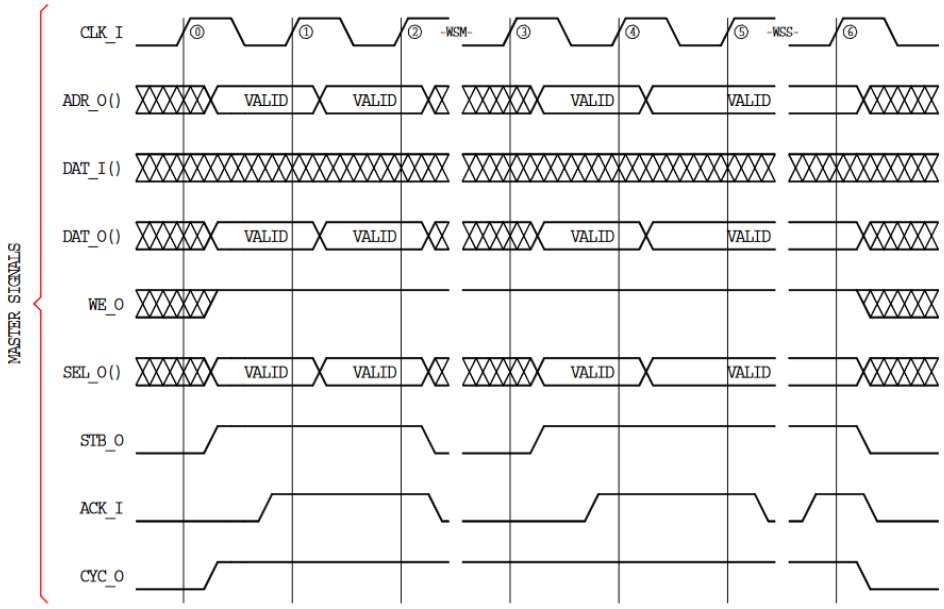

Figura 8 – Sinais wishbone em modo de escrita, enviando dados em rajada [8].

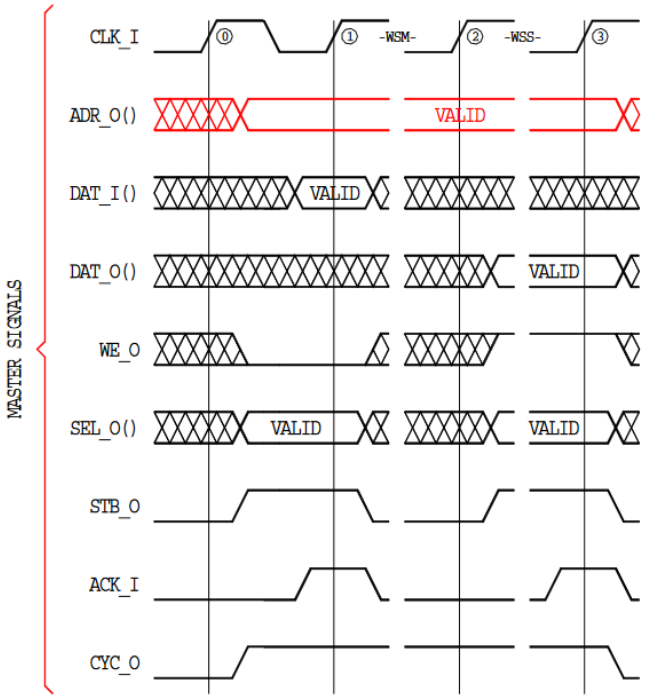

Figura 9 – Sinais de controle em escrita e leitura [8].

3. VISÃO GERAL DO TRABALHO DESENVOLVIDO

A Figura 1 apresenta uma visão alto-nível das etapas do processo de desenvolvimento do projeto, dividindo-as em três etapas:

1. Protocolo de comunicação, correspondendo à forma como a aplicação realiza a comunicação com o μkernel. Fazem parte desta etapa os componentes Aplicação e SysCall (chamadas do sistema);
2. A comunicação entre PE e periférico;
3. Interface de Comunicação, desenvolvimento de uma interface simples para suporte à comunicação com periféricos.

3.1 Nível de Aplicação e Protocolos de Comunicação

O nível de aplicação é o nível mais alto do protocolo no trabalho proposto. A implementação da aplicação depende do protocolo de comunicação. Explorou-se, durante este TCC, dois protocolos de comunicação entre PEs e periféricos.

A Figura 10 apresenta o primeiro protocolo, *connection-less protocol*. Neste protocolo o estabelecimento da comunicação entre o periférico e o PE é independente da aplicação. O periférico uma vez tendo atendido e finalizado a solicitação da aplicação, é liberado de forma automática para atender a outras solicitações, de outras aplicações. A numeração nas setas apresenta a ordem dos eventos. Quando o periférico atende a um pedido (seta 4), ele finaliza a comunicação (seta 5), permitindo outro pedido de conexão. Liberar a conexão assim que responder a requisição impede que uma dada conexão monopolize o uso do periférico. Apesar desta vantagem, este não foi o protocolo final implementado, conforme será explicado ao longo desta Seção.

A Figura 11 apresenta o segundo protocolo de comunicação, efetivamente adotada neste projeto, *connection-based protocol*. Neste protocolo, a aplicação tem controle sobre a conexão, recebendo um retorno logo após a conexão ser estabelecida. Diferente do protocolo anterior, este só desfaz a conexão quando a aplicação solicita, através de uma chamada de sistema denominada *PERIPHERAL_RELEASE* (detalhada no capítulo 4). Enquanto a conexão não é concluída, a conexão é mantida e a aplicação pode requisitar escritas e leituras quantas vezes forem necessárias (repetição do processo das setas 6 a 10).

Como foi citado no início desta Seção, o primeiro protocolo possui a característica de uso mais justo do periférico, evitando *starvation*. O segundo protocolo pode bloquear

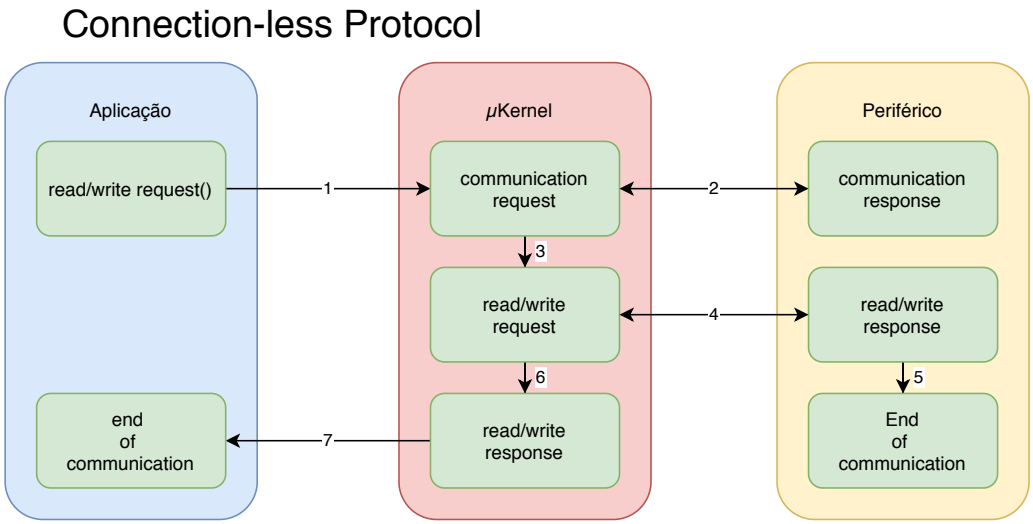

Figura 10 – Primeiro protocolo de comunicação avaliado (*Connection-less Protocol*).

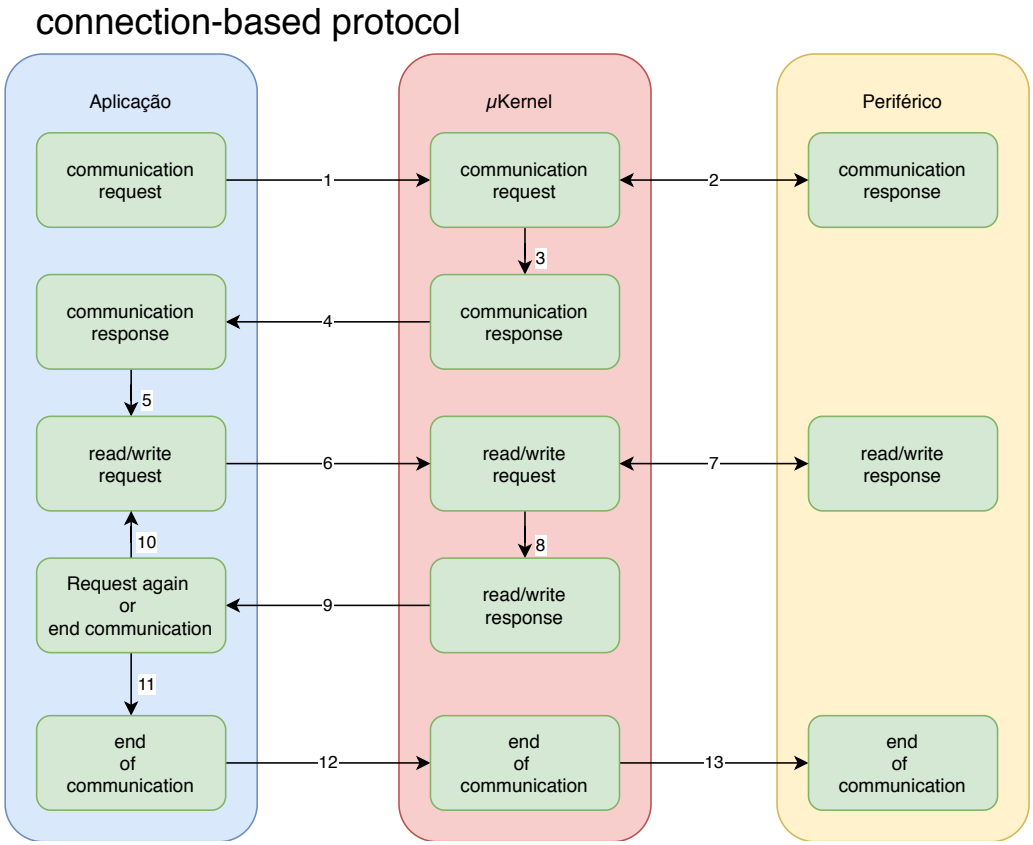

Figura 11 – Protocolo devidamente implementado (*Connection-based Protocol*).

o periférico permanentemente. Adotou-se o segundo protocolo por ele permitir transações em rajada. Soluções simples baseadas em temporizadores podem ser implementadas futuramente para evitar que o periférico fique conectado exclusivamente em uma determinada aplicação. No entanto, a eficiência de cada um dos métodos também depende das características do periférico que está sendo acessado.

3.2 Nível de Comunicação

As alterações feitas no Many-core Memphis estão restritas ao μ*kernel slave*, o qual gerencia a execução das tarefas das aplicações do usuário. Este μ*kernel* é responsável por preencher as informações necessárias em um pacote, de acordo com o serviço solicitado. Inicialmente haveria uma estrutura de pacotes própria, porém de forma a manter compatibilidade com o μkernel, decidiu-se reutilizar as funções de envio e recepção já existentes entre os processadores, utilizando a estrutura *ServiceHeader*, própria da Memphis. Manter está estrutura permitiu a utilização da função *send_packet*, a qual faz o envio de pacotes para a NoC. O Capítulo 4 descreve a estrutura destes pacotes.

3.3 Interface de Comunicação

A interface de comunicação possui duas faces. Um lado se comunica com a NoC e o outro lado com os sinais do protocolo Wishbone. O planejamento original previa a utilização da interface de rede (NI) do trabalho de Tadeu Marchese [5], porém devido ao curto periodo de tempo disponível para esta etapa, desenvolveu-se uma interface mais simples em System-C, a qual simula as transações do protocolo Wishbone. Esta interface recebe as solicitações oriundas dos PEs, fazendo o tratamento dos serviços e gerando os pacotes de resposta necessários para verificar se a comunicação aplicação/periférico está operando corretamente. Sua estrutura e operação é detalhada no Capítulo 5.

4. CAMADA DE APLICAÇÃO E API COM PERIFÉRICO

Este Capítulo está organizado em 3 Seções. A Seção 4.1 apresenta as funções no nível do usuário para a comunicação com periféricos. A Seção 4.2 descreve os serviços no nível de μkernel e a estrutura dos pacotes. A Seção 4.3 apresenta resultados relacionados à execução do software até a camada de rede.

4.1 Camada de Aplicação

Para a camada de aplicação, foram desenvolvidas quatro funções na API disponibilizada pelo μkernel para as aplicações. Estas novas funções são responsáveis por permitir que as aplicações possam se comunicar com os periféricos de uma forma abstrata:

```
PeripheralCommunication(status, perID);
PeripheralRead(data, perID, address);
PeripheralWrite(data, perID, address);
PeripheralRelease(perID);
```

Onde:

- *status* – ponteiro que armazena o resultado da solicitação de conexão, assumindo o valor 1 quando a conexão foi bem-sucedida e -1 caso falhe.
- *perID* – identificador do periférico. No processo de compilação da Memphis, os periféricos conectados recebem estes identificadores. A NoC decodifica o identificador para localizar o periférico.
- *data* – estrutura do tipo *message*, a qual armazena uma mensagem e seu tamanho.
- *address* – endereço inicial de onde a mensagem será escrita ou lida no periférico.

A função *PeripheralCommunication* tenta realizar a conexão com o periférico. O desenvolvedor da aplicação pode tanto decidir pelo bloqueio da aplicação até a liberação do periférico caso a conexão retorne um valor negativo, ou decidir realizar o acesso posteriormente. Por exemplo, caso o estabelecimento de uma conexão com o periférico não seja estritamente necessário e a solicitação for negada, o PE pode continuar executando a tarefa e fazer uma nova solicitação posteriormente. No entanto, o desenvolvedor pode escolher deixar a aplicação em um estado de bloqueio enquanto repetidamente solicita um novo acesso ao periférico. Por exemplo, através de um laço *while* testando o valor de retorno, até este ser positivo.

Uma vez que a conexão com o periférico foi realizada com sucesso, as funções *PeripheralRead* e *PeripheralWrite* são responsáveis pela comunicação com o periférico definido por `perID`. É importante destacar que a execução destas funções sem um prévio estabelecimento de conexão implica no descarte dos pacotes.

A função *PeripheralRelease* libera o periférico endereçado por `perID` para aceitar novas conexões de outras aplicações. Quando esta ordem chegar no periférico, caso este esteja efetivamente conectado ao PE solicitante, as informações referentes à conexão são reiniciadas, caso contrário, o pacote será apenas descartado (visto que o PE que enviou a ordem não possui uma conexão efetiva com o periférico). Desta forma, se houver uma nova solicitação, o periférico poderá permitir a conexão.

4.2 Camada da API com o Periférico

As funções descritas na Seção 4.1 acionam chamadas de sistemas. A Figura 12 apresenta o fluxo das chamadas. Quando a aplicação executa, por exemplo, *PeripheralCommunication*, a chamada de sistema *PERIPHERAL_COM* é acionada. Esta chamada invoca a função *send_peripheral_request* e atribui o valor 1 para a *flag* responsável por bloquear a tarefa até que seja recebida uma resposta do periférico. A função *send_peripheral_request* preenche uma estrutura chamada de *ServiceHeader* e faz o envio do pacote para NoC.

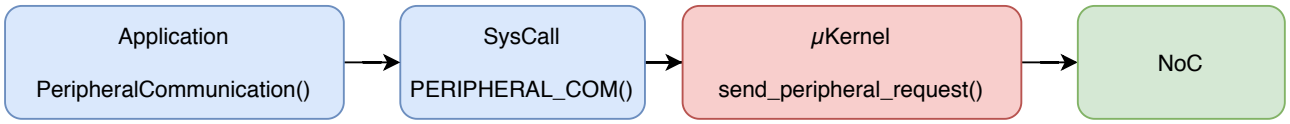

Figura 12 – Fluxo das chamadas de função até a rede.

A Figura 13 apresenta a forma como é montado um determinado pacote para envio do mesmo para a NoC. A linha 3 recebe um ponteiro para uma estrutura de cabeçalho de pacote, *ServiceHeader*, o qual é preenchido nas linhas 5 a 11, para posterior envio pela função *send_packet* (linha 13). A função *send_packet* é responsável por transmitir o pacote do μkernel para a NoC, através do DMNI. Os parâmetros da função *send_packet* compreendem:

- Estrutura *ServiceHeader*, com treze flits. Está estrutura possui campos que representam cada um dos parâmetros necessários para executar os serviços do μkernel. Nem todos os serviços utilizam todos os campos. Neste caso, não é obrigatório especificá-los.

- Endereço inicial do *payload*, dado que este é um serviço de requisição, não há dados a enviar . Só é necessário preencher este campo quando for uma requisição de escrita, pois serão enviados mais dados além das informações do serviço.
- Tamanho do *payload*. Define a quantidade de flits a se enviar no *payload*.

```
1  void send_peripheral_request( /* parametros... */) {
2      // gera um ponteiro com endereço válido.
3          ServiceHeader *p = get_service_header_slot();
4          // ID do periférico
5          p->header = perID;
6          // identificação do serviço
7          p->service = PERIPHERAL_REQUEST;
8          // identificação da tarefa
9          p->task_ID = task_id;
10         // PE de origem
11         p->source_PE = source;
12         // Função para envio do pacote
13         send_packet(p, 0, 0);
14  }
```

Figura 13 – Código da função send_peripheral_request().

Quando se solicita conexão com um periférico, o pacote não possui *payload*, apenas os treze flits padrões. Por isso o segundo e terceiro argumentos (da linha 13), estão com o valor zero.

A proposta deste TCC consistia no desenvolvimento de sete serviços no nível do μkernel, compreendendo a solicitação de conexão, as duas respostas de conexão (tanto positiva, quanto negativa), as requisições de leitura e escrita e suas respectivas respostas. A Figura 14 mostra a estrutura dos pacotes das mensagens, considerando uma organização própria, diferente da estrutura da Memphis.

Flits	1	2	3	4	5	6	7	N
Request:	TARGET	SIZE	REQUEST	SOURCE	TASK ID			
Request ACK:	TARGET	SIZE	REQ. ACK	PERIF ID	TASK ID			
Request NACK:	TARGET	SIZE	REQ. NACK	PERIF ID	TASK ID			
Read request:	TARGET	SIZE	READ REQ.	SOURCE	TASK ID	READ ADDRESS	READ SIZE	
Read response:	TARGET	SIZE	READ RESP.	PERIF ID	TASK ID	STATUS	DATA...	
Write request:	TARGET	SIZE	WRITE REQ.	SOURCE	TASK ID	WRITE ADDRESS	WRITE SIZE	DATA...
Write response:	TARGET	SIZE	WRITE RESP	PERIF ID	TASK ID	STATUS		

Figura 14 – Estrutura dos pacotes originalmente propostos para o projeto [5].

Observou-se que utilizar a proposta de pacotes apresentada na Figura 14 implicaria na perda de compatibilidade com o *driver send_packet()*. Esta perda de compatibilidade implicaria no desenvolvimento de uma nova API para envio e recepção de pacotes. Assim, adaptou-se a estrutura de pacotes da Figura 14 para o formato de *message header* de 13 flits da Memphis. Além disto, criou-se um oitavo serviço, *peripheral_release*. A Figura 15 apresenta a nova estrutura dos pacotes para a comunicação com periféricos.

Descreve-se abaixo os oito serviços executados no μkernel para a comunicação com os periféricos.

flit 0	flit 1	flit 2	flit 3	flit 4	flit 5	flit 6	flit 7	flit 8	flit 9	flit 10	flit 11	flit 12	payload
Target	payload_size	service	p_task	c_task	Source_PE	status		msg_length				initial_address	
					Per_ID								

flit 0	flit 1	flit 2	flit 3	flit 4	flit 5	flit 6	flit 7	flit 8	flit 9	flit 10	flit 11	flit 12	payload
PERIPHERAL_REQUEST													
Target	payload_size	0x00400	p_task		Source_PE								
PERIPHERAL_REQUEST_ACK													
Target	payload_size	0x00410		c_task	Per_ID	status							
PERIPHERAL_REQUEST_NACK													
Target	payload_size	0x00420		c_task	Per_ID	status							
READ_REQUEST													
Target	payload_size	0x00430	p_task		SOURCE_PE			size				read_address	
READ_RESPONSE													
Target	payload_size	0x00440		c_task	Per_ID	status		size					data
WRITE_REQUEST													
Target	payload_size	0x00450	p_task		SOURCE_PE			size				write_address	data
WRITE_RESPONSE													
Target	payload_size	0x00460		c_task	Per_ID	status							
PERIPHERAL_RELEASE													
Target	payload_size	0x00490	p_task		Source_PE								

ORIGINADO DO PE
ORIGINADO DO PERIFÉRICO

Figura 15 – *Message header* dos pacotes utilizado na Memphis, e sua relação com os serviços de comunicação com periféricos.

- **PERIPHERAL_REQUEST:** originado do PE, este serviço solicita uma conexão com o periférico.
- **PERIPHERAL_REQUEST_ACK:** originado do periférico, este serviço é a resposta positiva ao pedido de conexão. Quando o periférico responde o *request* com este serviço, ele armazena o PE de origem e o ID da tarefa solicitante.
- **PERIPHERAL_REQUEST_NACK:** originado do periférico, este serviço é a resposta negativa ao pedido de conexão. O periférico responde com esse serviço caso ele já esteja conectado a outro PE.
- **READ_REQUEST:** originado do PE, este serviço é a solicitação de leitura de um periférico. No corpo do pacote enviado, além de conter as informações da origem, há também o endereço inicial e o tamanho da informação que se deseja obter.
- **READ_RESPONSE:** originado do periférico, este serviço é a resposta ao pedido de leitura. Este serviço só é enviado caso a solicitação vier do PE e tarefa em que o periférico está conectado. O corpo de seu pacote contém o tamanho da mensagem lida e a própria mensagem em seu *payload*, há também uma *flag* informando se a leitura foi efetuada com sucesso.
- **WRITE_REQUEST:** originado do PE, este serviço é a solicitação de escrita em um periférico. No corpo do pacote enviado, além de haver as informações da origem, há também o tamanho da mensagem e o endereço inicial de onde ela será escrita, além de conter a própria mensagem em seu *payload*.

- **WRITE_RESPONSE:** originado do periférico, este serviço é a resposta ao pedido de escrita. Este serviço só é enviado caso a solicitação vier do PE e tarefa com quem o periférico está conectado. O corpo de seu pacote contém uma *flag* informando se escrita foi efetuada com sucesso.
- **PERIPHERAL_RELEASE:** originado do PE, este serviço libera o periférico para novas conexões. Este serviço não exige resposta, quando o periférico o recebe, as informações do PE e da tarefa são descartados para que uma nova conexão possa ser realizada.

4.3 Resultados

Esta Seção apresenta o processo de troca de pacotes gerados no nível de software, como apresentado no fluxo da Figura 12. O cenário simulado compreende uma tarefa mapeada no PE 2x2, *send.c*, que solicita acesso ao periférico conectado na porta norte do PE 0x2, como mostra a Figura 16. As linhas vermelhas mostram o caminho percorrido pelos pacotes originados pelo PE 2x2 em direção à interface, e as linhas verdes o caminho dos pacotes de resposta.

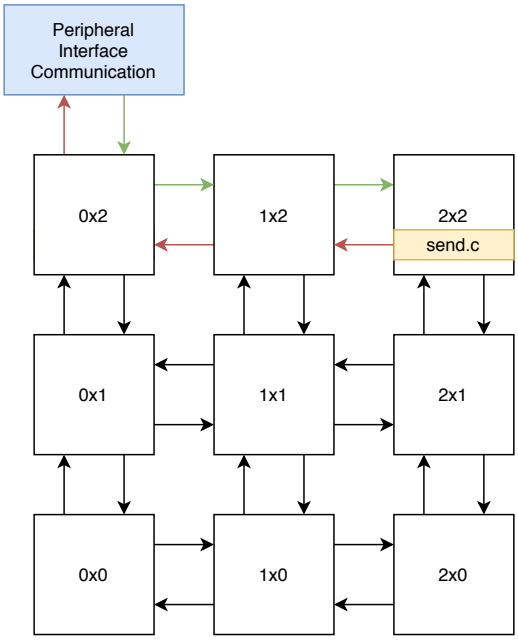

Figura 16 – Cenário utilizado para a obtenção dos resultados.

4.3.1 Aplicação

Apresenta-se abaixo um trecho parcial da tarefa *send.c*. Neste código são inicializadas as variáveis `status` e `perID`. A função *PeripheralCommunication* envia como parâmetros para a *syscall* o ponteiro da variável `status` e o identificador do periférico (`perID`). Envia-se o endereço da variável `status` porque quando o periférico responder à solicitação a resposta será armazenada nesta variável.

```
unsigned int status = 0;
unsigned int perID = 0xc0000002;
PeripheralCommunication(&status, perID); // status = ack/nack
```

A função *PeripheralCommunication* é de fato um *define*, que permite que a aplicação acione a chamada de sistema, mudando do espaço de memória de aplicação para o espaço de memória do μkernel.

```
#define PeripheralCommunication(status, per_id)
while( !SystemCall( PERIPHERAL_COM, (unsigned int*) status, per_id, 0))
```

A função *SystemCall* gera a interrupção de software necessária à interrupção da aplicação. A execução da *SystemCall* implica em um salto para uma posição de memória pré-definida pelo processador, onde há o salvamento de contexto da aplicação e o salto para a rotina correspondente no μkernel (*SysCall*). O parâmetro `PERIPHERAL_COM` é o identificador da chamada de sistema. Quando o μkernel do processador é acionado, ele executa a parte do código correspondente ao valor do `PERIPHERAL_COM` na função *SysCall*.

4.3.2 μkernel

O código abaixo apresenta, na linha 1, o valor do *PERIPHERAL_COM*. Na sequência, ilustra-se parte do código da função *Syscall*, repensável por realizar a chamada de sistema. O código da função *send_peripheral_request* está na descrito Seção 4.2. A linha 8 acionará o escalonamento de tarefas após a execução da chamada de sistema. A linha 9 indica que a tarefa está aguardando uma mensagem, bloqueando-a até a recepção da resposta da requisição (ACK/NACK). Este bloqueio implica, na prática, no não escalonamento da tarefa pelo μkernel.

```
#define PERIPHERAL_COM                11

int Syscall(unsigned int service, unsigned int arg0, unsigned int arg1, unsigned int arg2) {
    switch (service) {
        ...
        case PERIPHERAL_COM:
            send_peripheral_request(arg1, get_task_location(current->id), current->id);
            schedule_after_syscall = 1;
            current->scheduling_ptr->waiting_msg = 1;
            return 0;
            break;
}
```

4.3.3 NoC

A Figura 17 ilustra o pacote começando a ser transmitido para NoC, através do DMNI (NI da MEMPHIS). Como mostra a Figura 15, as informações relevantes para o serviço de *PERIPHERAL_REQUEST* estão contidas nos seis primeiros flits:

– *flit 0*: destino do pacote. A parte alta do pacote, 0xC, indica que é um pacote para periférico, e a parte baixa, x0002 (8 bits para endereço X e oito bits para endereço Y), o endereço de rede.

– *flit 1*: contém o tamanho do *payload* do pacote. Neste caso, o valor xB, ou seja, o pacote possui o tamanho padrão do *message header*, 13 flits: 11 de *payload* e 2 de cabeçalho do pacote.

– *flit 2*: identificador do serviço. A Figura 15 apresenta os valores dos identificadores para cada um dos serviços.

– *flit 3*: identificador da tarefa. Como a tarefa send.c foi a primeira tarefa a ser alocada no sistema, seu o ID é igual a zero. Cada tarefa possui um ID único, que é gerado de forma crescente conforme novas tarefas são alocadas no sistema.

– *flit 4*: é um flit vazio no pacote deste serviço.

– *flit 5*: PE origem deste pacote, no caso PE 2x2.

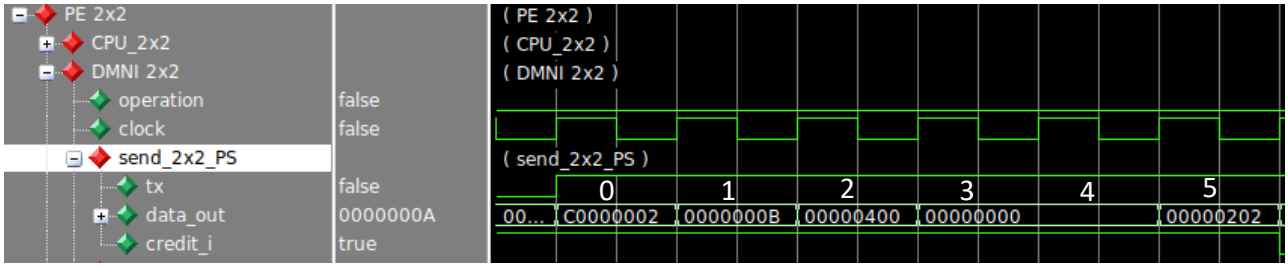

Figura 17 – Pacote do serviço PERIPHERAL_REQUEST sendo injetado na NOC através do PE 2x2.

O pacote é enviado através da NoC, do PE 2x2 até o PE 0x2, quando ele sai da NoC pela porta norte. A Figura 18 mostra o pacote gerado no PE 2x2 sendo transmitido ao periférico através da porta norte do PE 0x2.

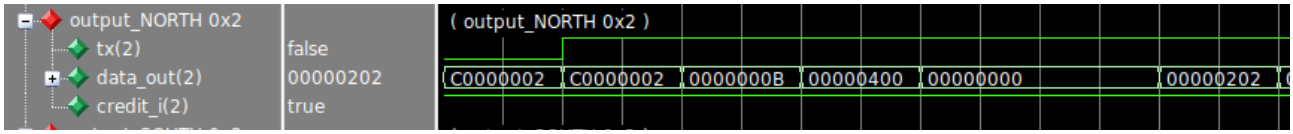

Figura 18 – O pacote do serviço PERIPHERAL_REQUEST saindo da porta norte do PE 0x2, em direção ao periférico.

A Seção 5.3 apresenta a continuação da execução desta tarefa no periférico.

5. INTERFACE DE COMUNICAÇÃO E PERIFÉRICO

Desenvolve-se uma interface de comunicação, em System-C, para validar os serviços desenvolvidos e as alterações realizadas no μkernel dos PEs. Sua estrutura possui uma máquina de estados para recepção de pacotes provenientes da NoC, outra para enviar pacotes para NoC, sinais de controle capazes de fazer a tradução dos sinais para o barramento Wishbone, e uma memória. Esta memória corresponde ao periférico conectado ao barramento Wishbone. As operações com a memória são realizadas através do barramento Wishbone. A Figura 19 apresenta a estrutura da interface, em conjunto com o periférico (*memory*).

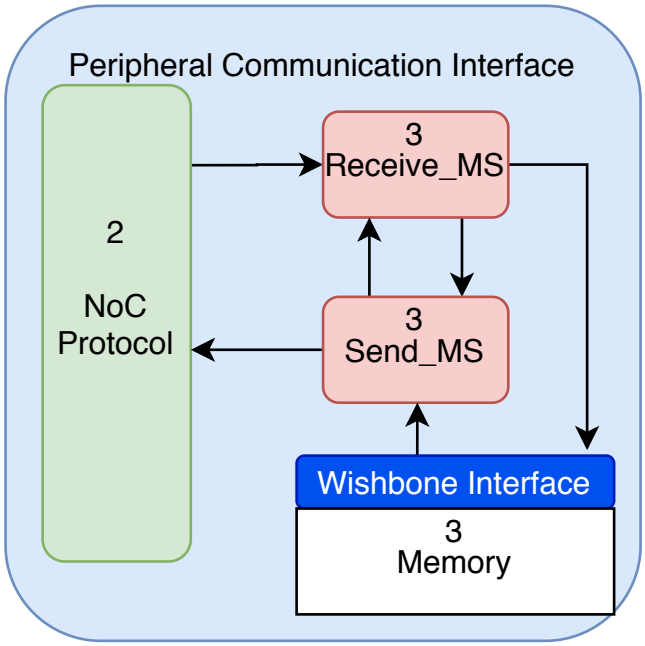

Figura 19 – Estrutura da Interface de Comunicação.

5.1 Interface de Comunicação

Uma máquina de estados (FSM) de recepção, é responsável pelo tratamento dos pacotes oriundos da NoC. A Figura 20 apresenta a FSM de recepção. Os estados iniciais, `HEADER`, `PAYLOAD_SIZE`, `SERVICE` e `TASK` realizam a leitura dos 4 primeiros flits. O estado `EMPTY_FLIT` realiza a contagem de flits para direcionar a FSM para o estado correspondente ao tratamento do serviço contido no pacote. A partir deste estado, o serviço contido no pacote é tratado:

- **PERIPHERAL_REQUEST**: a FSM lê a origem do pacote (`source PE`) e progride para o estado `WAITING_SEND`. Se a FSM estiver conectada a algum PE, a FSM envia um

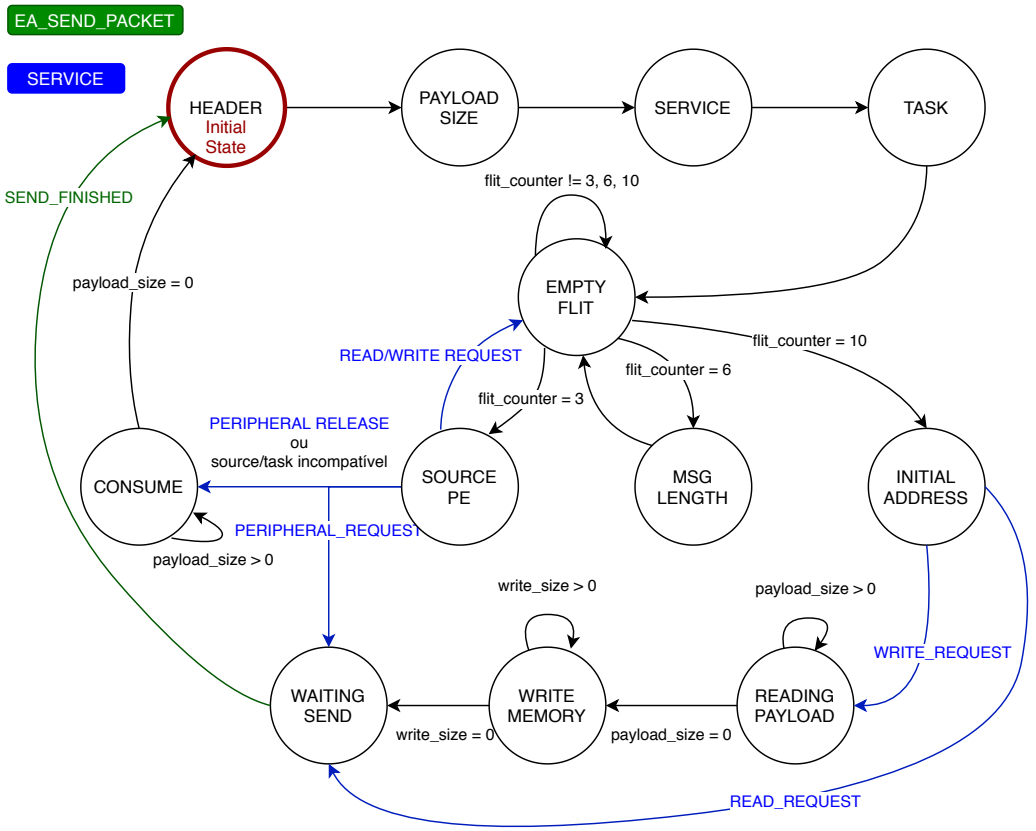

Figura 20 – Máquina de estados de recepção de pacotes oriundos da NoC.

pacote NACK para o PE origem do pacote, caso contrário, é enviado um pacote ACK, armazenando as informações do PE que solicitou a conexão (endereço e identificador da tarefa).

- **READ_REQUEST**: a FSM continua lendo os campos do pacote recebido para determinar o endereço inicial da leitura e a quantidade de dados solicitados. Após ler o endereço a ser armazenado, a FSM de envio é acionada quando a FSM de recepção chega no estado de `WAITING_SEND`.
- **WRITE_REQUEST**: a FSM continua lendo os campos do pacote recebido para determinar o endereço inicial da escrita e a mensagem a ser escrita. O estado `READING PAYLOAD` serve para a leitura da mensagem, enquanto o estado `WRITE_MEMORY` faz a escrita do *payload* na memória.
- **PERIPHERAL_RELEASE**: O periférico libera as informações do PE conectado e consome o restante da mensagem, disponibilizando a interface para uma nova conexão.

O estado `CONSUME` serve para consumir as informações que estão sendo lidas sem fazer nenhum tratamento, descartando os flits desnecessários. As duas ocasiões onde isto ocorre são:

- quando o campo de origem ou da tarefa não correspondem com as informações armazenadas no periférico;

- quando o pacote possui o serviço de liberação do periférico (PERIPHERAL_RELEASE), pois a interface consegue conferir se a origem da solicitação e a tarefas são equivalentes à informação salva na mesma com os primeiros seis flits da mensagem.

A FSM de envio, é responsável pelo envio de pacotes para a NoC. A Figura 21 apresenta a FSM de envio.

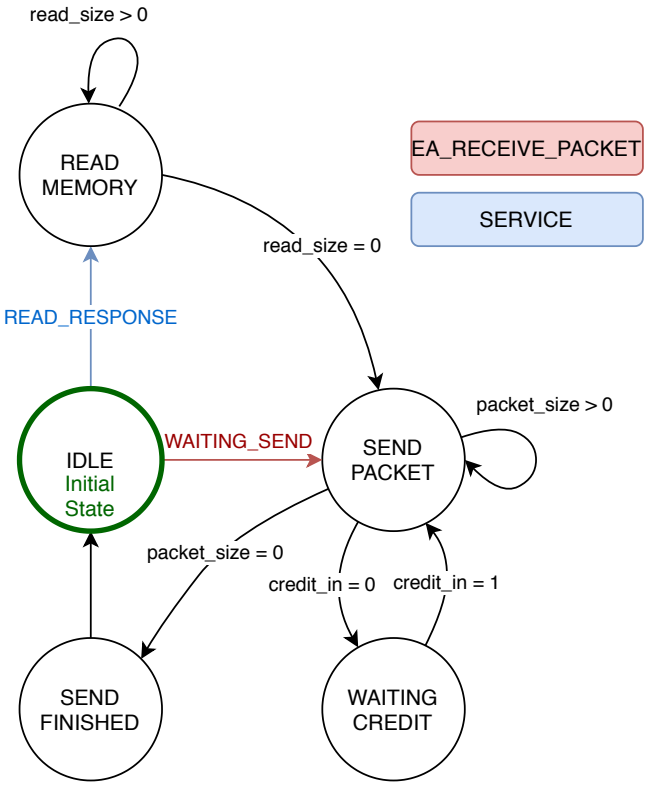

Figura 21 – Máquina de estados para envio de pacotes para a NoC.

A FSM de envio responde a quatro servicos: (*i*) PERIPHERAL_REQUEST_ACK; (*ii*) PERIPHERAL_REQUEST_NACK; (*iii*) READ_RESPONSE; (*iv*) WRITE_RESPONSE.

A FSM de envio é acionada quando a FSM de recepção alcança o estado WAITING_SEND. Antes de sair do estado IDLE, uma estrutura chamada *packet* é preenchida com as informações necessárias para responder o serviço solicitado pelo PE. Quando o serviço de resposta for READ_RESPONSE, a FSM passa para um outro estado, READ_MEMORY, antes de começar o envio do dados no estado SEND_PACKET. Em READ_MEMORY, é feita a leitura da memória e os dados lidos são incluídos na estrutura *packet*.

O estado de WAITING_CREDIT é acionado quando o crédito de entrada estiver em nível lógico baixo. Isto ocorre quando a NoC não está disponível para receber o próximo flit. A FSM se mantêm nesse estado até que o crédito volte para o nível lógico alto, continuando o envio do pacote. Um contador gerencia o envio dos dados, baseado na informação do tamanho da mensagem a ser transmitida. Quando este contador chegar a zero, a FSM prossegue para o estado SEND_FINISHED. Este último estado serve para avisar a FSM de recepção que o pacote no qual ela estava tratando foi enviado.

5.2 Memória (Periférico)

A memória é modelada como um vetor de endereços. O objetivo de modelar este periférico é permitir testar os serviços de escrita e leitura desenvolvidos para comunicação com periféricos. Esta memória comunica-se através do barramento Wishbone.

O código apresentado na Figura 22 mostra como os sinais do protocolo Wishbone controlam a memória. Os sinais sem o prefixo **mem** pertencem à interface, enquanto os que possuem pertencem a memória. A linha 10 verifica se o acesso à memória corresponde a uma operação de leitura, enquanto o da linha 13 verifica se corresponde a uma operação de escrita. A diferença está no sinal de *write enable* (mem_we_i), se seu valor for 0 é uma leitura, se for 1 é uma escrita.

```
1  void test_peripheral::wishbone() {
2
3          if (reset.read() == 1) {
4                  mem_adr_i = 0;
5                  mem_dat_i = 0;
6                  mem_dat_o = 0;
7                  mem_ack_o = 0;
8          } else {
9
10                 if (mem_stb_i == 1 && mem_we_i == 1 && mem_cyc_i == 1) {
11                         memory[mem_adr_i] = mem_dat_i;
12                         mem_ack_o = 1;
13                 } else if (mem_stb_i == 1 && mem_we_i == 0 && mem_cyc_i == 1) {
14                         mem_dat_o = memory[mem_adr_i];
15                         mem_ack_o = 1;
16                 } else {
17                         mem_ack_o = 0;
18                 }
19         }
20 }
```

Figura 22 – Código responsável por escrever e ler as informações da memória.

A Figura 23 ilustra uma simulação para escrita. A interface aciona os sinais *stb_o*, *we_o* e *cyc_o*, gravando o endereço em *adr_o* e o dado em *dat_o*, um ciclo depois os sinais da memória recebem os valores, fazem a escrita e retornam o *ack*. O próximo valor só é atribuído depois do *ack* ser retornado. Os sinais destacados em vermelho são gerados pela interface (*wishbone master*), e os em amarelo gerados pela memória(*wishbone slave*). O protocolo implementado assume envio de dados em rajada, o que significa que enquanto os sinais de controle, stb_o e ack_i, estiverem ativos, os dados continuarão sendo escritos na memória.

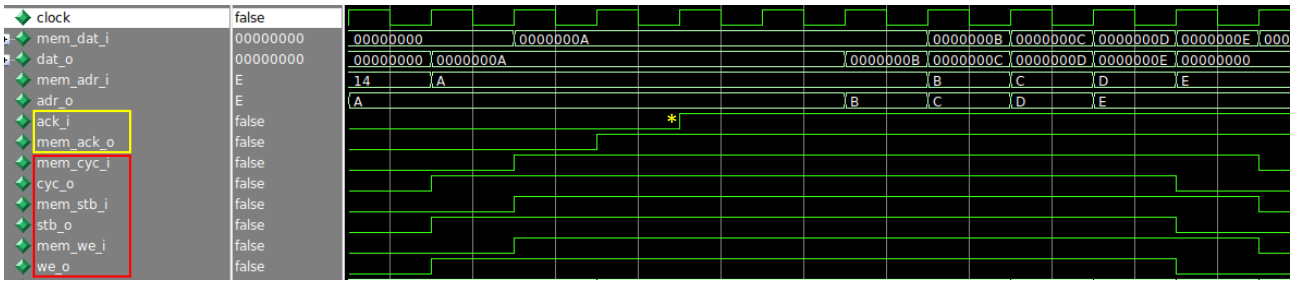

Figura 23 – Sinais de controle de wishbone para escrita na memória.

5.3 Resultados

Esta Seção apresenta exemplos de simulações relativas aos diversos serviços realizados pela interface.

5.3.1 Estabelecimento de Conexão

As formas de onda da Figura 24 mostram o período em que as informações do pacote com o serviço de conexão estão sendo lidas pela interface. Em uma requisição de conexão, todas as informações necessárias estão contidas nos seis primeiros flits do pacote. Assim que a interface lê a origem do pacote no estado `SOURCE_PE`, o próximo estado é o `WAITING_SEND` (estado de espera, que aciona a FSM de envio de mensagem). O * na Figura destaca o momento em que a FSM de envio sai do estado `IDLE` e inicia o processo de resposta ao pedido de conexão.

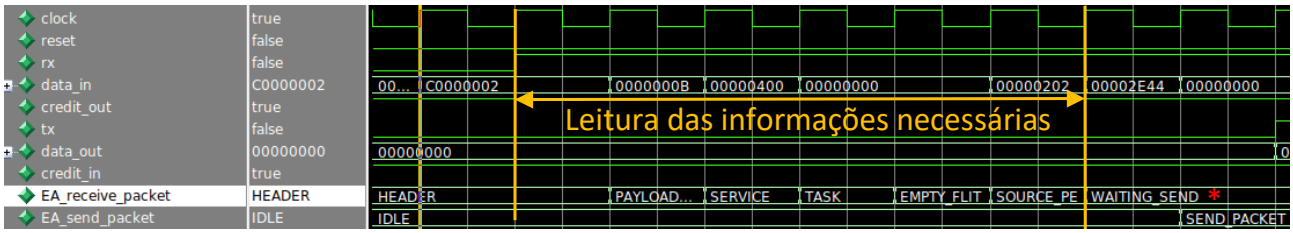

Figura 24 – Chegada do pacote de requisição de conexão com o periférico.

Após acionar a FSM de envio, o * na Figura 25 mostra o momento em que a interface gera sinais para armazenar as informações do PE conectado. Esta Figura também ilustra o envio do pacote de resposta. Observando (1) e (3) e comparando com o pacote de solicitação de conexão da Figura 24, percebe-se que são as mesmas informações, porém com as posições invertidas, garantindo que o pacote está direcionado para o PE origem. O valor de (2) é 0x00000410 que, de acordo com a figura 15, é o código para PERIPHERAL_REQUEST_ACK, confirmando que o periférico aceitou a conexão.

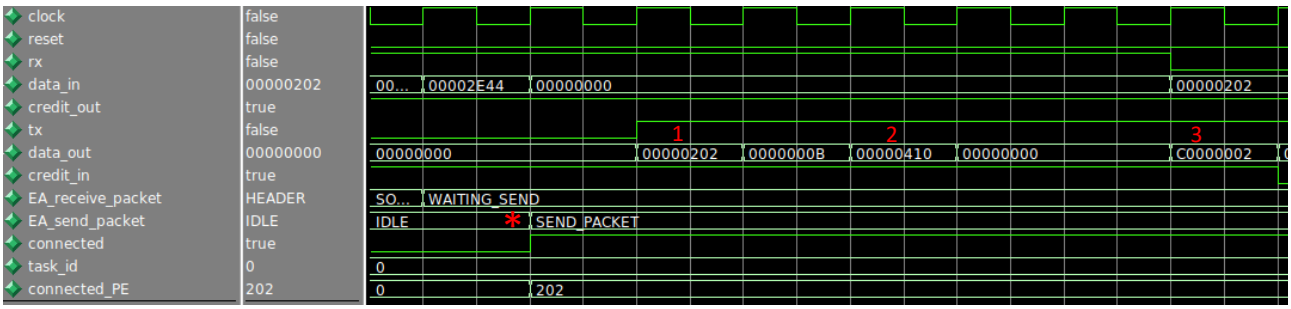

Figura 25 – Resposta à solicitação e estabelecimento de conexão com o periférico.

5.3.2 Escrita na Memória

A Figura 26 mostra um pacote de solicitação de escrita dividido em duas partes:

1. Leitura do pacote, com armazenamento das informações relevantes. Nesta parte, destacam-se os pontos (A) e (B). (A) é o tamanho do *payload*, igual a x10 (11 flits para o *message header* e 5 para o tamanho da mensagem a ser escrita). O tamanho da mensagem a ser escrita está destacada no *message header* em (B). O destaque (1) indica o endereço inicial da escrita, também contido no *message header*.

2. A segunda parte inicia em (C), correspondendo ao *payload* com os dados a serem escritos na memória. Neste exemplo o endereço inicial é igual ao valor do primeiro dado a ser escrito. O destaque *, indica o começo do processo de escrita na memória, processo este apresentado na Figura 27.

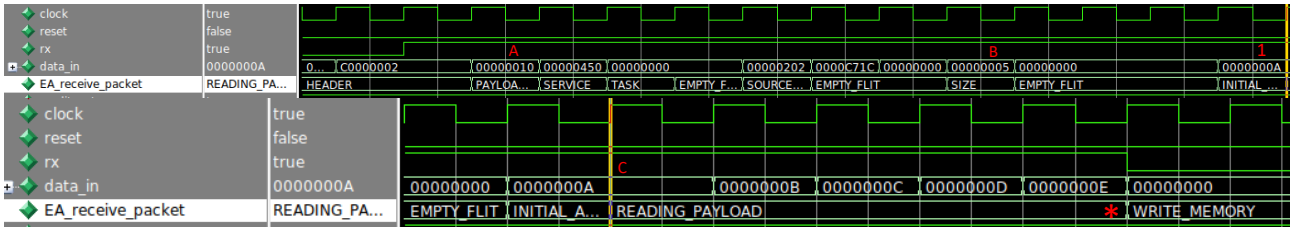

Figura 26 – Pacote de solicitação de escrita chegando ao periférico.

A memória foi pré-inicializada, visando facilitar o processo de depuração. A memória, neste cenário, possui 30 endereços, inicializada com os seguintes valores:

- Do endereço 0 ao 9: valor 0;
- Do endereço 10 ao 19: Valor 1;
- Do endereço 20 ao 29: valor 2.

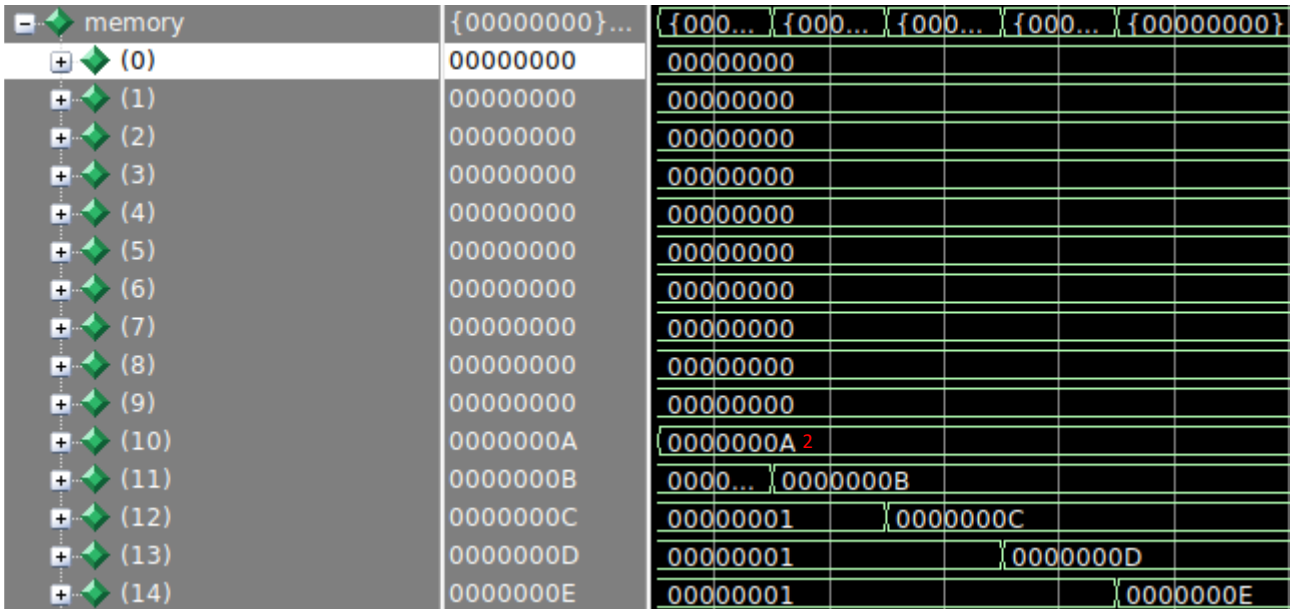

Figura 27 – Processo de escrita na memória, após receber a solicitação de escrita.

5.3.3 Leitura

A Figura 28 exemplifica a recepção de um pacote de solicitação de leitura sendo chegando a interface. Assim como nos outros exemplos, os seis primeiros flits possuem as informações de origem, tamanho do pacote e qual é o serviço, sendo neste caso é 0x00000430 (READ_REQUEST). No entanto, diferente do PERIPHERAL_REQUEST, existem informações necessárias nos próximos flits.

Os destaques (A) e (B) correspondem aos flits que informam o tamanho da informação a ser lida e o endereço inicial ((C) é um valor não usado pelo serviço). Os destaques (1), (2) e (3) ilustram o estado `EMPTY_FLIT`, utilizado na contagem de flits. Periféricos não possuem conhecimento da estrutura do pacote, sendo o papel da interface fazer este gerenciamento.

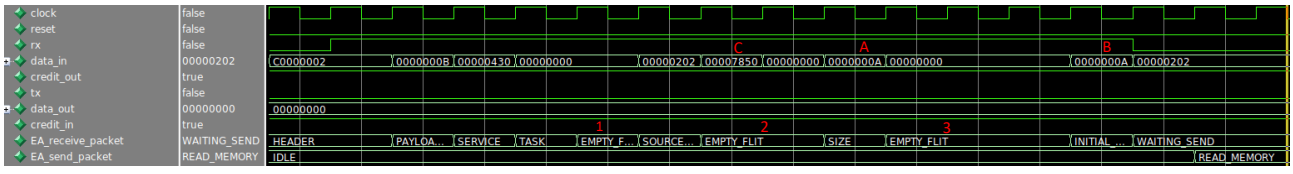

Figura 28 – Pacote de solicitação de leitura chegando na interface de comunicação.

Depois que as informações relativas à leitura dos dados são lidas do pacote, a FSM de recepção entra no estado `WAITING_SEND`, e a FSM de envio é acionada. A Figura 29 mostra a geração do pacote de resposta e como o destaque (1) ilustra, é um serviço de READ_RESPONSE. A Seção 5.1 explica que quando é resposta de uma leitura, antes de começar o envio da resposta, é preciso passar pelo estado `READ_MEMORY` para preencher

o pacote com as informações. O destaque (A) mostra as informações lidas da memória sendo gravadas no pacote.

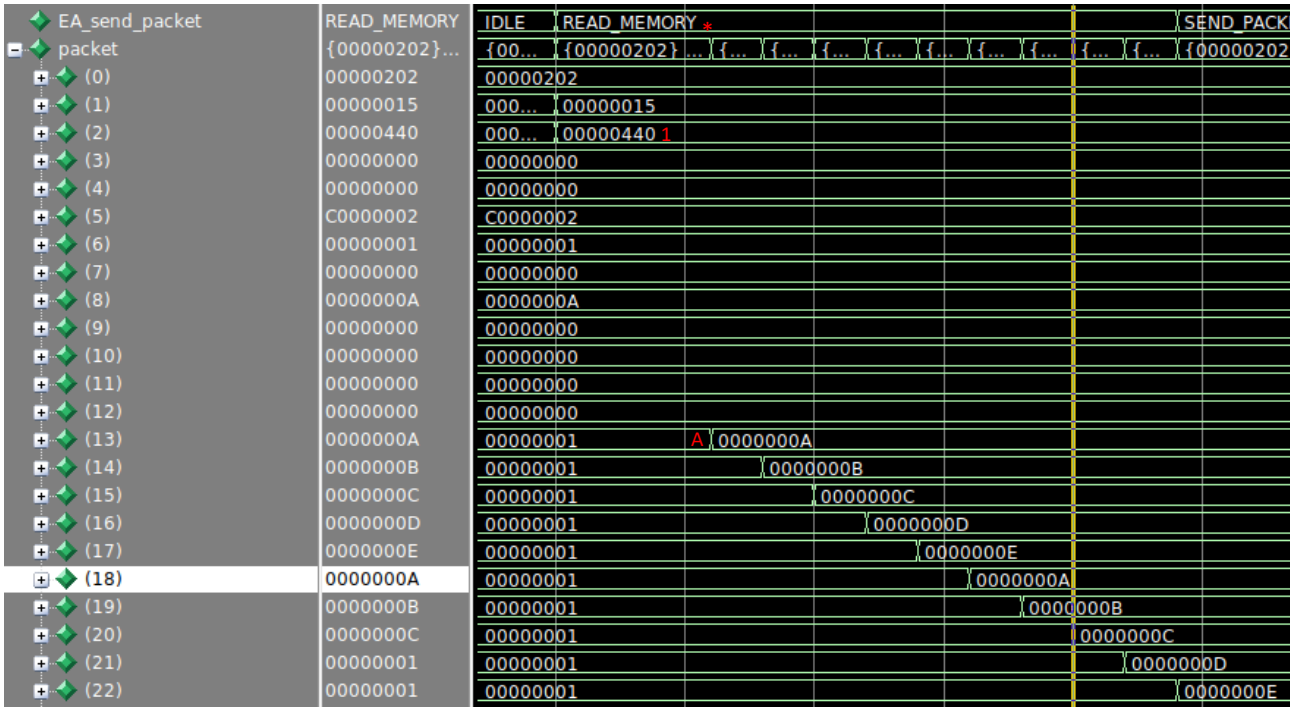

Figura 29 – Geração do pacote de resposta a solicitação de leitura.

A etapa final da leitura é o envio da resposta. A Figura 30 mostra o pacote da Figura 29 sendo enviado da interface para a NoC. O destaque (1) mostra que o tamanho do pacote é 21 (x15), ou seja, tamanho 11 para o *message header* e mais 10 que referente à mensagem lida. O destaque (A) é o flit responsável para dizer o tamanho da mensagem (xA). O destaque * mostra que mesmo que haja perda de crédito durante o envio, após a recuperação do mesmo, a mensagem continua sendo transmitida.

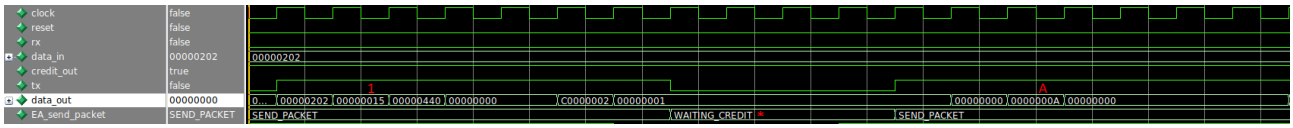

Figura 30 – Geração do pacote de resposta a solicitação de leitura.

5.3.4 Descarte de pacotes

Para exemplificar o descarte de pacotes foi feito uma simulação em um cenário diferente do descrito no início da Seção 4.3. Neste caso tem-se duas tarefas tentando acessar a memória. A interface está conectada ao PE 0x2, enquanto o PE 2x2 tenta solicitar uma conexão.

A Figura 31 mostra a chegada de um pacote com um serviço de solicitação de conexão ao periférico. Neste caso, o periférico já está conectado a outro PE. Por isso, na Figura 32 o periférico está respondendo com um *PERIPHERAL_REQUEST_NACK* (serviço 0x00000420).

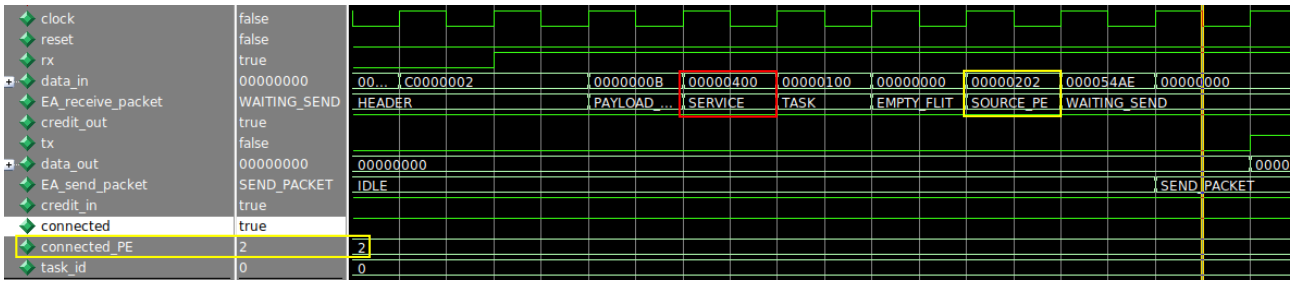

Figura 31 – Pacote de solicitação de conexão chegando na interface, porém já existe conexão ativa.

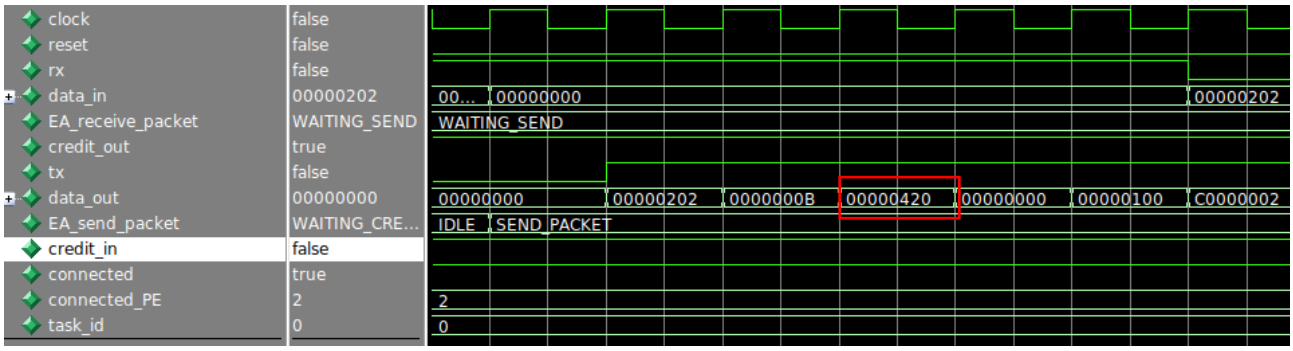

Figura 32 – Envio do pacote de negação de conexão.

Neste cenário, mesmo que o PE receba uma resposta negativa ao pedido de conexão, ele ainda tenta fazer uma leitura na memória do periférico (serviço 0x00000430). A Figura 33 mostra o descarte do pacote, onde a FSM fica no estado *CONSUME*. Isso acontece por que tanto o PE de origem do pacote, quanto o ID da tarefa são diferentes do que estão armazenados na interface. Para que a interface aceite o pacote, ambos os dados precisam ser iguais aos que estão armazenados em sua memória.

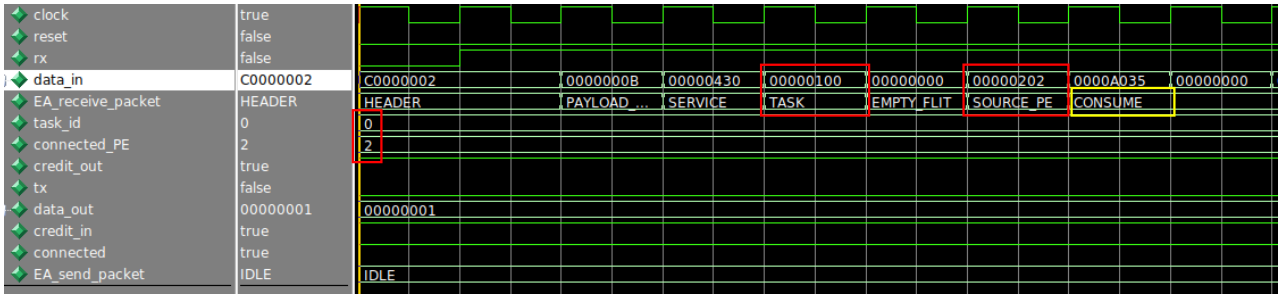

Figura 33 – Pacote sendo consumido sem tratamento, pois não é originado do PE que está conectado.

5.3.5 Liberação

O pedido de liberação, ou *PERIPHERAL_RELEASE*, é detalhado na Figura 34. Observando os valores de (1) e (2), e o estado da FSM de recepção no ponto (2), é possível perceber que o PE que o periférico está conectado é o 2x2 e este é o originador do pedido de liberação. O ID da tarefa, 4º flit é 0, o mesmo que está armazenado na interface de comunicação. Como as informações de origem e tarefa do pacote são equivalentes às que a interface possui, o * destaca o momento que as informações armazenadas são descartadas. A partir deste momento, o periférico está disponível para uma nova conexão.

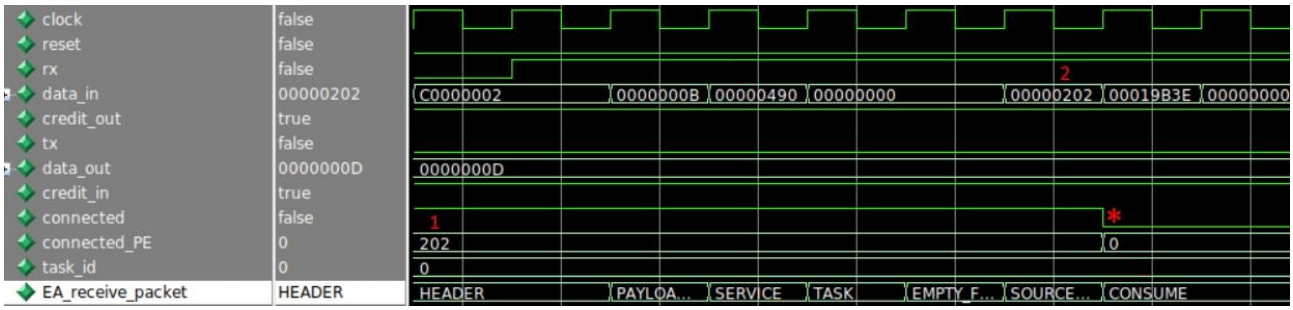

Figura 34 – Pacote de solicitação de liberação chegando a interface de comunicação.

6. CONCLUSÃO

Esta monografia apresentou o desenvolvimento de uma infraestrutura, para o sistema Many-core Memphis realizar a comunicação com periféricos, através de um protocolo padronizado. Para o atendimento a este objetivo foi necessária a implementação de uma interface que fizesse a tradução do protocolo interno do Many-core para o protocolo Wishbone.

O suporte ao acesso do periférico envolveu ações em vários níveis do Many-core. No nível de aplicação, foram disponibilizadas chamadas de sistemas que permitem as tarefas de usuário interagir com um periférico, alocando, utilizando e liberando recursos. Estas chamadas de sistema são tratadas no nível de sistema operacional, que transforma os argumentos passados pelas tarefas em informações enviadas para o periférico. A interface de comunicação implementada assume o papel de intermediária, traduzindo mensagens com protocolo NoC para protocolo Wishbone (e vice-versa), o que permite ao periférico acessar os dados da mensagem e, consequentemente, iniciar uma comunicação com o PE onde a tarefa está executando.

A realização deste trabalho exigiu conhecimentos de diversos conteúdos estudados no curso de Engenharia de Computação. Conteúdos como programação, arquitetura de computadores, sistemas operacionais e periféricos foram essenciais para a execução deste trabalho.

6.1 Trabalhos Futuros

A estrutura simples dos serviços de comunicação com periféricos possui espaço para aperfeiçoamento em diversos aspectos. Os principais aspectos que podem ser destacados são:

- Segurança: adicionar métodos de detecção de integridade dos pacotes.
- Segurança: adicionar criptografia para assegurar que, caso a mensagem essa interceptada no meio da transmissão, o interceptador não seja capaz de extrair as informações do pacote.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Benini, L.; Micheli, G. “ Networks on chips: a new SoC paradigm”, *Computer*, vol. 35–1, 2002, pp. 70–78.

[2] Castilhos, G.; Mandelli, M.; Madalozzo, G.; Moraes, F. “Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes”. In: IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2013, pp. 153–158.

[3] Greenhalgh, P. “ARM big.LITTLE processing with ARM cortex-A15 and A-7”. In: ARM White Paper, 2011, pp. 12.

[4] Hübner, M.; Becker, J. “Multiprocessor System-On-Chip”. Springer, 2010, 1 ed., 270p.

[5] Marchese, T. “Interfaces Seguras para Periféricos em MPSoCs”, Relatório Técnico, Seminário de Andamento. PUCRS-PPGCC, 2019, 50p.

[6] Moore, G. E. “Cramming more componentes onto integrated circuits”, *Eletronics*, vol. 38–8, 1965.

[7] Moraes, F. G.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. “HERMES: an infrastructure for low area overhead packet-switching networks on chip”, *Integration, the VLSI Journal*, vol. 38–1, 2004, pp. 69 – 93.

[8] OpenCores. “WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores”. Capturado em: https://opencores.org/cdn/downloads/wbspec_b3.pdf, 2002.

[9] Ruaro, M.; Caimi, L.; Fochi, V.; Moraes, F. “Memphis: a framework for heterogeneous many-core socs generation and validation”, *Design Automation for Embedded Systems*, vol. 23–3, Aug 2019, pp. 103–122.

[10] Ruaro, M.; Lazzarotto, F. B.; Marcon, C. A.; Moraes, F. G. “DMNI: A specialized network interface for NoC-based MPSoCs”. In: IEEE International Symposium on Circuits and Systems (ISCAS), 2016, pp. 1202–1205.

[11] Venkataramani, V.; Pathania, A.; Mitra, T. “Unified Thread- and Data-Mapping for Multi-Threaded Multi-Phase Applications on SPM Many-Cores”. In: DATE, 2020, pp. 1496–1501.